Miroslaw Malek
Manfred Reitenspieß
Jörg Kaiser (Eds.)

LNCS 3335

Service Availability

First International Service Availability Symposium, ISAS 2004
Munich, Germany, May 2004
Revised Selected Papers

Lecture Notes in Computer Science 3335

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
University of Dortmund, Germany
Madhu Sudan
Massachusetts Institute of Technology, MA, USA
Demetri Terzopoulos
New York University, NY, USA
Doug Tygar
University of California, Berkeley, CA, USA
Moshe Y. Vardi
Rice University, Houston, TX, USA
Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Miroslaw Malek   Manfred Reitenspieß
Jörg Kaiser (Eds.)

Service Availability

First International Service Availability Symposium, ISAS 2004
Munich, Germany, May 13-14, 2004
Revised Selected Papers

Springer

Volume Editors

Miroslaw Malek
Humboldt-Universität Berlin
Institut für Informatik Rechnerorganisation und Kommunikation
Unter den Linden 6, 10099 Berlin, Germany
E-mail: malek@informatik.hu-berlin.de

Manfred Reitenspieß
Fujitsu Siemens Computers
Munich, Germany
E-mail: manfred.reitenspiess@fujitsusimens.com

Jörg Kaiser
University of Ulm
Department of Computer Structures, Faculty of Computer Science
James-Franck-Ring, 89069 Ulm, Germany
E-mail: kaiser@informatik.uni-ulm.de

Library of Congress Control Number: 2004117793

CR Subject Classification (1998): C.2, H.4, H.3, I.2.11, D.2, H.5, K.4.4, K.6

ISSN 0302-9743
ISBN 3-540-24420-4 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11378730 06/3142 5 4 3 2 1 0

Open Specifications for Service Availability™

Manfred Reitenspieß[1], ISAS 2005 General Chair

The continuous availability of services has always been a metric for the success of telecommunications applications: the phone system must always be operational. Today, IP data network providers and enterprise IT departments face the same requirements. Service availability architectures and feature sets have traditionally been highly proprietary and customized to individual telecom equipment provider and application requirements. Each application and hardware platform had to be designed to fit with the specific service availability environment.

Today's market dynamics require companies to be able to raise the bar and meet new and aggressive time-to-market goals. By standardizing the interfaces for high-availability functions and management, the Service Availability Forum aims to create an open, off-the-shelf infrastructure for implementers and operators of highly available services.

The Service Availability Forum is unifying functionality to deliver a consistent set of interfaces, thus enabling consistency for applications developers and network architects alike. This means significantly greater reuse and a much quicker turnaround for the introduction of new products.

As the telecom and IT market recovery accelerates, meeting both functional and time-to-market goals will be essential for success. The Service Availability Forum offers a way forward for maximizing time-to-market advantage through the adoption of a consistent and standardized interface set. The set of open standard software building blocks includes functions for managing the hardware platform components (Hardware Platform Interface), high-availability service functions used by applications (Application Interface Specification), and functions for their associated management (System Management Services).

The International Service Availability Symposium 2004 brought together scientists, technical experts and strategists to discuss availability under a number of aspects:

1. Availability in the Internet and databases
2. High availability based on Service Availability Forum specifications
3. Measurements, management and methodologies
4. Networks of dependable systems
5. Standards and solutions for high availability

The Service Availability Forum is a consortium of industry-leading communications and computing companies working together to develop and publish high availability and management software interface specifications. The Service Availability Forum then promotes and facilitates specification adoption by the industry.

[1] President Service Availability Forum, Fujitsu Siemens Computers, Munich, Germany; manfred.reitenspiess@fujitsu-siemens.com

Program Chair's Message

The 1st International Service Availability Symposium (ISAS 2004) was the first event of its kind where a forum was provided for academic and industrial researchers and engineers who focus on next-generation solutions where services will dominate and their dependability will be expected and demanded in virtually all applications.

As with the birth of a new baby so it was with the first symposium: It was somewhat an unpredictable event and we did not really know how many paper submissions to expect. We were nicely surprised with 28 (including three invited ones), considering the rather specialized topic and short lead time to organize this meeting. We will broaden the scope of the Symposium next year by making it clear that anything that concerns computer services might be worthwhile presenting at ISAS to a good mix of academic and industrial audiences.

A significantly increased interest in dependable services should not be a surprise as we are expecting a paradigm shift where "everything" may become a service. Computer evolution began with data types and formats. Then the concept of objects was discovered and transformed later into components. A set of components (including a set of one as well) forms a service and this concept will dominate computing, ranging from sensor networks to grid computing, for the foreseeable future. In order to make services a viable replacement and/or extension to existing forms of computing they have to be highly available, reliable and secure. The main computer and communication companies, service providers, and academics are searching for innovative ways of increasing the dependability of services that are growing in complexity and will use mainly distributed resources. This trend will continue as computer services are bound to pervade all aspects of our lives and lifestyles. No matter whether we call the computing services of the future "autonomic," "trustworthy" or simply "reliable/available" the fact of the matter is that they will have to be there seven days a week, 24 hours a day, independent of the environment, location and the mode of use or the education level of the user. This is an ambitious challenge which will have to be met. Service availability cannot be compromised; it will have to be delivered. The economic impact of unreliable, incorrect services is simply unpredictable.

All submissions were subject to a rigorous review process. Hence only 15 papers were accepted. Unfortunately, many good, worthwhile manuscripts did not make it into the program due to the high quality threshold set up by the Program Committee. Each paper was reviewed by three Program Committee members. I would like to thank wholeheartedly our PC members whose hard work was exemplary. Those who spent time at the virtual PC meeting deserve an additional recognition. Our paper selection went extremely smoothly thanks to the tremendous effort of the reviewers and solid support from my secretary Sabine Becker and my Ph.D. student Nikola Milanovic of Humboldt University Berlin. Also, Prof. Joerg Kaiser from the University of Ulm deserves a special credit

for editing the symposium's proceedings and preparing the Springer volume of Lecture Notes in Computer Science. I thank all of them very much. And last but not least I would like to express my gratitude to Manfred Reitenspieß whose involvement and support were very helpful throughout the program preparation process.

The attendees enjoyed the final program as well as the lively presentations, got involved in many heated discussions, struck up new frienships, and, hopefully, got inspired to contribute to next year's symposium to be held in Berlin on April 25-26, 2005.

Munich, May 13, 2004

ISAS 2004 Program Chair
Miroslaw Malek
Humboldt–Universität Berlin
Institut für Informatik
malek@informatik.hu-berlin.de

Table of Contents

ISAS 2004

Architecture of Highly Available Databases
Sam Drake, Wei Hu, Dale M. McInnis, Martin Sköld, Alok Srivastava, Lars Thalmann, Matti Tikkanen, Øystein Torbjørnsen, Antoni Wolski 1

Data Persistence in Telecom Switches
S.G. Arun 17

Distributed Redundancy or Cluster Solution? An Experimental Evaluation of Two Approaches for Dependable Mobile Internet Services
Thibault Renier, Hans–Peter Schwefel, Marjan Bozinovski, Kim Larsen, Ramjee Prasad, Robert Seidl 33

OpenHPI: An Open Source Reference Implementation of the SA Forum Hardware Platform Interface
Sean Dague 48

Quality of Service Control by Middleware
Heinz Reisinger 61

Benefit Evaluation of High–Availability Middleware
Jürgen Neises 73

A Measurement Study of the Interplay Between Application Level Restart and Transport Protocol
Philipp Reinecke, Aad van Moorsel, Katinka Wolter 86

Service–Level Management of Adaptive Distributed Network Applications
K. Ravindran, Xiliang Liu 101

A Methodology on MPLS VPN Service Management with Resilience Constraints
Jong–Tae Park, Min–Hee Kwon 118

Higly–Available Location–Based Services in Mobile Environments
Peter Ibach, Matthias Horbank 134

On Enhancing the Robustness of Commercial Operating Systems
Andréas Johansson, Adina Sârbu, Arshad Jhumka, Neeraj Suri 148

A Modular Approach for Model–Based Dependability Evaluation of a Class of Systems
Stefano Porcarelli, Felicita Di Giandomenico, Paolo Lollini, Andrea Bondavalli 160

Rolling Upgrades for Continuous Services
Antoni Wolski, Kyösti Laiho 175

First Experience of Conformance Testing an Application Interface Specification Implementation
Francis Tam, Kari Ahvanainen 190

On the Use of the SA Forum Checkpoint and AMF Services
Stéphane Brossier, Frédéric Herrmann, Eltefaat Shokri 200

Author Index 213

Architecture of Highly Available Databases

Sam Drake[1], Wei Hu[2], Dale M. McInnis[3], Martin Sköld[4], Alok Srivastava[2], Lars Thalmann[4], Matti Tikkanen[5], Øystein Torbjørnsen[6], and Antoni Wolski[7]

[1] TimesTen, Inc, 800 W. El Camino Real, Mountain View, CA 94040, USA
drake@timesten.com
[2] Oracle Corporation, 400 Oracle Parkway, Redwood Shores, CA 94065, USA
{wei.hu, alok.srivastava}@oracle.com
[3] IBM Canada Ltd., 8200 Warden Ave. C4/487, Markham ON, Canada L6G 1C7
dmcinnis@ca.ibm.com
[4] MySQL AB, Bangårdsgatan 8, S-753 20 Uppsala, Sweden
{mskold,lars}@mysql.com
[5] Nokia Corporation, P.O.Box 407, FIN-00045 Nokia Group, Finland
matti.j.tikkanen@nokia.com
[6] Sun Microsystems, Haakon VII gt 7B, 7485 Trondheim, Norway
oystein.torbjornsen@sun.com
[7] Solid Information Technology, Merimiehenkatu 36D, FIN-00150 Helsinki, Finland
antoni.wolski@solidtech.com

Abstract. This paper describes the architectures that can be used to build highly available database management systems. We describe these architectures along two dimensions – process redundancy and data redundancy. Process redundancy refers to the management of redundant processes that can take over in case of a process or node failure. Data redundancy refers to the maintenance of multiple copies of the underlying data. We believe that the process and data redundancy models can be used to characterize most, if not all, highly available database management systems.

1 Introduction

Over the last twenty years databases have proliferated in the world of general data processing because of benefits due to reduced application developments costs, prolonged system life time and preserving of data resources, all of which translate to cost-saving in system development and maintenance. What makes databases pervasive is a database management system (DBMS) offering a high-level data access interface that hides intricacies of access methods, concurrency control, query optimization and recovery, from application developers. During the last ten years generalized database systems have also been making inroads into industrial and embedded systems, including telecommunications systems, because of the significant cost-savings that can be realized.

As databases are deployed in these newer environments, their availability has to meet the levels attained by other components of a system. For example, if a total system has to meet the 'five nines' availability requirements (99.999%), any single component has to meet still more demanding requirements. It is not unusual to require that the database system alone can meet the 'six nines' (99.9999%) availability

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 1 – 16, 2005.
© Springer-Verlag Berlin Heidelberg 2005

requirement. This level of availability leaves only 32 seconds of allowed downtime over a span of a year. It is easy to understand that under such stringent requirements, all failure-masking activities (switchover, restart etc.) have to last at most single seconds rather than minutes. Such databases are called Highly Available (HA) Databases and the systems to facilitate them are called highly available database management systems (HA-DBMS).

An HA-DBMS operates in a way similar to HA applications: high availability is achieved by *process redundancy*—several process instances are running at the same time, typically, in a hardware environment of a multi-node cluster. In addition to one or more *active processes* (Actives) running a service, there are *standby processes*, or redundant active processes, running at other computer nodes, ready to take over operation (and continue the service), should the active process or other encompassing part fail (Standbys). Database processes involve data whose state and availability is crucial to successful service continuation. Therefore we talk about *data redundancy*, too, having the goal of making data available in the presence of failures of components holding the data. Models of process and data redundancy applied in highly available databases are discussed in this paper.

Product and company names that are used in this paper may be registered trademarks of the respective owners.

2 HA-DBMS for Building Highly Available Applications

In addition to the database service itself, a highly available database brings another advantage to the HA framework environment. Just as a traditional database system frees developers from mundane programming of data storage and access, an HA-DBMS frees the developers of HA applications from some low level HA programming. To illustrate this, let us have a look at two situations. In Fig. 1, an application is running in an HA framework such as the SA Forum's Availability Management Framework (AMF) [1].

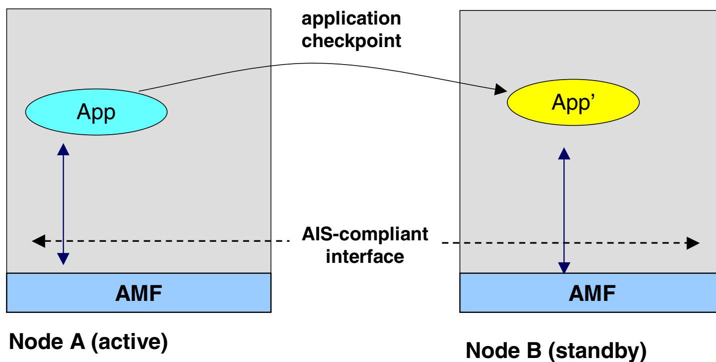


Fig. 1. An application running within AMF

Assume that the application is run in an active and a standby component (process). The application components are SA-aware meaning that they are connected to AMF in a way following the SA Forum Application Interface Specification (AIS) [1].

One demanding aspect of HA programming is to make sure that the application state is maintained over failovers. To guarantee this, application checkpointing has to be programmed into the application. The SA Forum AIS, for example, offers a checkpoint service for this purpose. Decisions have to be made about what to checkpoint and when. Also the code for reading checkpoints and recovering the application states after a failover has to be produced.

Another situation is shown in Fig. 2. In this case, the application uses the local database to store the application state, by using regular database interfaces.

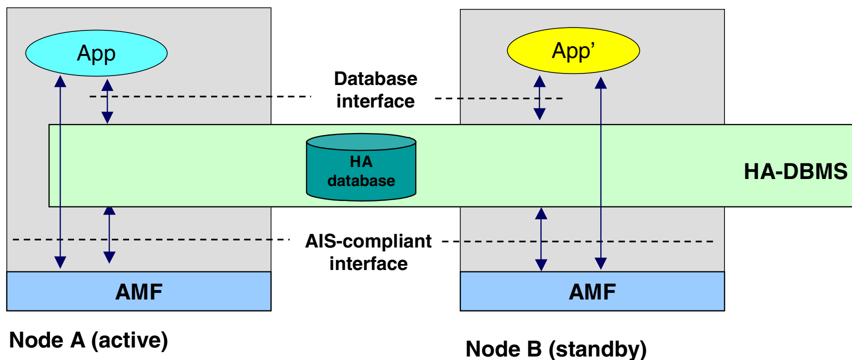


Fig. 2. A database application running within AMF

Because we deal with an HA-DBMS here, the latest consistent database state is always available after the failover at the surviving node. It is the database that does all the application checkpointing in this case. All this happens in real time and transparently. Additionally, as database systems operate in a transactional way preserving atomicity and consistency of elementary units of work (transactions), the database preserves transactional consistency over failovers, too. This way, an HA application programmer is freed from complex checkpoint and recovery programming. By bringing another level of abstraction into high-availability systems, HA-DBMS makes it easier to build highly available applications.

It should be noted, however, that the situation pictured in Fig. 2 is not always attainable. The application may have hard real-time (absolute deadlines) or soft real-time latency requirements that cannot be met by the database. Failover time of the database may be a limiting factor, too, if failover times below 100 ms are required. Finally, the program state to be preserved may not yield to database storage model. Nevertheless, the more application data is stored in a database, the more redundancy transparency is achieved.

3 HA Database Redundancy Models

Highly available database systems employ a number of redundancy concepts. All HA-DBMSs rely on having redundant database processes. When a database process dies (e.g., due to node failure), another database process can take over service. To provide correctness, each redundant process must see the same set of updates to the database. There are basically two means of ensuring this: one technique, replication, relies on the database processes to explicitly transfer updates among each other. Depending on the implementation, each replica can store its copy of the data either in main-memory or on disk. Replication is not exclusively done between individual databases. In distributed databases, one database is managed by several database processes on different nodes, with possible intra-database replication between them.

An alternate means for ensuring that all the redundant database processes see the same set of updates to the database is to rely on a shared disk system in which all the processes can access the same set of disks. Since all the processes can access the same set of disks, the database processes do not need to explicitly replicate updates. Instead, all the processes always have a single, coherent view of the data. Note that a shared disk system also has redundancy. However, it is built-in at lower levels — e.g., via RAID or by network-based remote mirroring.

The two approaches introduced above may be mapped to two known general DBMS architectures: *shared-nothing* and *shared-disk* [8], respectively. In this paper we take a more focused point of view on DBMS architectures: we concentrate exclusively on means to achieve high availability.

Several redundancy models are possible in an HA-DBMS and these are defined below. We distinguish between *process redundancy* which defines availability of the database processes and *data redundancy* which specifies, for replication-based solutions, the number of copies of the data that are explicitly maintained. Both process redundancy and data redundancy are necessary to provide a HA Database Service.

3.1 Process Redundancy

Process redundancy in an HA-DBMS allows the DBMS to continue operation in the presence of process failures. As we'll see later, most process redundancy models can be implemented by both shared-disk and replication-based technologies.

A process which is in the *active* state is currently providing (or is capable of providing) database service. A process which is in the *standby* state is not currently providing service but prepared to take over the active state in a rapid manner, if the current active service unit becomes faulty. This is called a *failover*. In some cases, a new type of process, a *spare process* (or, Spare) may be used. A spare process may be implemented as either a running component which has not been assigned any workload or as a component which has been defined but which has not been instantiated. A spare may be elevated to Active or Standby after proper initialization.

Process redundancy brings the question of how (or if) redundancy transparency is maintained in the HA-DBMS. Of all running processes, some may be active (i.e. providing full service) and some not. In the case of failovers active processes may change. The task of finding relevant active processes may either be the responsibility of applications, or a dedicated software layer may take care of redundancy transparency.

3.2 Data Redundancy

Data redundancy is also required for high availability. Otherwise, the loss of a single copy of the data would render the database unavailable. Data redundancy can be provided at either the physical or the logical level.

3.3 Physical Data Redundancy

Physical data redundancy refers to relying on software and/or hardware below the database to maintain multiple physical copies of the data. From the perspective of the database, there appears to be a single copy of the data. Some examples of physical data redundancy include: disk mirroring, RAID, remote disk mirroring, and replicated file systems.

All these technologies share the common attribute that they maintain a separate physical copy of the data at a possibly different geography. When the primary copy of the data is lost, the database processes use another copy of the data. These technologies can differ in terms of the failure transparency that is supported. Disk mirroring and RAID, for example, make physical disk failures completely transparent to the database.

Physical data redundancy is frequently combined with process redundancy by using a storage area network. This allows multiple nodes to access the same physical database. If one database server fails (due to a software fault or a hardware fault), the database is still accessible from the other nodes. These other nodes can then continue service.

3.4 Logical Data Redundancy Using Replication

Logical data redundancy refers to the situation where the database explicitly maintains multiple copies of the data. Transactions applied to a primary database D are replicated to a secondary database D' which is more or less up-to-date depending on the synchrony of the replication protocol in the HA Database. In addition to inter-database replication, intra-database replication is used in distributed database systems to achieve high availability using just one database. Note that we speak about replication in general terms since the replication scheme is vendor specific (based on the assumption that both database servers are from the same vendor). The replication can be synchronous or asynchronous, be based on forwarding logs or direct replication as part of the transaction, transactions can be batched and possibly combined with group commits. The method chosen depends on the database product and the required level of *safeness* [2]. With a *1-safe* replication ("asynchronous replication") transactions are replicated after they have been committed on the primary. With a *2-safe* replication ("synchronous replication") the transactions are replicated to the secondary, but not yet committed, before acknowledging commit on the primary. With a *2-safe committed* replication transactions are replicated and committed to the secondary before acknowledging commit on the primary. In the *very safe* replication all operations but reads are disabled if either the primary or the secondary becomes unavailable. An overview 1-safe and 2-safe methods is given in [14]. Various optimizations are proposed in [5],[4], [10] and [21]. Although most of the work on safeness-providing methods has been done in the context of remote backup, the results are applicable to in-cluster operation too.

4 Data Redundancy Models

For the rest of this paper, data redundancy refers to *logical* data redundancy. It represents the number of distinct copies of data that are maintained by the database processes themselves via replication. It does not count the copies that may be maintained by any underlying physical data redundancy models. For example, two copies of the data that is maintained by a disk array or by a host-based volume manager would be counted as one copy for the sake of this discussion, while two copies of the data maintained by the database would count as two. Note that in both cases, the loss of one copy of the data can be handled transparently without loss of availability.

We discuss data redundancy models in detail first because this is an area that is fairly unique to HA-DBMSes.

4.1 Database Fragments, Partitioning, and Replication of Fragments

To define the data redundancy models we need to define what we are actually replicating, i.e. *database fragments*[1]. Database *fragmentation* is a decomposition of a database D into fragments $P_1 ... P_n$ that must fulfill the following requirements:

1. *Completeness*. Any data existing in the database must be found in some fragment.
2. *Reconstruction*. It should be possible to reconstruct the complete database from the fragments.
3. *Disjointness*. Any data found in one fragment must not exist in any other fragment[2].

The granularity of a fragment is typically expressed in terms of the data model used. In relational databases, fragments may be associated with complete SQL schemas (called also catalogs) or sets of tables thereof. The lowest granularity achieved is usually called *horizontal* or *vertical* fragmentation where “horizontal” refers to dividing tables by rows and “vertical”—by columns. Note that this definition of fragmentation does not exclude viewing the database as one entity if this is a required logical view of the database.

A non-replicated, *partitioned database* contains fragments that are allocated to database processes, normally on different cluster nodes, with only one copy of any fragment on the cluster. Such a scheme does not have strong HA capabilities. To achieve high availability of data, *replication of database fragments* is used to allow storage and access of data in more than one node. In a *fully replicated* database the database exists in its entirety in each database process. In a *partially replicated* database the database fragments are distributed to database processes in such a way that copies of a fragment, hereafter called *replicas*, may reside in multiple database processes.

In data replication, fragments can be classified as being *primary* replicas (Primaries) or *secondary* replicas (Secondaries). The primary replicas represent the actual

---

[1] Fragment is a generalization of the common definition of table fragmentation in relational databases.

[2] This normally applies to horizontal fragmentation, but it does not exclude vertical fragmentation if we consider the replicated primary key to be an identifier of data instead of data itself.

data fragment[3] and can be read as well as updated. The secondary replicas are at most read-only and are more or less up to date with the primary replica. Secondary replicas can be promoted to primary replicas during a *failover* (see section 0).

4.2 Cardinality Relationships Among Primaries and Secondaries

1*Primary/1*Secondary
Here every fragment has exactly one primary replica which is replicated to exactly one secondary replica. This is a very common redundancy model since two replicas has been found adequate for achieving high-availability in most cases.

1*Primary/Y*Secondary
Here every fragment has exactly one primary replica and is replicated to a number of secondary replicas. This model provides higher availability than 1*Primary/1*Secondary and allow for higher read accessibility if secondary replicas are allowed to be read.

1*Primary
Here every fragment exists in exactly one primary replica. This model does not provide any redundancy at the database level. Redundancy is provided below the database by the underlying storage. It is used in shared disk systems and also in centralized or partitioned databases.

X*Primary
Here every fragment has a number of primary replicas and is used in N*Active process redundancy models (sometimes called multi-master). This model allow for higher read and update accessibility than 1*Primary if the same fragment is not attempted to be updated in parallel (since this would lead to update conflicts).

4.3 Relationships Between Databases and Fragments

Non-partitioned Replicated Database
The most common case is when the database and the fragment are the same. Consequently, the whole database is replicated to the Secondary location (Fig. 3). NOTE: all cases in this subsection are illustrated assuming the 1*Primary/1*Secondary cardinality.

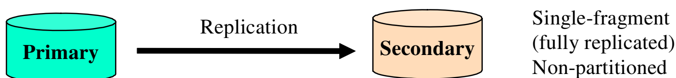


Fig. 3. Non-partitioned database

Partitioned Replicated Database
In this model, there are fragments having the purpose of being allocated to different nodes or of being replicated to different nodes (Fig. 4).

[3] If a primary replica is not available then the fragment is not available, thus the database is not available.

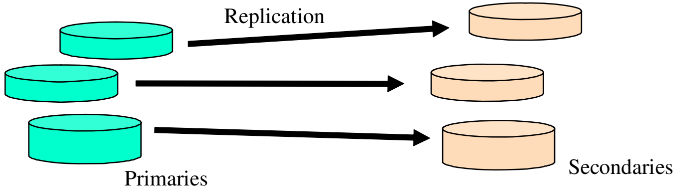


Fig. 4. Partitioned database

Mixed Replicated Fragments
A special case of a partitioned database is a database with mixed partitions whereby a database may host both Primaries and Secondaries. A special case is two databases with symmetric fragments (Fig. 5).

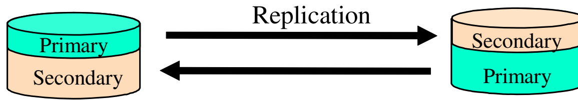


Fig. 5. Two databases with symmetric fragments

5 Process Redundancy Models

5.1 Active/Standby (2N)

Active/Standby (sometimes referred to as 2N) is a process redundancy model for HA-DBMS that is supported by both replication and shared-disk systems. Each active database process is backed up by a standby database process on another node. In Fig. 6, a replication-based example is shown while Fig. 7 provides a shared-disk based example. All updates must occur on the active database process; they will be propagated via replication, or via a shared disk, to the standby database process.

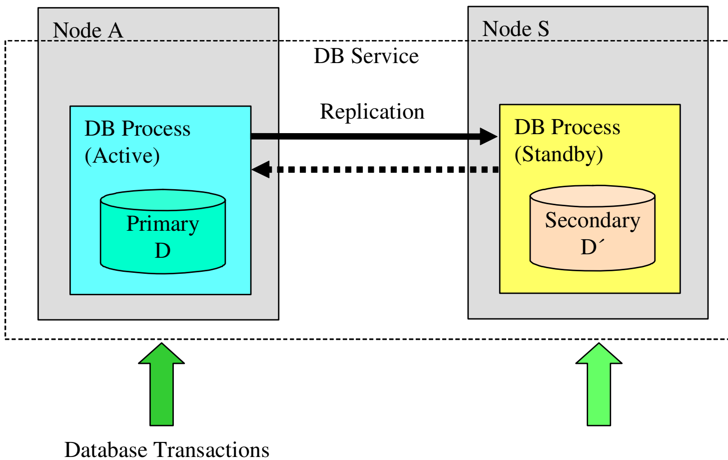


Fig. 6. Active/Standby Redundancy Model using Replication

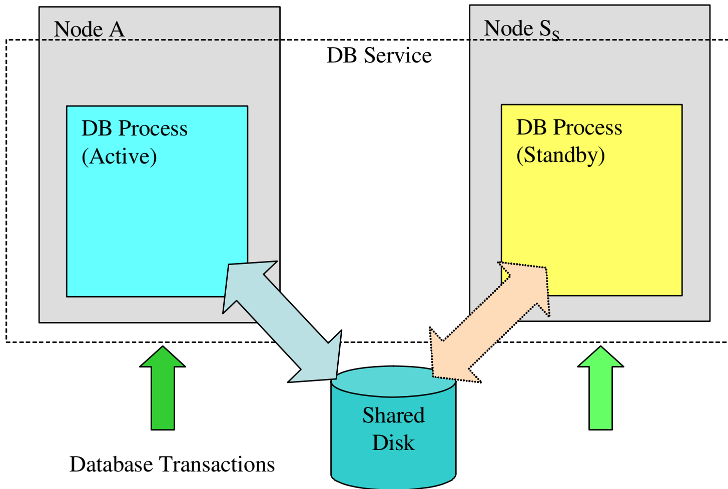


Fig. 7. Active/Standby Redundancy Model using Shared Disk

In the case of a failure of the active database process (for any reason such as software fault in the database server or hardware fault in the hosting node) the standby database process will take over and become the new active database process (Fig. 8). If the failed database process recovers it will now become the new standby database process and the database processes have completely switched roles (Fig. 9). If the HA Database has a *preferred active* database process it can later switch back to the original configuration.

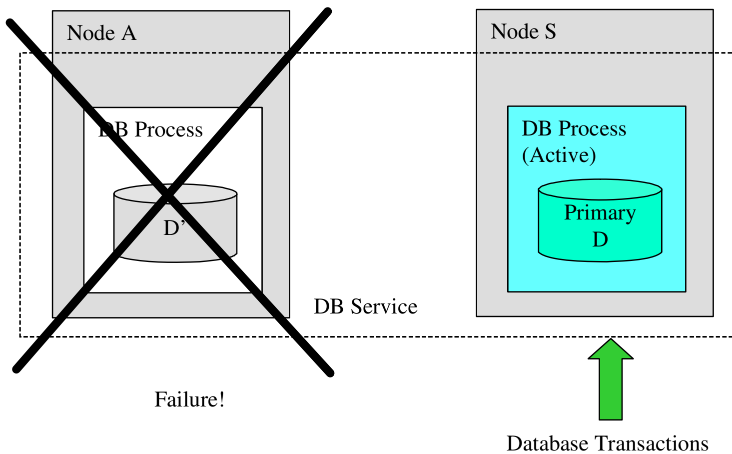


Fig. 8. Failure of Active Primary, Switchover

The standby database process can be defined as more or less ready to take over depending on the chosen safeness level and the HA requirements of the applications. To classify the non-active database processes we separate between *hot standby* and *warm standby*.

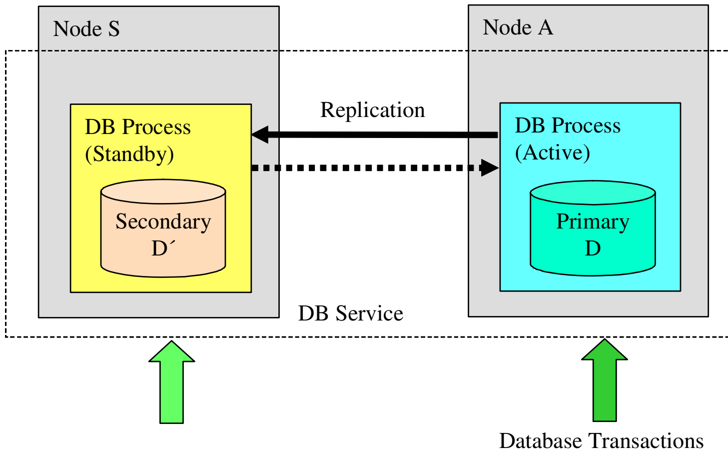


Fig. 9. Reversed Roles

Hot Standby

One active database process is being backed up by a standby database process that is ready to more or less instantly (in sub-second time) take over in case the active database process fails. The applications can already be connected to the standby or be reconnected to the standby (now active).

Warm Standby

One active database process is being backed up by a standby database process that is ready to take over after some synchronization/reconnect with applications in case the active database process fails. In this case, the failover may last from few tens of seconds to few minutes.

In the next section we introduce spares and we distinguish between standbys and spares since it is possible to have a model Active/Standby/Spare.

There are many commercial incarnations of active/standby HA database systems. In Oracle Data Guard [12] and MySQL replication [11], the active primary database ships transactions to one or more standby databases. These standby databases apply the transactions to their own copies of the data. Should the primary database fail, one of these standby databases can be activated to become the new primary database. Oracle Data Guard also supports both synchronous and asynchronous log shipping along with use selectable safeness level ranging from 1-safe to very-safe. The Carrier Grade Option of the Solid Database Engine [16] also uses an active-standby pair with a fully replicated database and dynamically controlled safeness level.

5.2 Active/S*Spare

Active/S*Spare (one Active and S Spares) is a configuration in which several *spare* database processes are pre-configured on some node(s). It is supported both by shared disk systems and replicating systems. An example of a shared-disk based architecture with a spare process is shown below (Fig. 10).

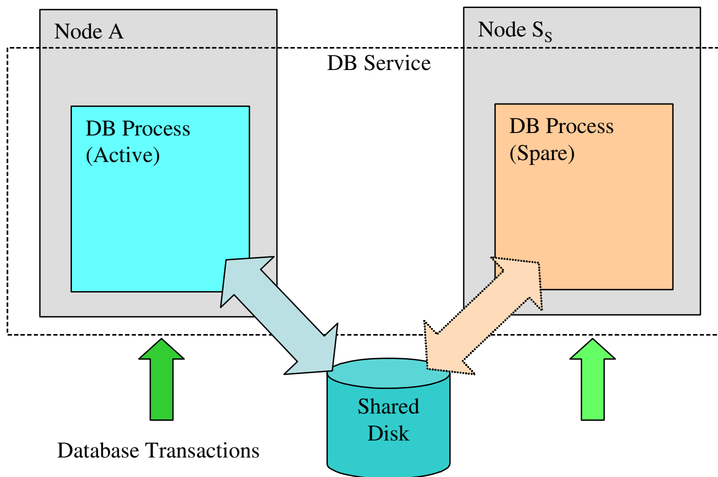


Fig. 10. Active/S*Spare Redundancy Model using Shared Disk

In a shared-disk database, if the active database process fails, the shared disk containing the database files is mounted on the node hosting the Spare (a spare node), theSpare becomes initialized with the database, and the database becomes active on that node. If the failed node restarts it will now become a new spare node. The nodes have therefore completely switched roles. If the HA Database has a preferred active database process it can later switch back to the original configuration.

In a replicating HA-DBMS, the Spare gets the database before becoming Active. The level of availability offered by this model is lower that that of Active/Standby because of the additional time needed to initialize the Spare.

This kind of operation represents the model that is supported by commercially available clustering frameworks such as Sun Cluster [17] and IBM HACMP [7]. These clustering frameworks support all the major DBMSes.

5.3 N*Active

In larger clusters, the database system can utilize more than two nodes to better use the available processing power, memory, local disks, and other resources of the nodes, to balance the load in a best possible way. In the N*Active (sometimes referred to as N-Way Active) process redundancy model, N database processes are active and applications can run transactions on either process. Here all processes support each other in case of a failure and each can more or less instantly take over. All committed changes to one database process are available to the others and vice versa. In shared-disk systems, all the database processes see the same set of changes. In a replication-based system, all changes are replicated to all the processes.

The database is fully available through all database processes and all records can be updated in all processes. In case of simultaneous conflicting updates, copy consistency may be endangered, in a replicating system. This is taken care of with a distributed concurrency control system (e.g. lock manager) or a copy update reconciliation method. In a shared disk system, the database infrastructure may be simpler because the data

objects are not replicated. The database internally implements a lock manager to prevent conflicting updates to the same data. Fig.11 shows a shared disk based N*Active model. Note that we used 2 nodes as an example even though the model supports more than 2 nodes. Fig. 12 shows a replication-based 2-node N*Active model.

There are several commercial implementations of N*Active HA database systems. The Oracle Real Application Clusters [13] is an example of an N*Active configuration whereby all the instances of the database service are active against a single logical copy of the database. This database is typically maintained in a storage-area-network (SAN) attached storage. The HADB of Sun ONE [18][6] uses also the N*Active approach that can be applied to the whole database of fragments thereof. MySQL Cluster [3][19] has an N*Active configuration in which the processes are partitioned into groups. Each operation of a transaction is synchronously replicated and committed on all processes of a group before the transaction is committed. MySQL Cluster provides a 2-safe committed replication if the group size is set to two.

N*Active configurations have demonstrated scalability with real applications. SAP, for example, has certified a series of Oracle Real Application Clusters-based SAP SD Parallel Standard Application benchmark that demonstrates near linear scalability from 1 through 4 nodes [15]. In the TPCC benchmark, a 16-node Oracle Real Application Clusters demonstrated 118% of the throughput of a single multiprocessor that contains the same number of CPUs[20].

If the database is not fully replicated and there are mixed fragments in all databases, the process model is always N*Active. For example, in Fig. 13, a 2*Active HA-DBMS is shown utilizing symmetric replication. With symmetric replication, concurrency control problems are avoided and the advantage of load balancing is retained.

Unlike most N*Active environments, in Fig. 13, the partitioning scheme is visible to the application and is often based on partitioning the primary key ranges. The applications are responsible for accessing the correct active process. Inter-partition transactions are normally not supported.

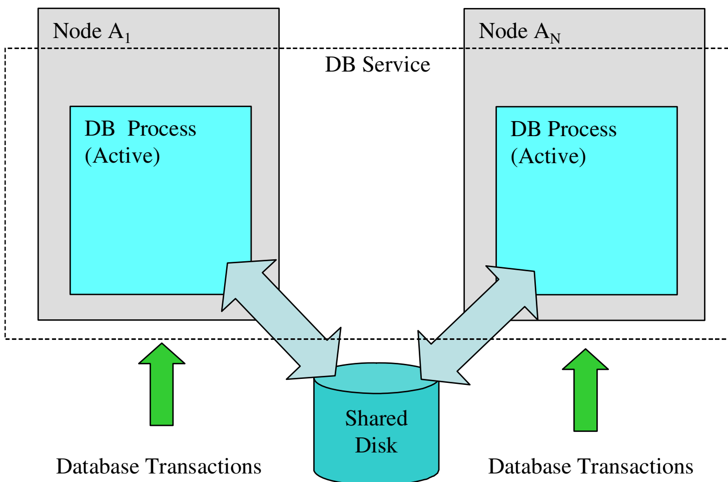


Fig. 11. N*Active, Shared Disk

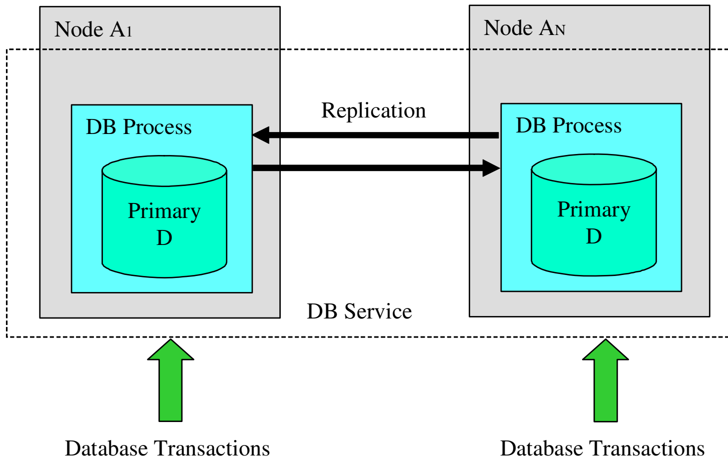


Fig. 12. N*Active, Full Replication, Redundancy Model

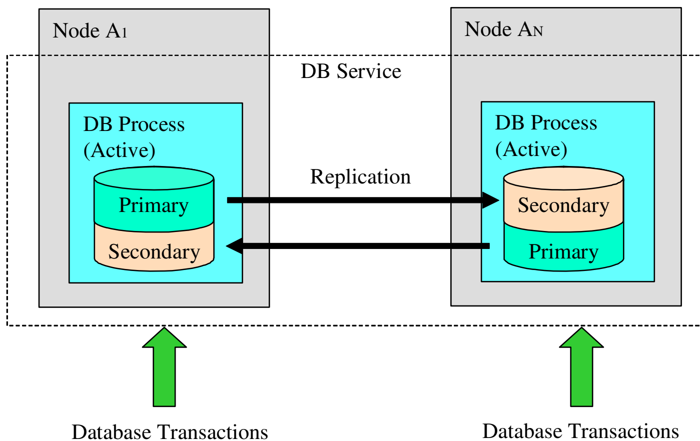


Fig. 13. A 2*Active symmetric replication database system

5.4 N*Active/S*Spare

N*Active/S*Spare (N times Active and S times Spare) is a variant of Active/S*Spare where N Active database processes provide availability to a partitioned database. As in the N*Active model, the active processes may rely on a shared disk, may use fully replicated databases or mixed fragments (partially replicated databases).

An example of a partially (symmetrically) replicated database with Spares is shown in Fig.14.

Each database process maintains some fragments of the database and Spare processes can take over in case of failure of active database processes. A Spare must get the relevant fragments of the active database process at startup.

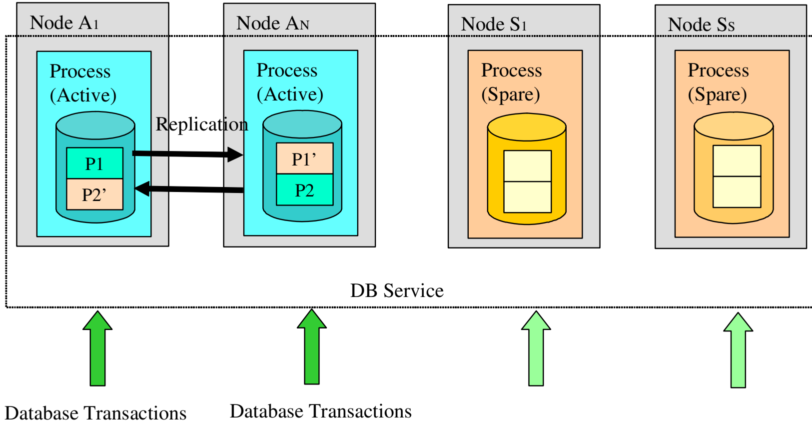


Fig. 14. N*Active/S*Spare Redundancy Model, Partially Replicated Database

5.5 Other Redundancy Models

Some systems combine multiple redundancy models to achieve different degrees of data and process redundancy. M-standby, cascading standby and geographically replicated N*active clusters [9] are several examples. Since they are composed of the other redundancy models presented in this paper, they will not be further discussed.

6 Application View

Applications may or may not be aware of the redundancy models used by various components of the database system. In Active/Standby configurations, applications normally need to be aware so that they can connect to the active instance. Moreover, different aspects of the database system may in fact use different redundancy models. For example, a database management system may have one set of processes that manage data, and another set of processes that execute queries and to which client applications connect. These sets of processes may have completely different redundancy models, and may communicate with each other in various ways.

A related topic is the partitioning scheme. In general, it is better to hide the application from the actual partitioning of the data fragments. This allows the application to remain unchanged should the partitioning scheme be changed due to planned or unplanned reconfigurations. Keeping the partitioning scheme internal to the database server allows for internal load balancing and reorganization of data without affecting applications. For performance reasons some systems provide some concept of locality and support for co-locating applications and data on the same node. This can sometimes be controlled through “hints” from the applications to the database server about where data is most effectively stored. Logical partitioning schemes for both applications and data are often combined with common load balancing schemes built into distributed communication stacks.

Products from Oracle, Sun, and MySQL maintain the process distribution transparency with various approaches.

7 Summary

Database management systems are critical components of highly available applications. To meet this need, many highly available database management systems have been developed. This paper describes the architectures that are internally used to construct these highly available databases. These architectures are examined from the perspective of both process redundancy and logical data redundancy. Process redundancy is always required; it refers to the maintenance of redundant database processes that can take over in case of failure. Data redundancy is also required. Data redundancy can be provided at either the physical or the logical level. Although both forms of data redundancy can provide high availability, this paper has concentrated on logical data redundancy since that is a case where the database explicitly manages the data copies. We believe that process and data redundancy are useful means to describe the availability characteristics of these software systems.

References

1. Application Interface Specification, SAI-AIS-A.01.01, April 2003. Service Availability Forum, available at www.saforum.org.
2. Gray, J. and Reuter, A.: Transaction Processing Systems, Concepts and Techniques. Morgan Kaufmann Publishers, 1992.
3. How MySQL Cluster Supports 99.999% Availability. MySQL Cluster white paper, MySQL AB, 2004, available at http://www.mysql.com/cluster/.
4. Hu, K., Mehrotra, S., Kaplan, S.M.: Failure Handling in an Optimized Two-Safe Approach to Maintaining Primary-Backup Systems. Symposium on Reliable Distributed Systems 1998: 161-167.
5. Humborstad, R., Sabaratnam, M., Hvasshovd, S-O., Torbjørnsen, Ø.: 1-Safe Algorithms for Symmetric Site Configurations. VLDB 1997: 316-325.
6. Hvasshovd, S., et al.: The ClustRa Telecom Database: High Availability, High Throughput and Real-time Response. VLDB 1995, pp. 469-477, September 1995.
7. Kannan, S. et al.: Configuring Highly Available Clusters Using HACMP 4.5. October 2002, available at http://www.ibm.com.
8. Norman, M.G., Zurek, T., Thanisch, P.: Much Ado About Shared-Nothing. SIGMOD Record 25(3): 16-21 (1996).
9. Maximum Availability Architecture (MAA) Overview. Oracle Corporation, 2003, available at http://otn.oracle.com/deploy/availability/htdocs/maa.htm.
10. Mohan, C., Treiber, K., Obermarck, R.: Algorithms for the Management of Remote Backup Data Bases for Disaster Recovery. ICDE 1993: 511-518.
11. MySQL Reference Manual. MySQL AB, 2004, available at http://www.mysql.com/documentation/.
12. Oracle Data Guard Overview. Oracle Corporation, 2003, available at http://otn.oracle.com/deploy/availability/htdocs/DROverview.html.
13. Oracle Real Application Clusters (RAC) Overview, Oracle Corporation, 2003, available at http://otn.oracle.com/products/database/clustering/.
14. Polyzois, C.A., Garcia-Molin, H.: Evaluation of Remote Backup Algorithms for Transaction-Processing Systems. ACM Trans. Database Syst. 19(3): 423-449 (1994).

15. SAP Standard Applications Benchmarks, SD Parallel, June 2002, available at http://www.sap.com/benchmark/.
16. Solid High Availability User Guide, Version 4.1, Solid Information Technology, February 2004, available at http://www.solidtech.com.
17. Sun Cluster 3.1 10/03 Concepts Guide, Sun Microsystems, Part No. 817-0519-10, October 2003.
18. Sun ONE Application Server 7 Enterprise Edition – Administrator's Guide, Sun Microsystems, Part no. 817-1881-10, September 2003.
19. Thalmann, L. and Ronström, M.: High Availability features of MySQL Cluster. MySQL Cluster white paper, MySQL AB, 2004, available at http://www.mysql.com/cluster/.
20. Transaction Processing Performance Council TPC-C Benchmarks, December 2003, available at: http://www.tpc.org.
21. Wiesmann, M., Schiper, A.: Beyond 1-Safety and 2-Safety for Replicated Databases: Group-Safety. EDBT 2004: 165-182.

Data Persistence in Telecom Switches

S.G. Arun

Force Computers, Bangalore, India
Arun.sg@fci.com

Abstract. This paper explores the need for data persistence in Telecom Switching applications, the different kinds of data that need persistence and the use of databases to achieve this persistence for providing High Availability. Depending on the complexity, architecture and capacity of the telecom application, the data persistence also needs suitable architecture. Some of these applications and corresponding database models for data persistence in the control plane are described in this paper, with some performance reviews.

1 Introduction

Today it is a given that any telecom product that is to be commercially deployed must be carrier grade. A general understanding of carrier grade product is that, at a minimum, the system must meet the "5 Nines" availability criterion. This means that a system can be out of service for at most six minutes in a year, or must be available 99.999% of the time. This includes planned and unplanned down time. Telcos demand this kind of performance simply because the cost of down time is high both in terms of lost revenue and the image or the credibility with the customers, or even regulatory requirements.

In this paper, we start with the concept of High Availability (HA) and how it leads to data persistence. We see the choices that we have in accomplishing the data persistence and some standards prevalent in this field. Then we see some typical telecom architectures and how database can be integrated to meet the HA needs of each type of architecture. Finally, we see some performance numbers to see the real world performance of database based persistence models.

1.1 HA and Related Concepts

Closely related to the concept of HA are the concepts of fault coverage and Reliability. Fault coverage determines how the design of the system takes care of all expected and possible faults or things that can go wrong. On the other hand, it is recovering from unknown faults that determine how good the HA criteria are met.

Reliability is whether the system runs without a fault. For example, a system might crash for a millisecond every week, so its reliability is not high, whereas its availability is better than 5 Nines. On the other hand, another system might

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 17–32, 2005.
© Springer-Verlag Berlin Heidelberg 2005

not crash at all for 6 months and then be down for 2 weeks. Here the reliability is high, "HA-ness" is low.

Now we are closing in on what is meant by HA. It is the probability that the system is available at any given sampling time. Reliability is the determinism of the system. Fault coverage, or the handling of faults is necessary for both, but not sufficient for HA.

1.2 Aspects of Application HA

Data from IEEE [1] shows that a significant amount of downtime is caused by software. Contrary to the earlier myth that throwing more hardware at the system makes it more robust, data shown below shows that this is not the case. What we end up with more hardware is more cost, not HA. Clearly, the focus areas need to be on the software and upgrade / maintenance times, that account for the maximum downtime.

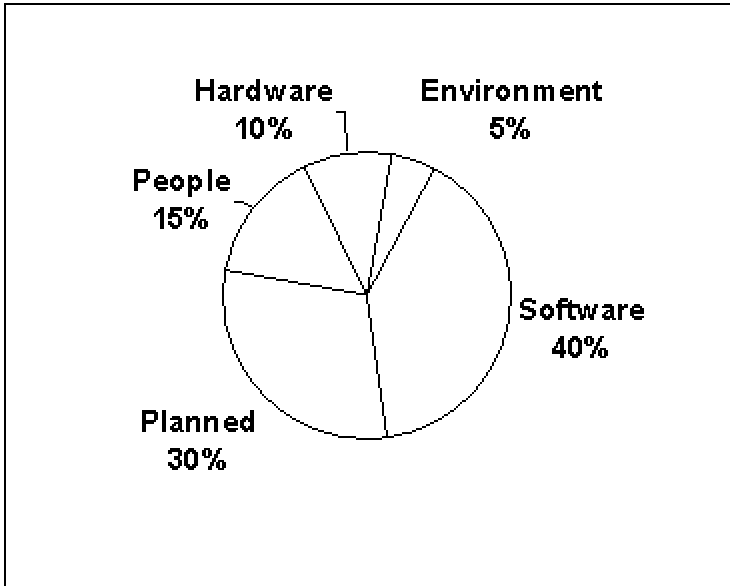


Fig. 1. IEEE data on system Failure

At the application level, the Service Availability Forum (SAF) defines the state of the application within the context of ITU's X.731 state management[2]. This defines the application to have states that enable us to determine the capability to provide service, or capability to take over service in case of failure.

However, when a standby application takes over, there is a need to know "where" the previously active application stopped (or crashed). This is where the concept of checkpointing comes in. SAF has standardized this interface too.

A typical telecom switch consists of three operational planes, or the context of the application: control, data and management plane. The control plane executes various signaling and routing functions that result in dynamically configuring the data plane. The data plane transports the user or bearer traffic. The management plane provides an administrative interface into the overall system, and is responsible to keep the system healthy and operational.

There are several other aspects of the application level fail over that need consideration and these are addressed in the SAF as well. Some of them are:

1. Event services framework.
2. Cluster management framework.
3. Messaging framework.
4. Data management framework

The data management framework within the context of a telecom switching application is the focus of the current paper and is limited to control plane.

2 Data Persistence

2.1 Need for Data Persistence

With the states of the application being monitored and managed by a distributed mechanism, further discussion assumes that there is a well-defined procedure for the fail over of the application process. What is needed along with the state management is to be able to determine the current calls already established by the failed application, before it stopped or crashed.

From the Telco's perspective, it is very important to not drop existing calls, as this is a customer visible fault. If an existing call is dropped, not only is the customer billed for the call that he is not able to complete, there is also an added irritation of the need to re-dial. Thus the first and perhaps the most important criterion for an application is that the existing calls be held, which means that all information related to the existing calls needs to be persisted. The second and derived requirement is that the persisted data be available as soon as possible to the back up process on fail over of the active process.

There have been traditionally three classifications of achieving HA, namely hotstandby, warm standby and cold standby. Hotstandby is when the back up is as good as the primary and is already in a form of pseudo-service, except for a minimal switching needed by the Automatic Protection Switching (APS), which is typically of the order of few tens of milliseconds. Cold standby is when the data as well as the application process need to first come up to speed to the state of the active process, before being capable of providing service. Warm stand by is when the standby process is already running, but needs some time to synchronize with the active application. While hotstandby is clearly the preferred choice, it comes at a cost in that there needs to be a designated hotstandby for every service or process(or) that needs this kind of fail over mechanism.

2.2 What Kind of Data?

In the context of telecom switch, we have seen that there are three operational planes. Now let us look more closely at the role of each of these planes and the kind of data that is present in each domain.

The control plane, which is also called the call processing, contains and manipulates data that typically impacts the service and hence needs persistence. For example, a media gateway would need to persist the source IP and port numbers, destination IP address and port numbers, PSTN line number identifiers or Circuit Identifier Code (CICs) and time slot numbers, codec types, bit rates, some Call reference number, timestamp of start of the call etc. Not having persistence for call data means that HA cannot be accomplished.

Apart from the call specific states, there is also a need to persist the management plane data. This includes, for example the number of current alarms in the system. This is needed for recovery action to take place in spite of a fail over. Also needed are other data like the audit timers, time based jobs etc. While this kind of non-call data is not immediately visible to the customer, yet it is important, because if not handled, they eventually lead to system crashes and hence unplanned down time. We currently focus on the control plane HA in this paper.

2.3 How Much Data?

In simple terms, the data persistence requirements are clearly dictated by the system capacity. One measure of the system is easily the number of ports in the system and this determines the amount of data that might need persistence. Another important measure is the throughput, which is the amount of data that needs to be persisted per second. This performance number represents the number of times the persisted data needs to be updated, deleted or new data inserted.

This has bearing on the Busy Hour Call Attempts (BHCA) specified for the system. Clearly, the same switch installed in a metro network has a different BHCA specification than if it is installed in a rural or semi-urban network. For a given system, in a given network, the BHCA is specified as the maximum number of calls that are attempted on the system in an hour.

To get a feel for some numbers, let us work through a simple example and start with a network element that is specified at 6000 ports. Assuming that the average hold time is 120 seconds (other times are ignored in this simple example), also the offered Erlang is 0.8 per line (which shows an extremely high loading), then the system is expected to handle

$$\left(\frac{6000X60X0.8}{3600X2}\right) = 40 \tag{1}$$

40 calls per second. This is then the number of times call related data must be persisted. If we have some figure like about 150 bytes that need to be persisted per call, then we have 150X40 = 6000 bytes per second to be persisted.

On the other hand, if we are looking at the distributed system, then the call densities are much higher. A typical switch capacity could be around 60,000 ports. This means that for every second, there are about

$$\left(\frac{60000X60X0.8}{3600X2}\right) = 400 \tag{2}$$

400 calls per second. Now we find that there is a need to persist about 60,000 bytes of data per second.

This kind of call related data need persistence for the duration of the call. Should there be fail over any time, the persisted data enables the application to continue without customer visibility, thus ensuring HA. At the end of each call, the call related data could be deleted from the persistence, unless needed for other administrative purposes, like backing on tape etc. In any case, the data deleted from the persistence reserved for HA is not necessary for HA anymore after the call has terminated. In this paper, the model does not take into account the backing up procedures nor the load that it may momentarily place on the processor.

3 Data Persistence Models

Here we see some of the common data persistence models necessary for different telecom architectures, driven among other factors, by the system capacity. After a quick look at database as persistence tool and some commonly used terminology, we then look at the usage of database in typical telecom architectures.

3.1 Database as Persistence Tool

As one of the primary goals of providing HA is interoperability with multiple vendors, it is essential that the architecture conform to some standard. In this case, the relevant standard is the SAF, which defines the HA in terms of the interfaces, leaving room for individual implementers to provide their own specific architectures.

See chapter 7 of Service Availability^{TM} Forum Application Interface Specification SAI-AIS-A.01.01 [3] for more details about the check pointing interfaces and requirements. The Service Availability checkpoint service provides an efficient facility for software processes to record checkpoint data incrementally. Using the checkpoint service to record checkpoint data protects the application against failures. After a fail over procedure, the checkpoint service is used to retrieve the previous checkpoint data and resume execution from the state recorded before the failure, minimizing the impact of the failure.

While SAF provides the specifications, it is left to the implementation to realize the checkpoint services in an efficient manner to meet the needs of the telecom application.

It is possible to implement the checkpoint services at a purely application level service, as shown in the figure below. Reference [4] shows such an implementation.

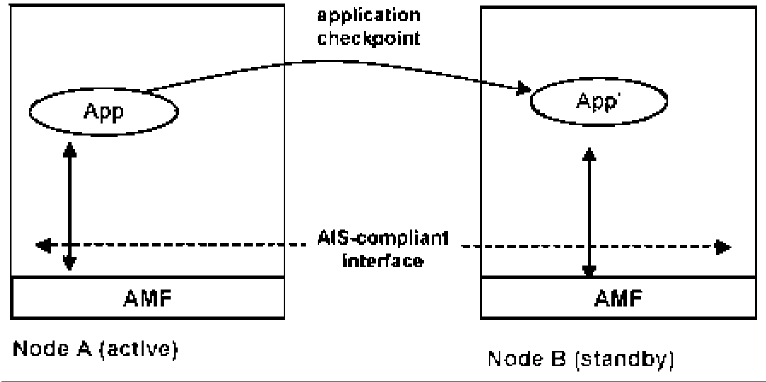


Fig. 2. Application level Check Pointing service

AMF = Availability Management Framework, AIS = Application Interface Specification

In this scenario, the application is responsible for the check pointing and it has to be built into the design and implementation. Producing and reading check points, along with sufficient protection, handling multiple clients or consumers of the data etc has to be managed by the application itself, which is a non-trivial part of the overall telecom architecture and effort.

On the other hand, using a standard database approach to manage the persistence has the advantages that the check pointing, both locally and over the network can be transparently taken care of by the database. See the figure below for a typical implementation of the database version of the check pointing services.

One of the major advantages of using a regular database is the usage of the concept of "transactions" that preserve the atomicity of each work unit and the

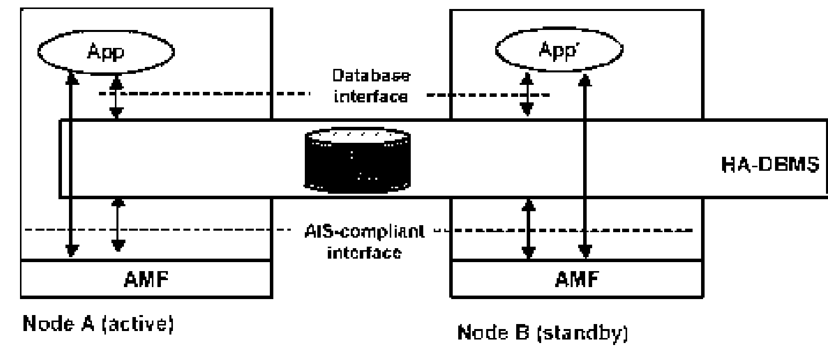


Fig. 3. Check Pointing using Database

consequent operations that are possible like rollback, commit on a transaction that would otherwise be a significant part of the application development.

Before we investigate common telecom switching architectures, let us briefly summarize some terminology used in the database domain and how they map into the context of this discussion.

Catalog: A catalog logically partitions a database so that data is organized in ways that meet business or application requirements. Each logical database is a *catalog* and contains a complete, independent group of database objects, such as tables, indexes, procedures, triggers, etc. In the current scenario, a catalog can represent an instance of call processing (see later sections) and the call data required thereon.

Tables: This is a unit of database that can logically exist on its own; in other words, a table contains all information that is complete by itself. For example, there can be a table of all connections, called connection table.

Hotstandby: A second database server that is linked to the first and is ready to take over in case the first fails. The secondary server has an exact copy of all committed data that is on the primary server.

Replica: The replica database is the storage of local data and saved transactions. All replica data, or a suitable part of it, can be refreshed from the master database whenever needed by subscribing to one or multiple publications. The local data includes data of the business applications, typically a subset of the master database, as well as system tables that contain information specific to the particular database.

Diskless: The Diskless option allows replicas to run in processors that do not have hard disks, such as line cards in a network router or switch. A diskless server is almost always a “replica” server. This gives the highest performance in terms of the data storage capability.

Transaction: This is an indivisible piece of work, that is either completely accomplished or not at all. There may be multiple statements within each transaction, but unless all of them are completed, a transaction is said to be not complete.

Transaction Log: This is one of the ways of protecting data. This log tracks what pieces of the transaction have been completed and incomplete transaction can be rolled back based on the transaction log, in case the transaction is not able to complete. There are variants of logging like strict, adaptive and relaxed. These are relevant to the disk version of the database and refer to the degree of data safety. In strict version, the transaction data is logged into disk after every transaction. A database like SOLID’s BoostEngine [5] uses a REDO log to roll back transactions.

Commit: To commit a transaction is to make the transaction permanent in the database. A transaction is completed when it is committed. A two-safe commit ensures that the secondary has the data, even if the primary node were to go down.

Having familiarized with the commonly used database terminology and how they apply to the telecom switching domain, let us now look at some of the more

common telecom switching architectures and how database can help achieve the data persistence.

3.2 Centralized Control Plane

This architecture is suitable for small and medium sized telecom switching applications, where the expected call density is not very high. In the centralized architecture, the central control processor (CP) hosts the call processing that is connected to the external network element. This in turn drives the traffic processor (TP) whose role is limited to the carrying traffic data. From a data perspective, all the data is at the control processor and this then acts a source for the data that needs to be persisted. Here we see that a single tier of database is sufficient to meet the requirements of persistence.

In this scenario, the CP as a result of call processing (STEP 1), sets up the data path and communicates the connection details to the relevant TP (STEP2). After the TP completes the path set up of the D-plane, it acknowledges back to

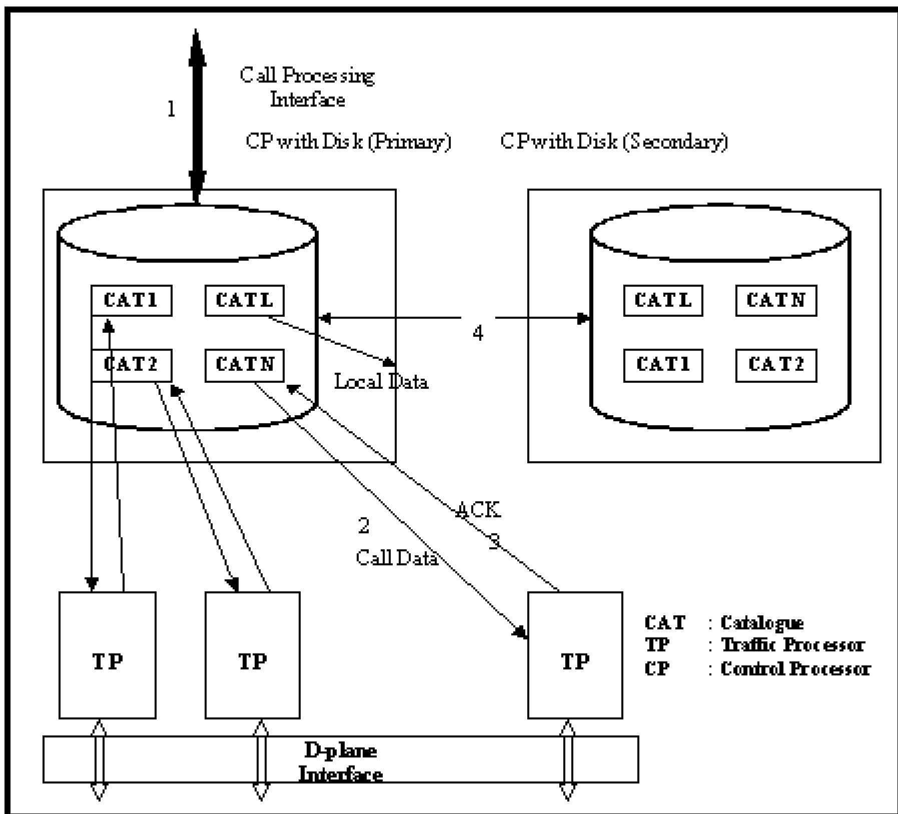


Fig. 4. Data Persistence Model for Centralized C-plane

the CP about the success of the operation (STEP3). At this point, the details of the successful call is persisted in the local database and also sent to the hotstandby for persistence (STEP4). This is when the transaction (or the call details) is committed. Note that separate catalogues are maintained for each of the traffic processor and that the TP itself has no call data records and all the data is available at the control processors. To avoid a single point of failure, it is recommended that the 2-safe acknowledgement protocol between primary and secondary with adaptive or relaxed logging be used. Only in cases where there is a need to avoid catastrophic failures where both the primary and secondary nodes go down, it is necessary to have strict logging.

As a result of this operation, what we have is that each call that is successful has the data stored in the CP as well as the stand by CP. This means that this architecture is capable of supporting a failure of a TP and a CP. In case of failure of a TP, the persisted data on the CP needs to be provided to another TP before the TP is capable of continuing. Clearly, hotstandby of the TP is not possible, and there may be a user visible break in the data path. This may be acceptable for some data-only applications, but is not acceptable for applications that involve voice. However, the control path is provided with a higher level of HA in the sense that the standby processor already has the call data and needs no further time to start being operational.

3.3 Distributed

a. Primary with Replica. This model is necessary when the call density is high and the single call control processor or centralized approach cannot handle the processing load. The solution is to distribute the load across many processors. In the figure, there may optionally be a load balancer, which is not shown. Note that in this approach, the distributed processor (DP) is responsible for both the control plane as well as the data plane.

From a data persistence perspective, we see that in this two-tier model, the CP has the master database and each DP maintains the call data in its own replica. For each call transaction (call successfully set up) in DP, call data is pushed from the replica to the master. The replica allows a commit, subject to approval from the master, which checks for consistency. In this case, since each DP owns a catalogue, it is easy to see that there will normally not be any contention issues. Being in-memory of the DP, performance is enhanced. An asynchronous store and forward protocol guarantees that the data is backed into the master database and is completely backed up or not at all, thus ensuring consistency from the application perspective. Clearly, here we see the advantages of using a database vis-à-vis an application level check pointing.

In this architecture, the role of the CP is restricted to provide the data persistence along with other management functions like network management, provisioning etc and does not participate in the call processing. The Catalogues in the figure represent the call data stored as a consequence of completed or stable calls.

In this model, there are a few disadvantages. For example, every transaction has to be sent to CP from each DP and the load on CP would be quite high.

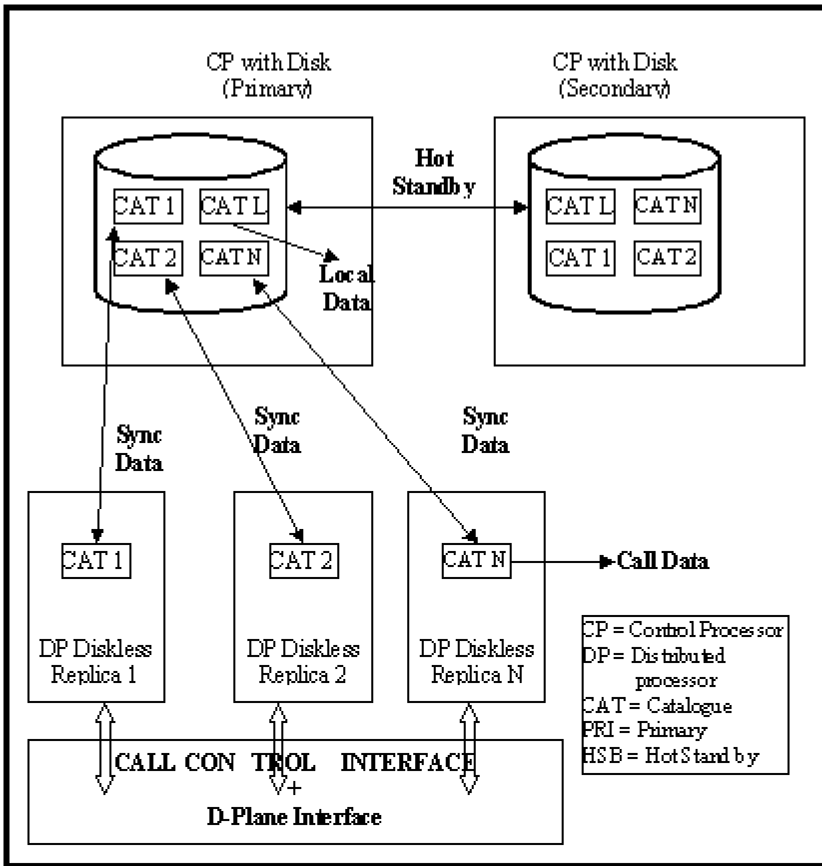


Fig. 5. Data Persistence for Distributed C-Plane, Option 1

Another implication is that this model inherently does not support hotstandby operation. This is because the primary database is centrally stored and any standby of the DP necessarily has to be initialized with the state of the faulty DP. So, at best, a warm stand by for the DPs can be achieved.

On the positive side, for a given number of blades in a chassis, this provides the maximum number of ports possible, as there are no dedicated or designated DPs in hotstandby and hence all the active DPs can be in a load-sharing mode. Alternatively, there can be some DPs in a standby pool, one of which can take over the role of a failed DP, but after getting the state information. Assuming that the failed DP was handling about 2000 calls, this means getting about 2000 X 150 bytes = 300KB of data over the network from the CP, which can mean a fail over time of a few seconds to few tens of seconds. Clearly, data path HA is limited in this architecture to only cold or possibly warm standby, whereas the control path HA is provided with no calls getting dropped. Based on the application (data-only or voice), the limitation on data path can be acceptable or not.

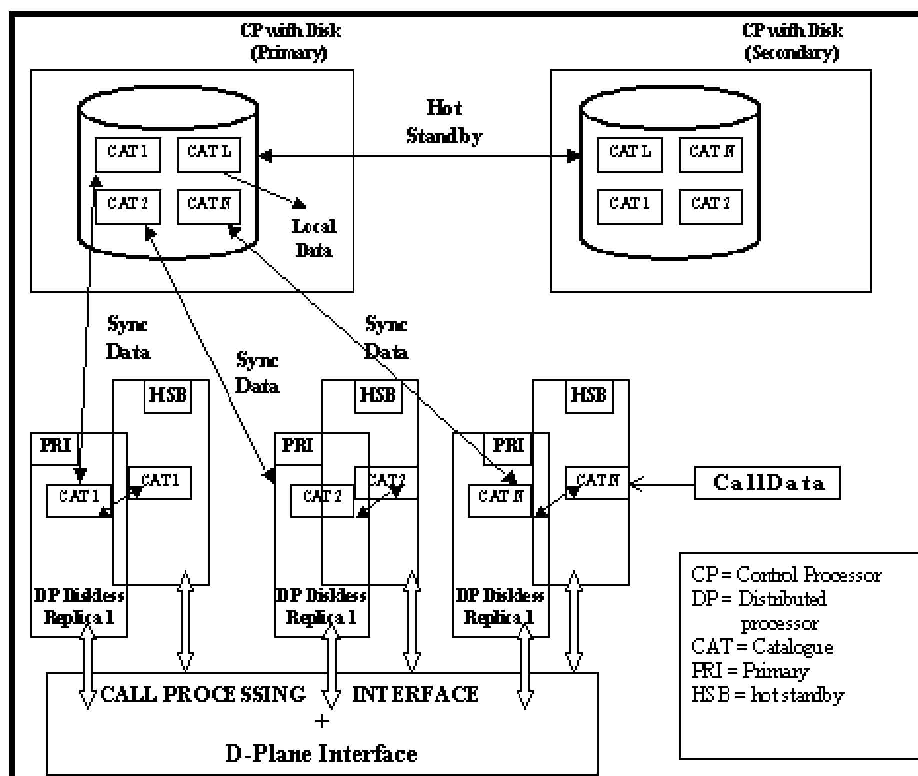


Fig. 6. Data Persistence for Distributed C-Plane, Option 2

The local catalogue can be some management catalogue like the alarms that are currently active, or the timer based jobs that are pending etc. relevant only to CP.

b. Central Primary, Replica with Hotstandby. This architecture is used in the case of high-density switches. Here call processing is distributed across the processors, with the result that there is no single point where the global data is available. The main difference with the previous architecture is that there is a designated hotstandby processor for each of the active data processor. This architecture enables the hotstandby of the data processor allowing data plane fail over as well as control plane fail over.

Hence the data persistence model is different, because the data is generated and consumed locally, without recourse to a central database.

In this architecture, the local DP maintains a diskless version of the database catalogues that contain the call details. This is also backed on designated hotstandbys for the individual processors. On fail over of the active processor, the hotstandby has a ready copy of the database to continue the processing from where the active processor failed and there is no network traffic generated for

this purpose at the time of fail over. Another advantage is that the load on control processor would be much reduced as the replicas send data periodically (could be in the order of seconds or tens of seconds).

The replication policy can be re-configured in the event of a replica not having a standby. The disadvantages of this architecture are that additional resources (CPU/memory) for maintaining a secondary replica are needed and since it is the responsibility of a replica to "sync-up", CP is at the mercy of the replicas and may not have control over replication interval. Some databases (for example, BoostEngine from SOLID Information Technology) do provide the flexibility for replication intervals.

A catastrophic failure only happens when an active DP and the active CP fail at the same time. However, the standard expectation for HA in the current context is limited to a single point of failure, not what is called "double fault" scenarios. The hotstandby of the CP is also in synchronism with the active CP, so any individual failure of either the DP or the CP is handled gracefully in this architecture.

The fact that there is a hotstandby for the DP means that there is a possibility of providing a data path HA as well. This feature then depends on whether there

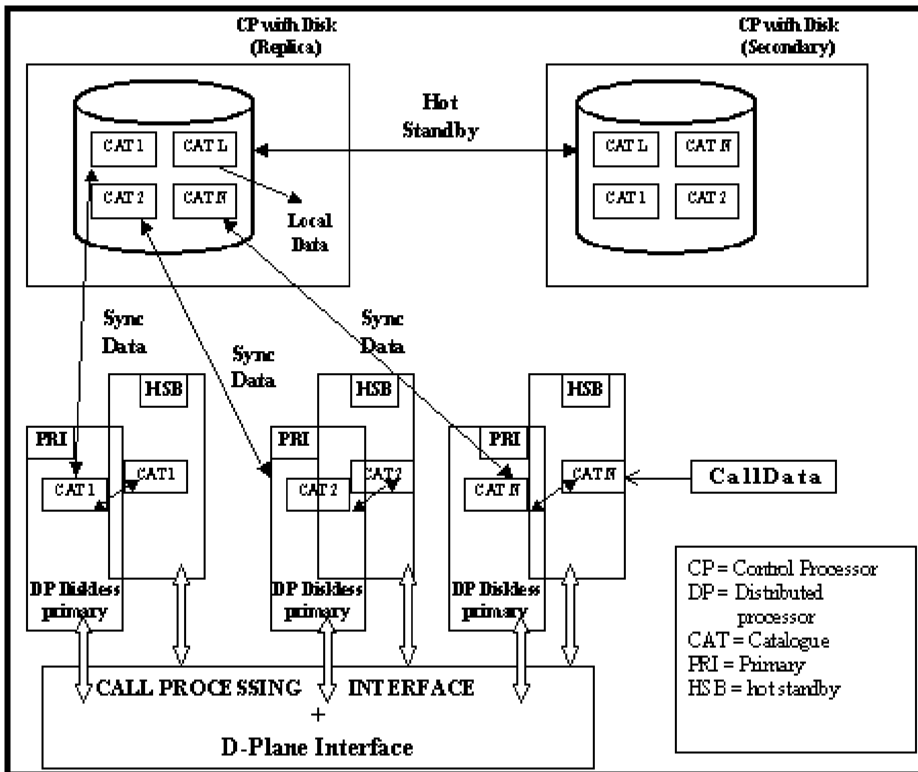


Fig. 7. Data persistence for Distributed C-plane, Option 3

is support from the lower layers. For example, providing an automatic protection switch (APS) controlled by the fail over mechanism ensures that when the DP has switched, the corresponding I/O is also switched.

c. Central Replica, Distributed Masters with Hotstandbys. This is similar to the previous architecture in terms of functionality. The major difference here is that the individual DP has the master database and the CP only has the replicas of the individual DP. In other words, this is a multi-master configuration.

The motivation for this design is that the "call data" is primarily generated and consumed by a DP and hence it is appropriate to designate this data as "master". Each DP maintains the call data in a master and this has a standby. For each call transaction in DP, data is pushed from the primary to its secondary. Periodically, the CP replicas pull data from each DP. Some of the advantages of this model are that the load on CP would be much reduced as the replicas send data periodically (could be of the order of minutes) and in general, a replica initiates the replication ("pull" model) and so the CP can exercise control over the synchronization interval by having the replicas. The main disadvantage is that additional resources (CPU/memory) for maintaining a secondary replica. Some of the central maintenance functions in the CP also persist data (like currently active alarms) etc need for management functions need to be prevented from reaching the active DP, as this central data is irrelevant in the DP's context. This may need additional replication policy changes.

4 Reference Implementation of Database HA

If a database is used in a system for providing data persistence, it is essential that the database itself also be HA compatible. While there is no special effort needed in terms of the data itself, what is required is the state management and fail over strategies, which is what we discuss in this section. In other words, here we briefly look at process redundancy of the database itself.

Reference implementation is done on Force Computers' EndurX ^{TM} system for telecom switching applications. The event agent is an entity that works on the publish-subscribe model and is responsible for notifications of all events to registered subscribers. The fault manager is responsible for handling of the events and in a typical implementation, is a rule based entity, that has a defined response to each of the event received by it or that is expected to be handled by it.

The DB monitor is a watchdog processes that heartbeats with the database and is capable of state management of the database. The DB server could be for example, a process that has been built with the database library, represents the database in the figure and is the interface seen by the core call processing application as well as the management application. The DB monitor heart beats the database and notifies the event agent of the health of the database. The event agent and fault management are application processes that operate on the data provided by the DB monitor.

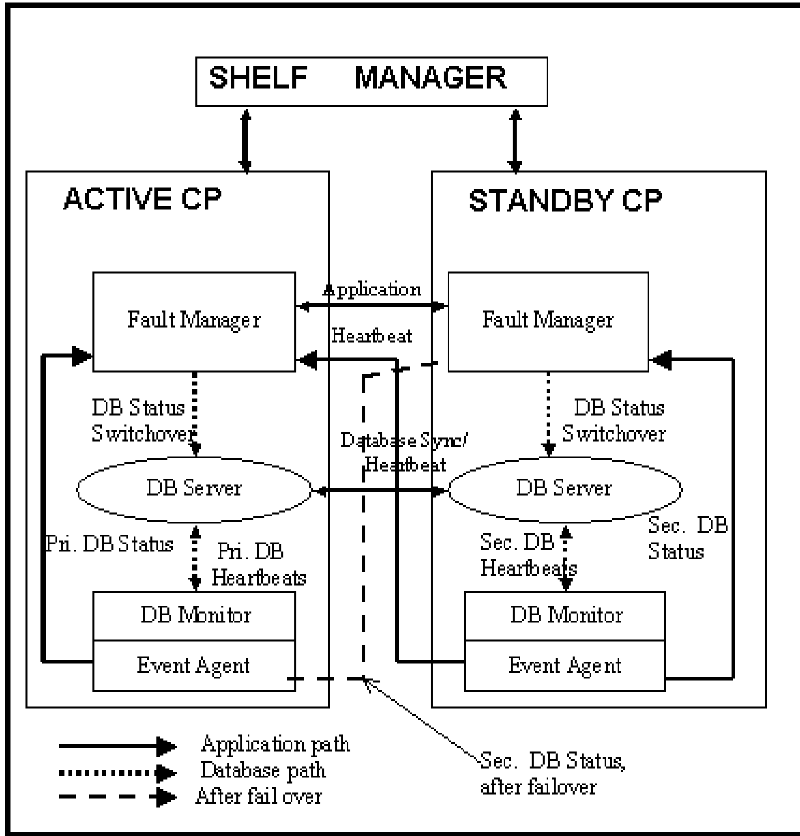


Fig. 8. Reference Implementation in EndurX

If the primary or active DB server is detected to be un-healthy by the event agent, then the fault manager initiates action to fail over the process to the mate. This involves fail over of information like connected clients, virtual IP etc. Data is available through one of the persistence models described in earlier sections.

While this is a very simplistic model that is described, it also provides for more features in terms of actual implementation. There is also a higher level of fail over provided by the shelf manager that is connected to both the blades through the IPMI [6] interface and this ensures that the node level HA is possible. Events are escalated to shelf manager only in case of the fault manager not being able to handle the failover. Fault mangers are always in synchronism about the individual roles.

5 Performance

The goal of the performance testing is to be able to verify whether the persistence model that uses database is able to handle the typical call densities and the

persistence demands thereon and also to investigate the relative performances of the different persistence models.

The test systems are Pentium P3 1.4 GHz systems running Linux. A sample benchmark program inserts rows in a table (7 columns - 2 integers, 5 characters), each row being of length 256 bytes. A commit is issued after each insert. ODBC API was used by client program for database transactions.

In the examples below, there are some tests done on a stand-alone system with only the persistence model having full access to the CPU. While this is not an intended usage model in a carrier grade system, it is useful, because it provides some delineation of performance when the data is persisted over the network, as well as differences in CPU utilization. With this in view, it is more fruitful to look at relative numbers, as shown below.

Table 1. Disk

Standalone	**Hotstandby (same machine)**
2.5X tps (Durability – Strict & Adaptive) 25X tps (Durability – Relaxed) (tps = transactions per second)	1X tps (Durability – Strict) 4.5X tps (Durability – Adaptive) 5.5X tps (Durability – Relaxed)

X = Reference taken for the case where the Hotstandby is on the same machine and the durability is "strict"

We see that as durability is relaxed, the performance numbers increases, i.e. we are able to persist more data. In case of relaxed durability, we are able to persist about 5.5X calls details; but this does not enable recovery from the crash of both primary and secondary nodes. On the other hand, we see that for strict durability, we can recover from "double faults", but then we are able to persist only about 1X call data.

Table 2. Diskless

Standalone	**Hotstandby (same machine)**	**HSB (over network)**
90X tps	3.8X tps	6X tps

Clearly, the diskless version provides the highest performance, as the data is only in memory and no disk access is involved. If the hotstandby is another processor, loss of data is minimized in case of failure of the primary processor. This is still the centralized call-processing model, so the performance is good for a centralized server.

6 Conclusion

Using a SAF compliant database is a good and flexible option for providing data persistence in telecom applications. Apart from freeing application developers

from the burden of providing persistence, use of database provides an easy means to tailor the persistence architectures to suit the application requirements in terms of performance, complexity and architecture. SAF standardized interfaces facilitate the use of compliant databases from multiple vendors.

Acknowledgements

I'd like to thank Nagaraj Turaiyur, Nitin Nagaich , Pavan Kumar my colleagues in Force Computers (Bangalore). Henry Su, Anders Karlsson, Kevin Mullenex and Tommi Vartiainen of SOLID Information Technology provided good review. Markus Leberecht, my colleague at Munich and Ushasri T.S. at Bangalore deserve a special thanks.

References

1. Institue of Electrical and Electronics Engineers, Inc., http://www.ieee.org
2. International Telecommunication Union: X.731 specification. http://www.itu.org
3. Service Availability^{TM} Forum Application Interface Specification SAI-AIS-A.01.01 http://www.saforum.org/specification
4. Service Availability^{TM} Forum: Implementing HA databases within an SA Forum AIS-compliant Framework – White Paper SAF-AS-WP1-01.01
5. Solid Information Technology Corporation: Boost Engine^{TM} Database product. http://www.solidtech.com/
6. PCI Industrial Computer Manufacturers group: http://www.picmg.org

Distributed Redundancy or Cluster Solution? An Experimental Evaluation of Two Approaches for Dependable Mobile Internet Services

Thibault Renier[1], Hans-Peter Schwefel[1], Marjan Bozinovski[1], Kim Larsen[1], Ramjee Prasad[1], and Robert Seidl [2]

[1] CTIF, Aalborg University, Niels Jernes Vej 12,
9220 Aalborg Ost, Denmark
{toubix, hps, marjanb,kll, prasad}@kom.auc.dk
[2] Siemens AG, ICM N PG SP RC TM, St.-Martin-Straße 76,
D-81541 Munich, Germany
seidl.robert@siemens.com

Abstract. Third generation mobile networks are offering the user access to Internet services. The Session Initiation Protocol (SIP) is being deployed for establishing, modifying, and terminating those multimedia sessions. Mobile operators put high requirements on their infrastructure, in particular - and in focus of this paper - on availability and reliability of call control. One approach to minimize the impact of server failures is to implement redundant servers and to replicate the state between them in a timely manner. In this paper, we study two concepts for such a fault-tolerant architecture in their application to the highly relevant use-case of SIP call control. The first approach is implemented as a distributed set of servers gathered in a so-called pool, with fail-over functionality assigned to the pool access protocols. The second is a cluster-based solution that normally implies that the servers are confined in a local network, but on the other hand the latter solution is completely transparent to clients accessing the service deployed in the cluster. To evaluate these two approaches, both were implemented in an experimental testbed mimicking SIP call control scenarios in 3rd generation mobile networks. An approach for measurement of various dependability and performance parameters in this experimental setting is developed and concluded with a set of preliminary results.

1 Introduction

Beginning with the deployment of 3rd generation mobile networks, the increased wireless access bandwidths open up possibilities for enhanced IP-based services, with one prominent class being multimedia services. The standardization bodies have taken this development into account and in reaction agreed on signaling and call control infrastructures for such services. In Universal Mobile Telecommunications Systems (UMTS) release 5, the standardization bodies have introduced the so-called IP-based Multimedia Subsystem (IMS) [1], which relies on the Session Initiation Protocol (SIP, [2]) as an application layer protocol for establishing, modifying and terminating multimedia sessions.

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 33–47, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The introduction of SIP call control in mobile core networks poses new requirements on that protocol, which was originally developed with the mind-set of the Internet architecture. These requirements are symptomatic for the process of convergence of fixed and wireless networks. In particular, the desire for high availability and reliability of such a call control system requires technical solutions which in turn must not lead to degradation of performance (such as e.g. increased call-setup times).

A widely applied solution for reliable service provisioning is the deployment of redundant servers in so-called clusters [3], in which failure detection and fail-overs are performed transparently to the entity that accesses the service. More recently, an alternative approach has emerged that is designed along the lines of general Internet paradigms: distribute the redundancy in the network and move certain failure detection and fail-over functionalities to standardized networking protocols that provide access to that server set. The Reliable Server Pooling (RSerPool) framework [4] is an important example of such a standardization effort. Redundancy for IMS-like SIP call control is further complicated by the stateful nature of the call control servers; the state has to be replicated to provide correct behavior after failing over.

The contribution of this paper is to describe and conceptually compare the aforementioned two different approaches for redundancy deployment, and map them to the scenario of SIP call control in IMS (Section 3). Furthermore, the implementation of the two solutions in an experimental test network is described and a framework for evaluation and comparison as well as an experimental methodology is developed (Sections 4 and 5). Finally, preliminary results for a range of performance and dependability parameters are obtained from the experimental set-up and lead to first conclusions about the properties of the two different approaches (Section 6).

2 Background: Call Control for IP-Based Multimedia Sessions

In the following, we give a brief introduction to the call control protocol SIP and its deployment scenario in third generation mobile networks.

2.1 The Session Initiation Protocol (SIP)

SIP was defined by the IETF in RFC3261 [2]. It is an application-layer protocol for creating, modifying, and terminating sessions over IP. These sessions can be multimedia conferences, Internet telephone calls and similar applications consisting of one or more media types. SIP is designed in a modular way so that it is independent of the type of session established and of the lower-layer transport protocol deployed beneath it. A SIP session (also call dialog or call leg) is a series of transactions, initiated and terminated respectively by an INVITE and a BYE transaction. There are also other transactions types, such as REGISTER, CANCEL, OPTIONS, NOTIFY and MESSAGE [2]. A SIP transaction consists of a single request, some potential provisional responses and a final response. Provisional responses within a transaction are not mandatory and are only informative messages. A transaction is successfully completed only when the final response is successfully received, processed and forwarded by the proxy server. SIP is based on the client-server model: typically, SIP requests are originated at a User Agent Client (UAC), pass through one or more SIP proxy servers and arrive at one or more SIP User Agent Servers (UAS), which are in charge of responding to the requests.

2.2 Deployment Scenario in 3rd Generation Mobile Networks: IP-Based Multimedia Subsystems (IMS)

In order to support IP-based multimedia sessions in UMTS, a call control infrastructure that uses SIP was introduced in Release 5. In the IMS, the SIP signaling is performed by entities called Call State Control Functionality (CSCF). The network architecture consists of three different SIP CSCF servers [5], [6] plus an additional supporting database. An overview of these four entities is given in the following, see [7] for more details:

- HSS (Home Subscriber Server) is an integrated database that consists of a Location Server, which stores information on the location of users, and a profile database, which stores service profile information for subscribed users.
- I-CSCF (Interrogation CSCF) acts as first contact point for other IMS networks and has additionally the task of selecting an appropriate S-CSCF.
- P-CSCF (Proxy CSCF) is the server initially contacted by the SIP devices. All SIP requests are sent from the sending device to the P-CSCF.
- S-CSCF (Serving CSCF) is mainly responsible for managing each user's profile and call states. It performs service control and furthermore provides interfaces to application servers.

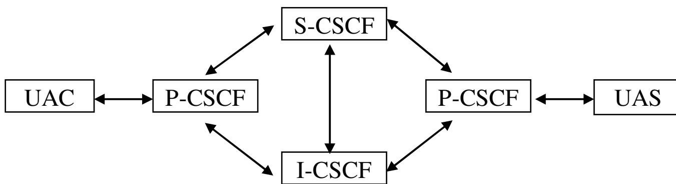


Fig. 1. SIP call control servers (CSCFs) in IMS

Figure 1 shows an example of SIP message paths in the IMS between a UAC and a UAS. CSCF servers may need to maintain states. There are three types of CSCF servers in that respect: stateless server, (transaction) stateful server and call stateful server. A stateless server is an entity that does not maintain any state of the ongoing calls or transactions when it processes requests. A stateless server simply forwards each request and every response it receives in both directions. A stateful server, or transaction stateful server, is an entity that maintains the client and server transaction states, which are updated after successful processing of a request. A call stateful server retains the global state for a session from the initiating INVITE to the terminating BYE transactions. The call state is updated after completion of each transaction.

3 Approaches for Fault-Tolerance in IMS

When designing the IMS, one of the final goals is to provide highly dependable call control, e.g. keep the SIP sessions alive, despite the possible failures of SIP servers (and despite possibly occurring network errors). Thus, the SIP-based CSCF servers have to provide fault-tolerance.

3.1 Fault-Tolerant Requirements in the SIP Architecture

In general, nodes and links are vulnerable to failures of different types: hardware component crashes, incorrect behavior of software, performance overload, human errors, physical damages of wires, connectors, etc. However, all failures can be classified into two distinct classes:

- Node failure: the corresponding node stops providing its service(s), due to a crash of a subsystem (hardware or software);
- Link failure: the corresponding link stops serving as a transmission medium to convey messages.

By definition, fault-tolerance is the property that after a failure another operational component of the system has to take over the functionality of the failed counterpart. Therefore, one challenge is to achieve a seamless transition. In the SIP architecture, redundancy in the IMS network is essential in order to provide fault-tolerance, and implies that these important functionalities are implemented in the network:

- State-sharing (SS) algorithm: replicates the states of server's existing sessions to its peers. The ultimate goal is to maintain an identical image of all states of interest (call or transaction state) in each server of a state-sharing set of servers. The state-sharing functionality naturally requires that the S-CSCF, at least, is stateful or call stateful.
- Dissemination protocol: it is the transfer mechanism of the state-sharing mechanism. It is triggered by a state change (event-driven) and its task is to distribute state updates to all peers in a state-sharing pool of servers;
- Failure-detection (FD) mechanism: a method, based on special criteria, is required that determines when a server is failed or unavailable (over-loaded).
- Fail-over (FO) management: it is activated when a failure is detected. Its task is to switch the connection to a new active server within a state-sharing pool according to the deployed server selection policy.
- Server selection policy (SSP): defines the next server candidates in case of a fail-over. This policy might be a round robin, weighted round robin, backup, persistent backup, least used, most used, etc. Note that the SSP in a fault-tolerant system is combined with the load-balancing scheme (a server is chosen at the beginning of every transaction for load distribution purposes). A list of all servers in the state-sharing pool is maintained as statically configured or dynamically obtained and updated. The first option is simpler to implement but it is hard to maintain the list updated in case of dynamic re-configuration of the state-sharing pool ((de)registration and/or failure of servers).

3.2 Support of Fault-Tolerance in Native SIP

There is no state-sharing mechanism defined for the SIP level, and therefore no fail-over mechanism either. Nevertheless, SIP includes some basic failure detection features. There exist different classes of response to the SIP requests. Classes 1xx and 2xx correspond to normal processing and/or successful completion of the transaction. If the UAC receives a response belonging to another class, it means that an error occurred. As a consequence, a retransmission of the request can be triggered by the UAC.

Native SIP also supports a timeout mechanism in association with a retransmission algorithm: if the response to the request is not received by the end of the timeout period (e.g. in case of link failure), the UAC retransmits the request to the same server. This operation is repeated until completion of the transaction within a timeout period or until the UAC drops the transaction after a certain number of failed attempts. The timeout values and number of retransmissions depend on the transport protocol and are detailed in [2].

The lack of state-sharing and fail-over mechanisms makes stateful SIP servers very sensitive to failures. In order to achieve good fault-tolerance levels, specific solutions must be added into the system. We describe two different concepts for a fault-tolerant system: a distributed system and a cluster-based system. An appropriate way to compare them is to focus on how they fulfill the three requirements for fault-tolerance support: state-sharing, failure detection and fail-over.

3.3 Distributed Architecture: RSerPool

The RSerPool concept is very simple and relies on redundancy to be deployed anywhere in an IP network (even in different sub-networks). Hosts that implement the same service (called pool elements, PE) form a so-called pool, which is identified by a unique pool handle (i.e. a pool identifier). The users of a server pool are referred to as pool users (PU). A third party entity, called name server or ENRP server, is in charge of monitoring the pool, keeping track of the PEs' status, and to help the PUs know which PEs the requests can be sent to. The RSerPool architecture is presented in Figure 2.

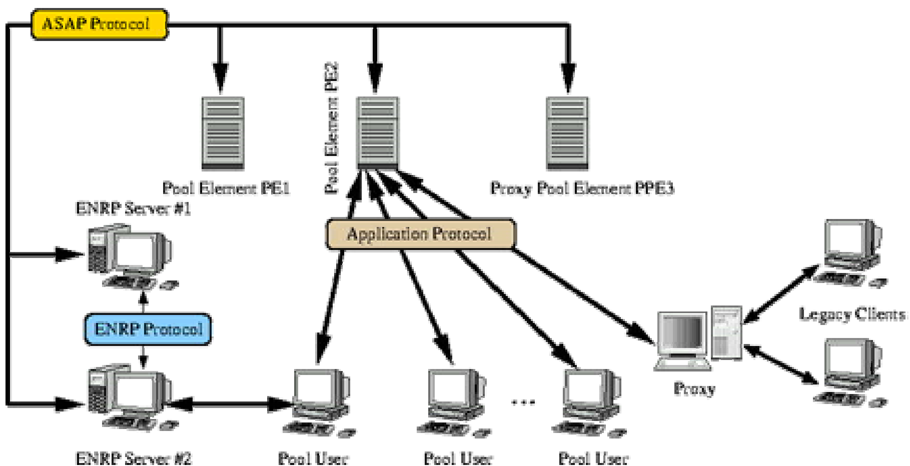


Fig. 2. The RSerPool architecture [18]

The functionality of RSerPool is based on two novel protocols: Endpoint Name Resolution Protocol (ENRP) [8] and Aggregate Server Access Protocol (ASAP) [9]. ASAP has replaced DNS in RSerPool since it is the protocol that provides the translation, called name resolution, of a pool handle sent by a PU into a set of transport ad-

dresses (IP addresses and port numbers) and adds a suggestion for a *server selection policy*. The information obtained from the NS can be kept in a cache that the PU can use for sending future requests. The second RSerPool protocol is ENRP. Name servers use this protocol mainly to disseminate the status of their PEs among their peers to make sure that the information is consistent and up-to-date in every pool (a PE can belong to more than one pool).

The requirements for high availability and scalability defined in RSerPool do not imply requirements on shared state. ASAP may provide hooks to assist an application in building a mechanism to share state (e.g. a so-called cookie mechanism), but ASAP in itself will not share any state between pool elements.

SCTP [10] is selected to be an underlying transport layer protocol for RSerPool. It makes use of its multi-homing capability to provide network fail-over. SCTP can detect node un-reachability with inherent failure detection mechanisms (retransmission timer and heartbeat auditing). When it detects a failure of the primary path, it switches the future communication over the next available active path between the two end-points (changing the network interfaces, which are connected to different networks when possible).

While delivering a message, ASAP (at the client side) always monitors the reachability of the selected PE. If it is found unreachable, before notifying the sender of the failure, ASAP can automatically select another PE in that pool and attempt to deliver the message to that PE. In other words, ASAP is capable of transparent fail-over amongst application instances in a server pool. When detecting a failure, an ASAP endpoint reports the unavailability of the specified PE to its home NS.

A name server may also choose to "audit" a PE periodically. It does this by sending healthcheck messages to this PE at the ASAP layer. While the SCTP-layer heartbeat monitors the end-to-end connectivity between the two SCTP stacks, the ASAP healthcheck monitors the end-to-end liveliness of the ASAP layer above it.

When a PE failure occurs, the PU initiates a fail-over by requesting another name translation at the NS, to get an up-to-date suggestion for an active PE, or by using the server status information in the cache and try another server in the list returned by the NS during the previous name resolution. Using the cache, the fail-over can be done as soon as the failure is detected, but with some probability that an unavailable server is selected; a repeated name resolution on the other hand slows down the fail-over, but increases the chance to fail-over to an active server.

3.4 Cluster-Based Architecture: RTP

In telecommunications environments, the clustering concept has been widely exploited [11], [12]. Redundancy is deployed via a cluster of nodes, physically collocated in the same sub-network. A single virtual IP address is advertised, so the users of the service are not aware of the cluster internal architecture. The latter property is also referred to as “single system image”. A failure is detected by a built-in failure-detection mechanism, and a server-initiated fail-over is implemented.

The specific cluster solution that is used in the experimental set-up for this paper is the Resilient Telco Platform (RTP) [13]. It consists of several autonomous nodes that are linked together via a cluster interconnect. In addition to the operating system, cluster packages link the individual nodes into a cluster. They support the cluster interconnects and offer applications well-defined interfaces that are required for clus-

ter operations, like e.g. inter-node communication. The inter-node communication supports redundant connections for availability reasons and a proprietary low overhead protocol, so-called ICF (Inter-node Communication Facility), which assures reliable and ordered packets delivery. An important characteristic of RTP is the level at which redundancy is implemented. Every process (RTP processes and the SIP application on top) is replicated among the nodes and, in case of failure, only the faulty process is failed over instead of the complete node. Therefore, all other processes are still redundant, even after one of them has been failed over.

We describe the three RTP components that are of most interest in our framework. Other RTP components were provided with the software but were not used extensively, for details see [13].

Node Manager: One of the primary objectives of RTP is to provide the application programmer with a single system image. The node-local components that contribute to the cluster-global process management (process startup, monitoring, restart or shutdown) are the node managers. Since they need to have a common view on all active RTP processes, node managers in a cluster compile their local RTP process information (process address and status) into a global process table. Any change in the local process configuration is immediately distributed to all other node managers in the cluster. The validity of the global process table is periodically verified, and consistency problems are resolved. The advantage over RSerPool is that heart-beating messages are not necessary anymore, only status updates (equivalent to ENRP messages) are needed: traffic due to the failure detection mechanism is reduced to a great extent. A direct consequence of the single image system is the isolation of the fail-over mechanism from the component that triggers the retransmission of a request. Therefore, it might happen that SIP detects a failure before the node manager. In that case the retransmission happens before the fail-over and is inefficient. The responsibilities of a node manager exceed the pure node-local process management and the contribution to a cluster-global process management. A node manager provides the communication resources for local RTP processes, informs observer processes, etc.

Context Manager: A context is a data storage accessible from anywhere in the cluster. In the example of an IMS SIP proxy, we use contexts in order to keep the call state available to any SIP instance that would need it, e.g., after a fail-over. For performance reasons, the concept of mirroring (replication) context data on another cluster node has been chosen, instead of using a common database: one CSCF node holds the master copy of the context while the other node saves the backup copy. To achieve good performance, it is recommended that any operation on a context always takes place on the node where the master copy is located. Accessing a context from any other node will result in a remote access to the master instance and will therefore impact the performance because of the inter-node communication that it requires. If the master context manager is no longer available (process or node failure), the backup context manager takes over its responsibility. When it is available again, the master context manager synchronizes its context data with the backup context manager and then resumes its old role. Even though this approach can induce latency in case the application is not located where the master context manager is running, inconsistency is completely avoided as the state is consistently read where it was written, there is no risk to access obsolete state information.

UDP Dispatcher: Its task is to act as a mediator between the RTP internal message system and systems outside of RTP, using UDP. The UDP dispatcher accepts (and sends) UDP datagrams on specified ports, analyzes these messages, and distributes them to RTP client applications according to algorithms implemented in the so-called RTP UDP plugin library attached to it.

One should be careful about failures in the external link because none of the RTP components can detect those failures; RTP processes only monitor the operations within the cluster. Therefore, the cluster approach may present a single point of failure, if there is only one “entrance” to the sub-network in which the cluster is implemented (but this is outside the scope of the current cluster solution). The solution is either to implement a ping functionality that would detect this type of failure or to use

Table 1. Summary of the main functionalities of an RSerPool-based approach as opposed to a cluster solution (RTP)

	RSerPool	RTP
Main features	- Distributed redundancy with additional node for management of the pool. - Name resolution needed prior communication	- Cluster architecture in the same sub-network. - Internal architecture completely hidden to the client: one system image. - One virtual IP address advertised.
State-sharing	- Not defined in RSerPool - Done at the application layer	- Context-based (master process accessed by default) - No inconsistency but latency in some cases
Failure detection	- SCTP: link (timer & heartbeats) - ASAP: application (timer & heartbeats)	- Node manager: process (timer & health-check) - Cluster software: node
Fail-over	- ASAP, 2 modes: ▪ Static (cache) ▪ Dynamic (name resolution)	- Node manager (dynamic, on process level)

two external links like it is done for the cluster interconnect, or in the RSerPool case when the SCTP multi-homing functionality is used. In other words, the current implementation of RTP is more sensitive to link failures than RSerPool.

Table 1 conceptually compares the two solutions by giving the main characteristics and fault-tolerance mechanisms designed.

4 Experimental Environment

4.1 Description of the Testbed

We want to investigate the call control part of the IMS in an experimental set-up whose architecture is shown in Figure 3: both user agents, client and server, are co-located in one machine. The second machine embeds the P- and I-CSCF servers. This is quite natural since both servers work together to redirect the requests from the UAC

to an S-CSCF in the core network, which processes and forwards them via the server-side P-CSCF to the UAS. On the way "back", the responses use the same path, as collected in the via field of the request. As the main service-control functionality is integrated in the S-CSCF, the fault-tolerant architectures are applied to this component, i.e. it is replicated in PC3 and PC4.

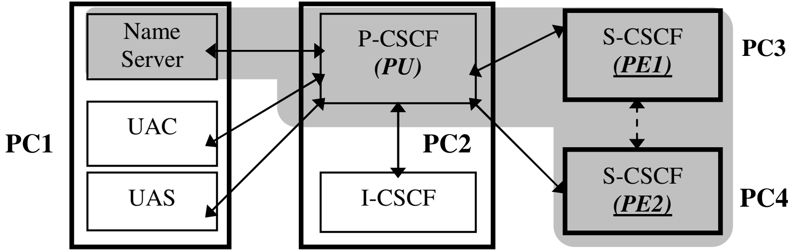


Fig. 3. Testbed physical and logical topology running the IMS/RSerPool system

RSerPool Topology: The gray shaded part in Figure 3 represents the logical RSerPool system. The redundant S-CSCFs form a server pool. The choice for the Pool User was motivated by the architecture of the IMS itself. The role of the PU is then taken by the P-CSCF, and possibly also the I-CSCF. However, since the I-CSCF is only in the SIP path for Register requests and those are not part of the investigated call scenario (see Sect. 4.2), the I-CSCF is not implemented as PU. The PU (P-CSCF) and PE (S-CSCF) use SCTP for name resolution and name registration/deregistration, respectively. The data exchange is done via UDP at each entity in the testbed.

RTP Topology: The physical topology is the same as in the case of RSerPool (Figure 3) but the fault-tolerance functionalities are in the cluster only, which is made up by the two S-CSCFs. The other entities know one IP address for the cluster, the virtual IP address, and they do not know or see which S-CSCF node processes the next SIP message. The RTP platform dispatches the message to the appropriate server that is chosen based on the SSP, which has been set in the node manager.

4.2 Traffic and Failure Models

There are possibly multiple UACs running at PC1; all initiate and terminate calls, not necessarily synchronously, which leads to parallel sessions (interesting for scalability evaluation). The number of simultaneous sessions equals the number of UACs in PC1 and is denoted by M. The UACs follow the same call/transaction generation pattern. Between the INVITE and BYE transactions, each UAC generates instant message transactions (denoted IM) with the MESSAGE request. Within one evaluation run, each UAC generates a number of sequential sessions, denoted by N.

The server selection policy is persistent backup. This means that all the transactions in a session are sent to the same S-CSCF until a failure is detected at this server.

Then, there is a fail-over to an alternative server, which becomes the new server by default for all the next transactions until this server fails, and so on.

In the RSerPool configuration, a UDP-based state-sharing protocol is deployed as described in [14]. Because of some SCTP software limitations, we preferred not to make use of the SCTP features for failure detection and link fail-over management (link failures are seen as being part of the node or application failures in that case, and are detected at the application layer). The failure-detection mechanism per SIP request is implemented in the P-CSCF. It is defined by the timeout value T_1. After the SIP timeout expires, the P-CSCF retransmits the request to the other S-CSCF. We simulate the cache functionality defined in RSerPool: after the first name resolution requested at the name server, the P-CSCF keeps the list of servers returned (described by their transport address). When a fail-over occurs, there is no need for another name resolution request and the P-CSCF uses the other transport address obtained.

In the RTP system, the state-sharing and access to the context is done according to the 'standard' RTP configuration: the primary context manager is contacted first. The dissemination protocol for state sharing in the RTP-based system is the proprietary low overhead protocol, so-called ICF (Internode Communication Facility), which assures reliable and ordered packets delivery. When a node manager detects a failure, it automatically fails over to the redundant SIP instance in the cluster.

To simulate failure and repair processes, an artificial ON/OFF random process is established in each S-CSCF. Samples for the random variables time-to-failure (TTF) and time-to-repair (TTR) are generated in each S-CSCF and these random variables describe an ON/OFF process, enforcing a server to go up or down. TTF and TTR are generated from exponential distributions, with mean values MTTF and MTTR, respectively. In practical systems, it normally holds that MTTR/MTTF<<1.

4.3 Input Parameters

For the call scenario definition, we need the following random variables:

- Within each call, the time interval between the reception of a response of one transaction and the time instance of sending the request for the subsequent transaction is called inter-transaction time. It is set to an exponential distribution with mean value $1/\lambda$ (exponential distributions are chosen in order to simplify analytic modeling of the system; however this is not in the scope of this paper).
- The number of IM requests generated within a call is a random number. It is also determined by exponential distribution, with the mean value K.
- The call duration is exponentially distributed with mean value $1/\mu$.

Although there is a possibility to use a link emulator to create artificial delays/losses, the experiments in this paper use the ideal setting in the emulator, infinite bandwidth B, and no delay (D) or packet loss (P), so that the link characteristics are purely determined by the physical 10Mb/s Ethernet links.

Table 2 lists all the input parameters and gives their values for each system.

Table 2. Input parameters in the testbed

	RSerPool	***RTP***
M	2	
N	1000	
$1/\mu$	120 sec.	
$1/\lambda$	2 sec.	
K	10	
MTTF	190 sec.	
MTTR	10 sec.	
dissemination protocol	UDP	ICF
T_1	1 sec.	
number of retransmissions	1 (to the other S-CSCF)	1 (to the same virtual IP address)
B, D, P	∞, 0, 0	
SSP	persistent backup	

4.4 Output Parameters and Metrics

When evaluating a fault-tolerant system, two sets of metrics are very significant and often in a trade-off relationship with each other: service dependability and performance. The first one directly gauges the benefit from the added redundancy while the second indicates e.g. whether the solution is suitable for services with hard or soft real-time requirements.

The service dependability metric set comprises the following metrics:

- $\hat{A}$: session availability (INVITE transaction)
- $\hat{D}$: IM service dependability (for non-INVITE and non-BYE transactions)
- $\hat{R}$: session reliability (BYE transaction)

The session availability is the probability that a session can be initiated when a UAC sends an INVITE request to the UAS. The IM service dependability measures the fault-tolerance level of the service provided to the user once the session has been initiated. The probability that a session is successful from its initiation to its termination is determined by the session reliability. Each of these metrics is measured as the ratio between the number of successful transactions of one type and the total number of requests for the type of transaction.

Note that the total number of BYE requests is equal to the total number of successful INVITE transactions. Indeed, a UAC can request a BYE transaction only if a session has been initiated. Note however that in the current setting, the IM transactions are always sent, even in case of an unsuccessful INVITE at session start.

The performance metric set consists of the metrics that give the transaction control time for each type of transaction. We distinguish between INVITE transaction time, IM transaction time and BYE transaction time. They are defined as the average duration of the interval between the moment of sending the request and the moment

of receiving a final response to the request at a UAC for their respective type of transaction. Cases when retransmissions occur are considered, but only for successful transactions. Note that the fail-over time is not explicitly measured because it is hidden in transaction control time. In fact, for a given system and scenario, the fail-over time influences the difference between the transaction control time in the fault-tolerant system and the transaction control time in the non-fault-tolerant system.

As for the dependability metrics, the transaction control times are measured at every user agent client.

Other metrics such as inconsistency, total call control time, and scalability (e.g. as measured by the increase of transaction times for increasing number of parallel sessions) are very relevant for evaluation purpose but beyond the scope of this paper.

5 Evaluation Approach and Preliminary Results

5.1 Evaluation Method

When evaluating a fault-tolerant system, in order to obtain significant and fair results, the system should only be subject to the artificial failures as described in Section 4.2. However, in reality, it is practically impossible to run long evaluation scenarios in a prototype without experiencing uncontrollable failures (not artificially introduced by the ON/OFF model). In order to check for the nature of the latter, we first ran the SIP software without fault-tolerant solution and with only one S-CSCF in the testbed. Furthermore, in this setting all the CSCFs were co-located in the same physical machine for simplicity. We observed that failures occurred at any CSCF, too frequently at the scale of an evaluation run to be negligible. The system could never recover from those uncontrollable failures so we chose to periodically restart the system. Each evaluation run was split into blocks of a fixed number of transactions. The desired outcome of implementing such a procedure is to minimize the probability of the uncontrollable failures and to allow for recovery. Concretely, a restart script was implemented at PC1. One UAC, called controlling UAC, is in charge of counting the number of transactions processed. When the counter reaches a certain value (100 in our setting), this UAC waits for the end of the ongoing session it is participating in before it triggers the restart script. This program shuts down all the SIP components in the system (UAs and CSCFs). The components are then started up and the UAs registered. When both UACs are registered, the traffic model explained in Section 4.2 is applied again during the next block. The artificial failure model in the S-CSCFs is restarted in the same state (ON or OFF) as it was in before, so that due to the memoryless property of the exponential distribution, it is not affected by the restarting.

It appears that the uncontrollable failures did not completely disappear: for a block-size of 100 transactions, approximately 95% of the blocks could be successfully finished. If an uncontrollable failure occurred, it did not show any particularly pronounced dependencies on the number of sessions.

5.2 Preliminary Results

The output metrics explained in Section 4.4 were measured in three environments (Table 3):

Table 3. First set of results

	SIP (co-located CSCFs)	SIP + RSerPool	SIP + RTP
A [%]	**91.278** (0.3971)	**97.32** (0.031)	**95.84** (0.0381)
D_{im} [%]	**90.538** (0.4696)	**97.24** (0.0406)	**95.56** (0.0319)
R [%]	**99.06** (0.0044)	**99.69** (0.0011)	**99.21** (0.0023)
T_{inv} [s]	**0.12**	**0.12**	**0.10**
T_{im} [s]	**0.04**	**0.08**	**0.06**
T_{bye} [s]	**0.04**	**0.08**	**0.06**

- Original SIP, with one S-CSCF node only; also P-CSCF and I-CSCF are co-located in the same machine as the S-CSCF.
- SIP in the distributed architecture (RSerPool, Sect 3.3).
- SIP in the cluster solution (RTP, Sect. 3.4).

Both fault-tolerant architectures are implemented with two S-CSCF nodes. The numbers in the parentheses indicate 95% confidence interval obtained from repeated evaluation runs. The observations from Table 3 are the following:

- The fault-tolerant solutions increase availability, dependability, and reliability, as expected, RSerPool showing slightly better results than RTP though.
- The confidence intervals also show a steadier behavior of the system with RSerPool and RTP as compared to the original SIP version.
- Transaction times for the IM transactions and the BYE transactions increase in the fault tolerant solutions. Thereby, in this special evaluation scenario, the cluster-based approach (RTP) was 15-25% faster than the RSerPool-based solution.

However, be careful to avoid generalizing too much from this single setting. The implementations was optimized for performance and statistics functions for the measurements can potentially put higher load on the processors and I/O systems. In particular, some implementation internal threads had to be duplicated for implementation purposes, which cause higher load and task switching times. Also, for the RSerPool-based implementation, additional file handling was added in the S-CSCF for evaluation purpose, which increases the processing time. In a performance optimized implementation this can be avoided.

Furthermore, we can observe that in all solutions, the reliability level (successful BYE rate) is higher than the two others. It is due to the evaluation method: when an INVITE is failed, the IM transactions are sent anyway and their dependability counted, which is not the case for the BYE. As we explained above, the system cannot recover a failure before the next restart so when the INVITE is failed, all the next IMs are as well (availability and dependability drop) while reliability is not affected because no BYE is sent.

Because of the uncontrollable failures, it is not possible to draw any conclusion concerning the absolute values of A, D, and R in the first test in Table 3. Nevertheless, looking at the uncontrollable failures and the ON/OFF model as two independent

processes, one could measure the level of uncontrollable failures and derive formulas to obtain absolute values for each metric, as in the case of controlled probability of failure and known distribution. If the two processes are independent the fault-tolerant system could be seen like in Figure 5.

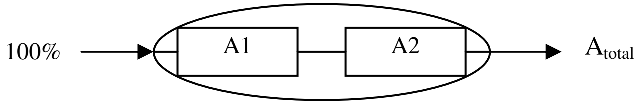


Fig. 5. Availability graph for the IMS fault-tolerant system splitting the system virtually in two parts, A2 determined by uncontrollable failure, while A1 is controlled from the artificial ON/OFF model

In Figure 5, A1 represents the availability that we would get if only the ON/OFF failure model were running. A2 is the availability when the uncontrollable failures only happen. As we assume that A1 and A2 are independent, we can derive the simple formula for calculating the overall availability:

$$A_{total}=A1*A2 . \quad (1)$$

In our testbed we know A_{total}, which is the availability measured when uncontrollable and ON/OFF failures impact the SIP transactions. The next step for us will be to run a new series of tests without the ON/OFF model so that we get A2. Then, the calculation of A1 (the value we are looking for) is straightforward:

$$A1 = A_{total} / A2 . \quad (2)$$

6 Conclusion

In the process of converging wired and wireless networks, IP protocols now face requirements that were traditionally posed on Telco products. In particular, technical solutions for highly reliable IP-based service provisioning are required. This paper describes two different approaches –a distributed server pool with standardized pool access protocols (RSerPool) and a cluster-based approach– and maps them to an IMS-like SIP call control infrastructure. The differences between the two approaches with respect to the main components of a fault-tolerant system (failure detection, fail-over, and state-sharing) are discussed conceptually and an experimental implementation is described. Evaluation scenarios, output parameters, and an evaluation approach are developed that allow to experimentally investigate various performance and reliability parameters of the two solutions implemented. Preliminary evaluation results show that unavoidable, inherent SW and HW instabilities complicate a sound experimental analysis and directions are pointed out to deal with such issues. Regardless of the methodological difficulties, the preliminary evaluations show the feasibility of both approaches and allow to obtain first insights into their behavior.

Acknowledgements

The authors would like to thank Siemens Mobile Networks, Germany, for supporting this research. Furthermore, we are grateful to Fujitsu-Siemens Computers (FSC) for providing the Linux based RTP implementation and for technical support for the RTP implementation. With respect to the latter we want to specially thank Dr. Manfred Reitenspiess and Anton Spirk, both FSC. Finally, we appreciated the support of Siemens PSE, in particular Jiri Prokes, for supplying the SIP007++ stack.

References

[1] 3GPP TS 23.228: "IP Multimedia (IM) Subsystem - Stage 2", Technical Specification, June 2001.
[2] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, Internet Engineering Task Force, June 2002.
[3] A. Helal, A. Heddaya, B. Bhargava, *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1996.
[4] M. Tuexen, Q. Xie, R. Stewart, M. shore, J. Loughney, "Architecture for Reliable Server Pooling", <draft-ietf-rserpool-arch-07.txt>, October 2003.
[5] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, D. Gurle: "Session Initiation Protocol Extension for Instant Messaging", <draft-ietf-sip-message-07>, September 2002.
[6] P. Kim and W. Boehm, "Support of Real-Time Applications in Future Mobile Networks: the IMS Approach", *Sixteenth Wireless Personal Multimedia Communications*, October 2003.
[7] Siemens. 2001. Architectural Design Specifications, Version 1.1, Project SIP007+.
[8] Q. Xie, R. R. Stewart "Endpoint Name Resolution Protocol", draft-ietf-rserpool-enrp-01.txt, November 2001.
[9] R. R. Stewart, Q. Xie: "Aggregate Server Access Protocol (ASAP)", <draft-ietf-rserpool-asap-01.txt>, November 2001.
[10] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, V. Paxson., "Stream Control Transmission Protocol", RFC 2960, October 2000.
[11] Vogels W., "The Design and Architecture of the Microsoft Cluster Service", in Proc. FTCS-28, 1998, pp.422-431.
[12] C. Yang, M. Luo, "Realizing Fault Resilience in Web-Server Cluster" SuperComputing and Networking 2000, November 2000.
[13] Resilient Telco Platform, V2.0 for Linux and Solaris, "RTP Overview and Programmer's Guide", Fujitsu Siemens Computers 2002.
[14] M. Bozinovski, L. Gavrilovska and R. Prasad, "Fault-tolerant SIP-based Call Control System", *IEE Electronics Letters,* Volume 39, Number 2, pp. 254-256, January 23, 2003.
[15] P. Conrad, A. Jungmaier, C. Ross, W.-C. Sim, M. Tüxen:"Reliable IP Telephony Applications with SIP using RSerPool", from the Proceedings of the SCI 2002, Volume X, Mobile/Wireless Computing and Communication Systems II; Orlando, USA; July 2002, pp. 352-356
[16] Haifeng Yu and Amin Vahdat, "Building Replicated Internet Services Using TACT: A Toolkit for Tunable Availability and Consistency Tradeoffs", *Second International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, June 2000.
[17] Kin K. Leung, "An Update Algorithm for Replicated Signaling Databases in Wireless and Advanced Intelligent Networks", *IEEE Transactions on Computers* 46 (3): 362-367 (1997).
[18] http://tdrwww.exp-math.uni-essen.de/dreibholz/rserpool/rsp2.png

OpenHPI: An Open Source Reference Implementation of the SA Forum Hardware Platform Interface

Sean Dague

International Business Machines,
2455 South Road P096,
Poughkeepsie, NY 12601
sldague@us.ibm.com

Abstract. OpenHPI is production ready reference implementation of the SA Forum HPI specification. It is designed to be easy to setup and use, as well as utilize and compliment existing hardware management standards. OpenHPI is available as open source so it can be easily productized by anyone requiring an HPI implementation for their hardware.

1 Refresher on HPI

Over the past decade, server hardware has evolved at an astonishing rate. One of the main areas of on enhancement has been the commoditization of management adapters. These adapters provide highly detailed monitoring and control of hardware devices, often through an out of band mechanism that works even without a functional Operating System. Such an explosive growth is generally accompanied by many competing solutions to the problem, which is exactly what has happened in this space. This has put a burden on application developers, generally systems management or high availability, that wish to take advantage of these capabilities, as they must develop to many different competing hardware technologies.

HPI, the Hardware Platform Interface, was designed to address this issue. It was first published by the Service Availability Forum in October of 2002, and defines a standard interface for Platform Management. This interface includes modeling of hardware as abstract resources, and adds capabilities to those resources as appropriate. These resources can then be controlled and monitored by applications linking to an HPI library. This provides a single, programmatic interface for application developers that desire access to hardware.

HPI has a legacy in Intel's IPMI (Intelligent Platform Management Interface) specification, so those familiar with IPMI will recognize many concepts in HPI. However, HPI is sufficiently abstract that an HPI implementation can interface well with non IPMI hardware. Those interested in HPI should consult that specification as the final definitive source.

2 Introduction to OpenHPI

The OpenHPI project was started in January of 2003 with its project registration on the SourceForge.Net website. The idea behind OpenHPI was spawned from the

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 48–60, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Carrier Grade Linux meetings in NYC at Linux World Expo that year. The major development push on OpenHPI started at the end of August 2003, after many of the OpenHPI members were able to meet face to face at the Ottawa Linux Symposium.

It is often said, "Standards are great, that's why we have so many of them." A specification only becomes a standard when it achieves a critical mass of adopters and users. Up until that point the specification is in danger of being co opted by competing efforts that offer equivalent functionality. The OpenHPI project is designed to build an ecosystem of developers and users around HPI to help take HPI from a specification to a standard.′

In order to do this effectively OpenHPI aims to be widely applicable, in every sense of the word. It must be usable on a variety of hardware platforms, from single systems to large heterogeneous multi-vendor clusters. It must be configurable to many different customer environments, with very little effort. It must show that although HPI was formed out of the needs of the Telecommunications sector, it is also widely applicable and usable beyond it. And lastly, to take hold in the Linux market place, it must be open source software.

This last requirement comes from a number of areas. The first is to make it easy to create a developer community around HPI. By providing a reference implementation of HPI in Open Source under a business friendly license, developers can take OpenHPI in its entirety and modify it. Or they can contribute the software component which enables the hardware they need. Some organizations have joined the OpenHPI effort to do just this, and more are looking to do so in the future.

The second, and possibly most important, is that it allows the distribution of the code base far and wide. Linux Distributions are free, and encouraged to make OpenHPI part of their base software package. Being under an acceptable license is only the first mountain one must climb to gain wide scale acceptance in the Linux community. The software must also be written to certain community standards for development, and solve problems in a "Linux like" way. It must prove its design is sound through rigorous peer review, and incorporate new features, and find and eliminate software bugs, in a timely manner. All of these challenges are the same as faced by any software project, however the Open Source nature means there is a wider audience watching every stage of development, and a much large potential developer pool that may contribute to the software.

The purpose of this paper will be to discuss how these forces have helped evolve the OpenHPI project, design decisions made in providing an open and freely available HPI implementation, and show the progress and current state of OpenHPI. It is hoped that this will introduce many to the OpenHPI development process, and encourage interested parties to experiment with OpenHPI for their platform management needs.

3 Design Goals

When starting any software project, design goals must be set. These goals influence all decisions of the development process, and help decide what kind of implementation is appropriate. There are any number of approaches that could be taken to fulfill the HPI API. The following shows how these design goals shaped the architecture for OpenHPI.

3.1 Simplicity

KISS, keep it simple stupid, is an overriding motto of the OpenHPI team. When doing interface design, this means taking the simplest possible approach to most problems. Simple solutions are mostly likely to be implemented correctly, and not contain hidden edge cases or design flaws.

Simplicity of design also means making usage simple for users. Many users of OpenHPI will do nothing more than download an RPM package from the OpenHPI website, install it, and expect to be off and running. So OpenHPI must conform to what they intuitively expect it to be from square one. For this reason, OpenHPI is only a single C shared library which includes a number of hardware plugins. This means that OpenHPI does not consume resources when not being actively used. It gives the user application control over the lifespan and activity of HPI. The downside from this approach is that events which occur while the HPI enabled user application is not running may be lost. However, if the user wasn't running their own application to deal with these events in a timely manner, then they probably do not care about the events in the first place.

As a correlation to the concept of simplicity, reuse is another mantra. Nothing should be reinvented if there is suitable code available under Linux to provide that functionality. Code that has been in general use in the Linux community will be more robust and better tested than anything that is newly written. OpenHPI uses glib for all complex data structures, net-snmp for SNMP access, libtool for plugin loading, and OpenIPMI for one of the IPMI plugins.

3.2 Broad Hardware Support

OpenHPI is meant to be a general purpose HPI implementation which will work with a very large array of hardware and management protocols. This includes IPMI based machines, white box Linux machines with an SMBus interface, network accessible servers or switches with SNMP interfaces, and just about any other network reachable device with some type of management interface. Given that the OpenHPI team is only a finite number of individuals, the only way to enable such a vast array of hardware is to make it easy for 3rd party developers to do it themselves.

This requires a well thought out and proven hardware plugin interface. The plugin interface makes it easy for anyone to add support for their hardware. For instance, support for the Linux watchdog device (aka /dev/watchdog) was added with only 360 lines of code. The robustness of this plugin interface has been proven by the fact that there are currently 8 plugins in the main distribution, which include 2 different approaches to IPMI enablement, IBM Blade Center enablement via SNMP, and a static hardware simulator.

In addition to the plugin interface, the OpenHPI team is always asking the question, "What additional internal interfaces would make it easier to create a plugin or other infrastructure code?". This driving question spawned the creation of internal interfaces for Resource and RDR entry management, Entity Path manipulation, and Id assignment. It is expected that many more such interfaces to be created over time.

3.3 Expectation for Commercialization

OpenHPI is licensed under a BSD style license, the complete terms of which are included with the OpenHPI distribution. The BSD license is generally considered the most business friendly of all open source licenses. It provides source code along with the binary application. Unlike the GPL, it does not oblige the user to open source changes they make to that source code if they don't want to.

It is the hope of the OpenHPI team that anyone working with OpenHPI, either to productize the existing code or enable OpenHPI for new hardware platforms, will contribute their changes back to the mainstream. This will reduce the development and maintenance cost for those working with OpenHPI. There are many organizations joining the OpenHPI team to do just this, and a number of additional organizations that have shown interest in joining the effort.

It is worth noting that these guiding principles for OpenHPI have given the project dramatic growth over the past year. In July 2003 there were 5 active developers on the project, as of January 2004, that number is over a dozen, with more interested in participating every day.

4 Internals

4.1 Infrastructure

The portion of OpenHPI which the user links to, and sits above the hardware plugins, is referred to as the infrastructure. The infrastructure is a C shared library built by libtool. Although development has been entirely focused on Linux at this point, the use of the GNU autoconf tool chain will make it easy to port to other operating systems.

The infrastructure responsibilities include interaction and coordination of information from the plugins. It also manages the Domains and all items that are domain specific, like the Resource Presence Tables, Domain level System Event Logs, and event queues. As will be discussed later, the plugins do not have a global enough view to support Domains directly.

Because all these items are managed by infrastructure, 20 of the 56 HPI API calls can be fulfilled by infrastructure without needing to communicate directly with a plugin. This means less work for plugin implementors.

4.2 Pluginsand Plugin API

Plugins are libtool archives which contain an oh_abi_v2 structure. This structure is a series of 36 named function pointers. These functions represent the plugin API, and hence the contract between the infrastructure and plugin. A plugin implementor need only implement those functions that will be supported in their plugin. All unimplemented functions will produce UNIMPLEMENTED errors if they are invoked via an HPI call.

The first step of OpenHPI initialization is the parsing of an OpenHPI configuration file. This will specify which plugins should be loaded in the environment. The user can load as many plugins as they desire, as each plugin will provide access to a different type of hardware.

Many of the HPI function calls are passed directly to the plugin API with almost the same parameters. A good example of this is the System Event Log APIs. If it is determined that a System Event Log API is not requesting information from a Domain Event Log, then it must be a Resource Event Log call. The ResourceId is used to lookup the associated handler in the Resource Presence Table. This handler is then passed its handler_state pointer, and all the same parameters that the Event Log API call was passed.

SA HPI Function Definition

```
SaErrorT SAHPI_API saHpiEventLogEntryGet (
    SAHPI_IN    SaHpiSessionIdT     SessionId,
    SAHPI_IN    SaHpiResourceIdT    ResourceId,
    SAHPI_IN    SaHpiSelEntryIdT    EntryId,
    SAHPI_OUT   SaHpiSelEntryIdT    *PrevEntryId,
    SAHPI_OUT   SaHpiSelEntryIdT    *NextEntryId,
    SAHPI_OUT   SaHpiSelEntryT      *EventLogEntry,
    SAHPI_INOUT SaHpiRdrT           *Rdr,
    SAHPI_INOUT SaHpiRptEntryT      *RptEntry
);
```

Plugin API Function Definition

```
SaErrorT (*get_sel_entry)(void *hnd,
    SaHpiResourceIdT id, SaHpiSelEntryIdT current,
    SaHpiSelEntryIdT *prev, SaHpiSelEntryIdT *next,
    SaHpiSelEntryT *entry);
```

It is worth noting that Rdr and RptEntry fields are not passed down to the plugin API call. Both of these can be derived by the infrastructure from the supplied ResourceId and returned SelEntry, so making the plugin do extra work to provide these separately isn't required.

4.3 Handlers

Just because a plugin is loaded doesn't mean that it is connected to any hardware. Although this could be automatically done for local hardware enablement, access to remote hardware needs additional information to connect to that hardware. For this reason, the concept of handlers was created.

Handlers are instances of plugins. Each handler contains a reference to the plugin API functions for the plugin it is derived from. It also contains local state data (often stored in the convenience oh_handler_state structure), and configuration data.

Handlers are created by stanzas in the OpenHPI configuration file, which include a reference to which plugin is being used, and an arbitrary set of name/value pairs

specified by the user. The name /value pairs are used as configuration data by the handler during its open call. An example of what this data looks like is listed below:

Handler Stanzas

```
handler snmp_bc {
    community = "myP@ss"
    host = "bc1.mydomain.com"
    entity_root = "{SYSTEM_CHASSIS,1}"
}
handler snmp_bc {
    community = "my0ther"
    host = "bc2.mydomain.com"
    entity_root = "{SYSTEM_CHASSIS,2}"
}
```

There is no limit on the number of handlers that can be created. Each handler represents one hardware connection. How many resources are provided by a single handler will be very plugin specific. For instance, the Linux watchdog plugin doesn't provide any resources, just rdrs that are associated with a different plugin's resource. The SNMP Blade Center plugin will produce over 50 resources when connected to a fully populated Blade Center chassis.

4.4 Resources and RDRs

Much of what is at the heart of the HPI specification is the concept of Resources and RDRs, and without some basic foundation in the concepts, the rest of this paper will make very little sense. In the simplest terms a Resource is a physical entity, something you can put your hands on. Examples of Resources include server racks, types of servers, cpus, disks, power supplies, and blades of various types. Resources have "attributes", known as Resource Data Records or RDRs, associated with them that provide information about the Resource. RDRs may be one of 4 types: Sensor (voltage, thermal, fan speed, etc.), Inventory, Watchdog, or Control. Resources can have capabilities associated with them which tell the user if the Resource is replaceable, hotswappable, and/or supports any of the RDR types. The first operation an HPI application should do is perform a discovery of all Resources to understand what hardware it is interfacing with.

In short, a Resource is something physical which can (and eventually will) fail. RDRs provide mechanisms for managing, controlling, and gathering information about these resources.

4.5 Entity Paths and Resource Ids

The HPI specification uses the ResourceId (a 32bit value) to uniquely identify a resource within the context of an HPI session. HPI makes no guarantees that this Resource Id references the same resource across sessions. Although convenient for

programmatic use, it makes it difficult to specify *a priori* that Blade 3 in Rack 2 is the machine you need to check on. Fortunately ResourceId is not the only guaranteed unique value for a Resource, Entity Path must also be unique for each resource.

Entity Paths in HPI represent the topology associated with Resources. They include a series of pairings of entity types with entity instances. For instance, in OpenHPI (which uses zero indexed entity instances) "{ROOT,0}{RACK,2}{CHASSIS,0}{SBC_BLADE,4}" means the 5th blade in the 1st chassis in the 3rd rack of all the resources seen by HPI. Given that this entity path must be unique and global within the scope of HPI, the OpenHPI team has decided to use it as the base for assigning Resource Ids.

This is done in the following way. A global table is created internal to OpenHPI. Whenever a new entity path is found, a ResourceId is allocated for it. This information is kept in a state file (/var/lib/openhpi/uid.map by default), and written out to disk on any change. Hence, anything short of file system failure won't reset the table. Because of this ResourceId 7 will always refer to the same Entity Path for all invocations of OpenHPI on the same machine.

This is the same Entity Path, and not the same Resource, as there are at least 2, and probably more, ways in which an Entity Path might not refer to the same resource. The first of which is for FRU resources. If a piece of hardware fails and is replaced, the new hardware will have the same entity path, however it will have different inventory information, and may be different in other ways. The second involves the entity_root, and will be described later.

4.6 Entity Root

Handlers are responsible for discovering their hardware and creating fully instantiated Resources and RDRs for them. The one sticking point involves the entity path that was described above. A plugin does not inherently have enough information to understand how it's resources fit into the over all entity scheme. A handler connecting to a server won't generally have any way to discover which rack that server is in, or even what instance value that server should have. These are decisions which must be left to the decision of the local site administrator.

Every handler stanza must therefore have an **entity_root** specified. The value for this is the canonical (as defined by the OpenHPI team) string format of the entity path. SAHPI_ROOT is assumed, as it is required for every entity path. For all parts of the path the format is {ENTITY_NAME,ENTITY_INSTANCE}, going from most to least significant (which happens to be the reverse order of the HPI struct definition). The entity names are actually the text of the enumeration values minus the SAHPI_ prefix, as it is redundant.

Although it isn't an error if two handlers share a common entity root it does produce a warning. There may be instances where this is required. The use of the Linux Watchdog device will often be one of these cases, as the rest of the resources will come from some other handler.

Obviously, if the administrator changes the entity_root definitions because of a lab reorganization all the Resource Ids will now reference different resources than they did before. It is expected that a change like this is not done lightly, and application owners in the environment would be notified before such a change, but the possibility exists nonetheless.

4.7 Domains

At this point, OpenHPI only supports a single domain. In the HPI 1.0 specification it is explicitly stated that this is allowed, and that resources may appear in multiple domains. This implies that all resources would appear in SA_HPI_DEFAULT_ DOMAIN. As there isn't yet a standard for security mechanisms in HPI, creating other domains seems to only produce more work for the user application at this time. This interpretation is 100% compliant with HPI 1.0.

OpenHPI will support multiple domains in the future. Domain structure will be specified by the site admin in the openhpi.conf file. Allowing plugins to generate domains doesn't make much sense, as they don't have a global view of all the resources in the system.

Given that Entity Path is unique per Resource, OpenHPI will provide multiple mechanisms for specifying domains based on matching of Entity Paths. For instance, one could specify a domain as everything which exists under "{ROOT,0}{RACK,1}". Another possibility is to lump all resources which have "{FAN,X}" in their entity path into a single domain (i.e. the "all fans domain"). By providing mechanisms for the site admin to specify their own domain definitions, local site policies can be used for domain structure instead of arbitrary domain instantiation based on the HPI implementation.

5 OpenHPI in Action

5.1 Resource Discovery

The first functionality that any plugin must provide is resource discovery. The OpenHPI team wanted to make this process straight forward so plugin writers could quickly have prototyped code, and be able to build from it. Resource discovery also demonstrates the event loop that OpenHPI uses to allow plugins to communicate information back up to infrastructure.

All communication from plugin to infrastructure happens via oh_events. The oh_event is a union which includes typing information as well as an SA HPI defined type. These events can be a standard HPI event type which will be processed and returned to the user when saHpiEventGet is called. They can also be internal only events for resource addition/removal, and RDR addition/removal. The later types are used during the discovery process to populate the RPT for the domain in infrastructure.

When saHpiResourceDiscover is called, each handler's discover_resources is called in turn. This lets each handler do whatever is needed to create resource definitions. It is expected that the handler creates those definitions as oh_events, and stores them in an internal event queue. As soon as this process is complete, the "event loop" is entered, and all handlers have their get_event function called until no more events are found. Each of these events is processed in turn. If it is a resource or RDR related event, the RPT is modified appropriately. If it is an HPI event, it is added to the event queue for the domain, where the user will find it on the next saHpiEventGet call.

The beauty of the plugin API is that it defines a very real data boundary between the handlers and the OpenHPI infrastructure. Information sent down to the handlers is

done so only as SA HPI defined types, and information sent up from the handlers is in the form of events that are mere wrappers for SA HPI defined types. This ensures that new contributors or plugin implementors for OpenHPI need not learn a whole new set of internal types to produce working plugins. Many members of the OpenHPI team who have been working on plugin enablement haven't ever looked into the OpenHPI internals. The interface is clear enough that it was never required.

6 Interoperability

Interoperability is extremely important to the OpenHPI team. These means both interoperability with other HPI implementations, as well as other management interfaces and protocols.

6.1 Mining SNMP Data

HPI has a legacy in IPMI, however it is sufficiently abstract that this isn't the only mechanism that can be used to fulfill HPI. Over the 20 years since SNMP was defined, it has become a de facto standard to communicate with network attached devices programaticly. Many servers and other network devices on the market today provide SNMP interfaces for management. It seems only natural that one would want to also have access to such devices and servers via SNMP.

The first working plugin which uses this methodology is the SNMP Blade Center plugin, which enables the IBM Blade Center. The management module (i.e. shelf controller) for the Blade Center provides a number of management interfaces. There is a web based control panel that is appropriate for human interaction, and an SNMP agent appropriate for programmatic interface. The SNMP agent exports everything needed for HPI enablement.

For instance, there are data fields that provide presence information for fans, power supplies, switches, blades, and the management modules. These are modelled as resources. Most of these items have associated serial number and version information, which fits well into Inventory Records. Thermal, Voltage, and Fan Speed sensors are all Sensor RDRs. User controllable LEDs are controls, while read only LEDs are discrete sensors.

The system chassis has an Error Log which maps nicely to the System Event Log (SEL) definition. All entries in the Error Log are provided to the user via the SEL interfaces. Those entries which can be translated to HPI defined events are, while those that don't have a related event type are passed as OEM events with the data payload being the first 31 characters of the error message. Each of these errors which translate to HPI events are also passed up through the HPI infrastructure as such.

The only real challenge for the enablement of the Blade Center was the mapping of SNMP data types to HPI types. A large number of definitions relating SNMP OIDs to specific HPI types and data elements was required. Over time this has been normalized to a set of definitions which can be applied to each blade, fan, or other resource in turn. Through this work the OpenHPI plugin API was adjusted to ensure maximum compatibility with SNMP devices.

It is expected that this work can be used by others to further enable other SNMP devices, like Network Attached Storage, programmable switches, intelligent power

supplies, or anything which allows reasonable control via SNMP. It is our hope is that the field of SNMP plugins will grow dramatically. The OpenHPI team is currently working on two more plugins in this model, and it is known that at least one outside group is using this approach for their cluster control.

6.2 IPMI and ATCA

As mentioned previously, HPI has a legacy in the Intelligent Platform Management Interface (IPMI) specification. The OpenHPI project is working diligently on enabling IPMI based servers with two different plugins. The first plugin uses the OpenIPMI library which supports a wide range of IPMI implementations. It provides access to Sensors, the Management Controllers, and Server Event Logs. There is also an OpenHPI plugin which interfaces directly with IPMI devices, which may be better suited for specific vendor implementations. Both approaches are fully functional, and part of the standard OpenHPI distribution.

With the advent of the Advanced Telecom Computing Architecture (ATCA), the OpenHPI team is committed to providing support via its IPMI plugins. Work is under way with the OpenIPMI team to enhance the software to support Shelf Managers and other ATCA specific extensions.

6.3 Subagent

In addition to mining SNMP data from existing devices, the OpenHPI team went a step further and defined an SNMP MIB which exports HPI in an SNMP friendly way. When the SNMP subagent work began, the first attempt was to produce a generic bladed MIB which would be used to describe any bladed environment. Although there would be a lot of value in this approach, it would relegate the HPI support only to bladed architectures. The complexity to do this correctly also balloons quickly.

A much simpler approach was taken, which in many ways is less SNMP-like. HPI can be thought of as a set of data structures: Resources, RDRs (all four types), EventLog Entries, and Events. The simplest, and most flexible possible representation of these items in an SNMP context, is to define a resource table, and dump all of the resources in there. The same can be done with Sensors, Controls, Watchdogs, and Inventory Data. In the end, this boils down to a small number of very large data tables.

The HPI SNMP subagent is just an HPI user application, and hence can be built against any HPI implementation. It can then turn any other HPI implementation into an SNMP source that data can be mined from. This will let OpenHPI interoperate with other HPI instances on the same cluster. As the code for both the subagent and the SNMP Client plugin are open, it will allow any other HPI implementation to use this same mechanism to interface with other HPI instances as well.

Some may argue that this is an extra level of abstraction that isn't required. There may be instances where a client application will be going from SNMP to HPI back to SNMP back to HPI back to SNMP, and so forth. This may add latency on each transition. However, SNMP defines a protocol, not a data structure. Current SNMP client applications must be written with one or more MIB in mind to access data in any reasonable way. The explosion of different hardware MIBs specific to each vendor has made interfacing with heterogeneous hardware via SNMP time

consuming. By using HPI to create a common structure for how this hardware will be modelled, the OpenHPI team has brought together the best of the legacy of the SNMP community with the unifying ideas of the HPI specification.

7 Testing

Testing and reliability is an important part of any software program, and OpenHPI is no different in this regard. Three different testing activities are happening as part of the OpenHPI test effort.

The first is the HPI conformance test. In order to claim that an application is HPI compliant one must produce a conformance test suite demonstrating that fact. The OpenHPI team started this effort when the original development effort was started. These tests are available for other HPI implementations to use as part of their own self certification. The HPI conformance test has already been adopted into the Linux Test Project, as a Linux API test.

The second effort is a new functional test. This delves deeper than the conformance test, and is specific to OpenHPI at this time, as it checks for specific expected data values. This test suite will continue to grow over time.

Both the conformance and functional tests look at OpenHPI as a whole. Although that is extremely valuable, it is hard to use these mechanisms to force edge conditions on lower level components. The only way to do this in a robust fashion is by creating unit tests for internal interfaces, and testing individual c files separate from OpenHPI as a whole. Fortunately, the automake infrastructure used by OpenHPI provides a unit testing framework for just this purpose.

Unit (or component) tests are created in a t/ subdirectory of each source directory. When the "make check" target is run, each of these are run in turn, and a total test result is reported. The gcc coverage tools are used as a gage of the code coverage that results from these tests. The unit test effort has only recently started, with plans for full unit test coverage of all infrastructure components by summer 2004, and significant test coverage of included plugins by the end of 2004.

8 Community

One of the main advantages of the OpenHPI effort is the open nature of the development. This provides rapid feedback to users on the direction that the software is developing. It also provides open forums for users to request new features, and drive the direction that OpenHPI evolves.

The OpenHPI (http://openhpi.org) website is the definitive source of infor-mation relating to the project. News relating to the project, detailed current implementation, documentation on OpenHPI, mailing list archives, and links to released and evolving source code are all provided.

The openhpi-devel mailing list is the best way to communicate with members of the OpenHPI team, as everyone watches that list closely. However feedback can also be given via and one of three public issue trackers as well. There is a bug tracker for known or suspected bugs found in the OpenHPI code. There is also a feature tracker for features that are desired for future OpenHPI releases. Any items added to these

trackers are quickly assessed, and assigned to a developer and a release number for when they will be addressed. New bugs reports and feature requests from users or potential users are *very welcome*. These trackers also provide a roadmap to near term deliverables for OpenHPI.

The final tracker is an HPI specification issues tracker. During the development of any HPI implementation, issues will be found with the specification. Sometimes they are clarifications which need an immediate interpretation, other times they are assumptions which might not hold true for all types of hardware. The OpenHPI team is committed to documenting these items which are found, and ensuring that any interpretation made by the OpenHPI team of the specification is well documented, and hopefully incorporated into future revisions of the specification.

OpenHPI is working under the "release early, release often" motto of Open Source development. The 15th of every month is release day, providing newly updated binary distributions around that time each month. Each release clearly specifies the features that have been added, and the bugs that have been resolved, most of which are references to bug or feature tracker entries, so detailed information on the issue and solution can be found by anyone.

The team at one point attempted to perform real time meetings every other week via Internet Relay Chat (IRC). However, the development team has grown to the point where it now spans East, Central, and Pacific timezones in the US, as well as China, and Germany. For this reason, email, which can be read and responded to at any hour of the day, is the preferred communication method for the OpenHPI team.

9 Future Directions

One of the obvious additions in the future would be the support for more hardware types. The project has started the process of enabling the IBM xSeries servers which have a Remote Supervisor Adapter. There is at least one external group working on cluster power control for their High Performance Cluster, which the OpenHPI team will be encouraging and helping along.

Similar to the SNMP MIB design done by the group, a CIM (Common Information Model) schema could map extremely well to the HPI specification. CIM is becoming very prevalent in the Enterprise computing space, and with the maturation of the Pegasus CIM implementation, there is now a robust open source CIM engine available to all. It many ways this modeling might be far more natural than the SNMP model. The OpenHPI team plans to work on such a CIM model this year along with other interested parties.

In the Linux world, C is still king, however few people write management applications in it. Higher level languages such as Perl and Python are preferred due to their more rapid development cycle. Although outside of the scope of the HPI interface specification, the project plans to create higher level language bindings against OpenHPI that will open up the HPI data and functions to a much wider range of application programmers.

The universality of hardware enablement, modeling in SNMP and CIM, and higher level language bindings will all help in building a community of application developers for HPI. Our hope is to have a number of management applications which

use HPI available from the OpenHPI website over the next year. The OpenHPI distribution already includes a set of HPI utilities, which are command line interfaces which use HPI to interface with hardware. There are also numerous open source clustering teams that have been waiting for something like HPI to enable much slicker install and maintenance mechanisms, and they are starting to take great interest in the OpenHPI effort. We hope to see the ecosystem for HPI user applications grow significantly during the next year.

10 Conclusions

HPI 1.0 has gone a long way to providing a common interface for hardware management. The OpenHPI effort is building even more momentum towards the adoption of HPI as an Industry standard. The open nature of the product brings HPI to an extremely wide audience. The plugin interface makes it easy for 3rd parties to HPI enable their hardware or network devices with a small amount of development effort. The integration with SNMP enables both the use of existing SNMP management software with HPI enabled hardware, and HPI management software with SNMP enabled hardware. The support for IPMI and ATCA further strengthens the use of these standards in the Linux world.

In all these ways, and many more the OpenHPI project is becoming a standard reference implementation, robust enough to use in production environments. OpenHPI is helping HPI gain acceptance and adoption in the Linux community and many industry sectors. We see this adoption increasing over time, and HPI becoming a standard system service on all Linux platforms due to the efforts of the OpenHPI project.

Quality of Service Control by Middleware

Heinz Reisinger

SIEMENS AG ÖSTERREICH,
Communication Systems and Solutions, Vienna
heinz.reisinger@siemens.com

Abstract. Assuring a negotiated quality of service for a client is an important matter of availability. In value added telecommunication servers quality of service control mainly means preferring high priority applications in situations of resource shortages at the expense of applications with lower priority.

Middleware that shall host arbitrary, heterogeneous services, which in turn are designed to serve heterogeneous applications, needs centralized control mechanisms for assignment of limited resources to services and applications. It must be explicitly emphasized, that every kind of centralized resource control itself consumes resources, which consequently reduces the system's performance. The challenge of the middleware is to find a balance between needed resource control and optimized performance.

On the other side intelligent clients shall deliberately select the transport medium to access the server that hosts their needed services, dependent on the priority and reliability requirements of the actual message.

The present contribution discusses several alternatives how servers can carry out quality of service control, bearing in mind that some alternatives must be provided in parallel because different clients have different needs.

1 Introduction

From the viewpoint of a telecommunication service provider availability is usually measured in out times per time unit. Several well-proven concepts for reducing the out time probability are in place in every middleware on the market:

- redundant hardware with automatic handover functions and optimized load balancing;
- distributed software with synchronization points and context switches;
- high available databases;
- automated backup / restore functions;
- etc.

The present contribution attends to another viewpoint, that of the telecommunication services' clients. Obviously they are less interested in overall availability of the server than in the availability of their individual applications.

Both viewpoints have one common rule: The higher the availability the more expensive are the services. In fact, even small improvements of availability usually result in considerable rises of costs, especially if the availability moves beyond the 99% margin. Different clients and even different applications of one and the same client obviously have different requirements regarding availability and want a customized balance between availability and costs.

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 61–72, 2005.
© Springer-Verlag Berlin Heidelberg 2005

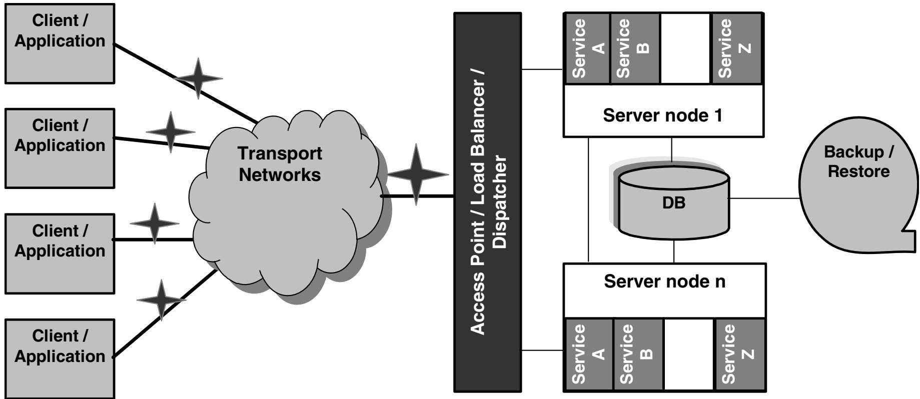


Fig. 1. Points of Interest from Client's View

Therefore the concept of a client / application specific quality of service has evolved. Clients negotiate the needed quality of service for their applications before they go on air and they select the transport medium that is appropriate for their needed quality of service. The higher the quality of service the more they are charged for message transport and processing.

2 Criteria of Quality of Service

Traditionally quality of service is an important measure for transport networks. A lot of standards for any type of networks have been published. As very coarse-grained summary of these concepts, quality of service for transport networks includes mainly

- maximum number of transmitted information units per time unit,
- average number of transmitted information units per time unit,
- maximum number of concurrent accesses,
- average number of concurrent accesses,
- maximum number of allowed transmission failures per time unit,
- average number of allowed transmission failures per time unit,
- maximum answer times,
- average answer times.

Different transport networks have different characteristics and different costs. Intelligent clients are aware of their various options and select the transport network according to the quality of service needed for every single message.

Once client requests have successfully reached a server we need another classification for quality of service. Heterogeneous service requests arrive in arbitrary intervals and are only limited predictable. Thus temporary shortages of a server's resources are inevitable. Other reasons for temporary resource shortages

are failures of redundant server units, which put more load on the remaining units. For the following considerations of the present contribution the following relation applies:

The higher quality of service an application has got guaranteed, the higher is its priority in the competition with other applications for scarce resources.

This means, high priority applications have a better chance to be served timely and especially not to be dropped in case of resource shortages.

With respect to this quality of service definition no absolute priority can be guaranteed. The priority of a concrete client's/application's request obviously is dependent on the concurrent requests of all other clients/applications. It can be only guaranteed that high priority applications are favored against ones with lower priority. Between applications with equal priority the strategy "first come, first serve" is best proven.

Typical examples for competing applications are voice dialogs versus SMS traffic or plain web browsing versus e-commerce applications.

- Voice dialogs need a guaranteed short answer time; SMS traffic is more delay tolerant. On the other hand, out times are less critical in voice dialogs.
- Both, plain web browsing and e-commerce applications, are not delay tolerant, but out times during plain web surfing can be much easier accepted than in e-commerce.

Agreements, which guarantee clients / applications a minimum number of processed messages per time unit, can be implemented by giving that client/ application a top start priority which decreases when the guaranteed number of successfully processed requests[1] is reached. The present contribution assumes generally that clients / applications must be registered together with various attributes at the telecommunication service provider's servers before they can access any services. Some attributes may specify dynamic priorities, which depend on the number of previously processed requests.

The second prerequisite is a client / application specific measurement of the number of requests. Both prerequisites are state of the art in telecommunication servers and are not further discussed in this contribution.

Typical limited resources that must be assigned to client / application requests with caution, are

- CPU time,
- memory,
- permanent storage,
- message waiting queues,
- service enablers,
- bandwidth of interfaces to service enablers.

---

[1] Of course it is assumed that the server's capacity is sufficient to process its top priority.requests at all.

3 Server View

3.1 Quality of Service Negotiation

All concepts of quality of service control have one common prerequisite. It must be strictly avoided that the majority of clients / applications issue high or highest priority requests. Such scenarios destroy every strategy. Then the few low priority requests are either not served at all or - on the contrary - executed prior to the majority of high priority requests, which must be handled "first come, first serve". The ineffective resource control puts additional load on the resources. In fact, if such scenarios cannot be avoided clients are better off with no quality of service control at all.

The most effective means for providers of servers hosting telecommunication services to avoid an inflation of high priority requests is via the price of access. The costs of high priority access must be considerable higher that that of requests with medium or low priority. Providers need administrative means to adjust the ratio between quality of service and price dynamically.

Another option is to enable only VIP clients to send high priority requests.

Clients / applications may get their relative quality of service

- either fixed when registering at the telecommunication provider's system,
- or variable per request.

If the first option is selected clients must have the possibility to dynamically change their applications' assigned quality of service.

A scale with 2 to 5 priority levels has proven to be feasible. According to our experience a higher number of levels enhances the basic load of the middleware with no further advantages for the clients.

3.2 Quality of Service Control

The SIEMENS middleware offers 3 mechanisms of resource control, which can be combined due to the concrete needs of service providers and their clients. In order to be able to select a specific combination of mechanisms a service provider must be aware of the general strategy how the middleware processes concurrent requests.

The most important measure for systems providing telecommunication services is their performance; for the service providers this is even more important than availability. The best performance can be achieved by architectures that deploy a fixed number of processes with a fixed number of threads, which resolve service requests in parallel as mutually independent as possible. Adding and removing threads and processes during service execution is very resource consuming and is avoided wherever possible. Any kind of resource control impairs the threads' independency and consequently the performance of the whole system.

The following three resource control mechanisms have different impacts on the system's performance.

Servers must keep in mind which kind of resource control a registered application can undergo principally. For instance priority control of intermediately stored messages is unacceptable for real time or near real time applications, even if they are served with a high priority.

3.3 Quality of Service by Reservation of Threads for High Priority Requests

The middleware holds a pool of threads available for processing the requests of clients / applications. In this pool a subset of threads is explicitly reserved for high priority requests. Crucial to this concept is a highly efficient dispatcher function, which screens incoming messages and assigns them a high priority or standard thread. Criteria for dispatching are

- either the message type
  (if a fixed relation between message type and priority exists),
- the identification of the sending client / application,
- or an attribute in the message header.

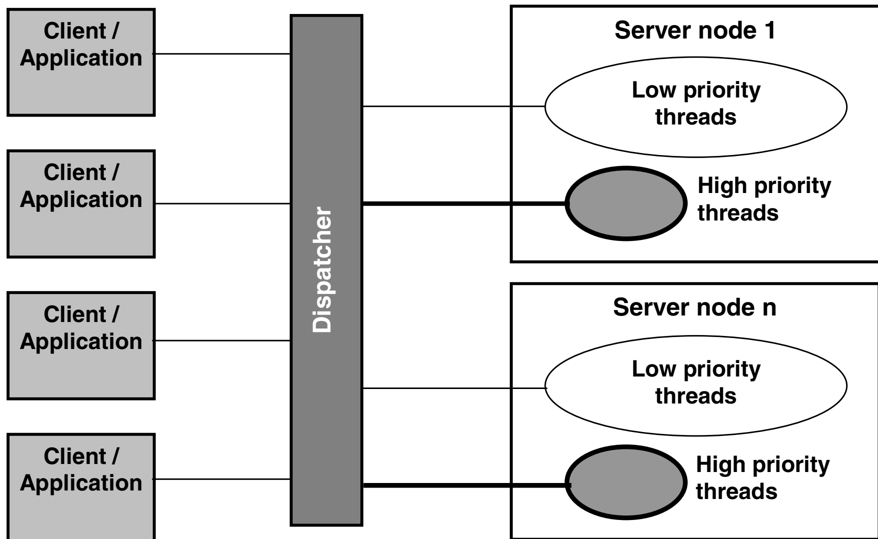


Fig. 2. Reservation of Threads for High Priority Requests

This strategy guarantees resources for a fixed amount of high priority requests regardless of the amount of concurrent lower priority requests. If the amount of high priority requests exceeds this fixed amount the remaining ones are handled as low priority requests.

Advantages. This mechanism of priority control puts the least load on the system; this means it is the most performant one. The used dispatching algorithms have proven to be sufficiently efficient. The ratio of high priority and standard threads can be easily adjusted due to access statistics.

The dispatcher can be easily configured to consider new clients/applications and message types.

Disadvantages. High priority threads cannot be used for lower priority requests; they really must be reserved. If the number of reserved threads for high priority requests is not in line with the actual average number of high priority requests the server leaves some of its capacities unused.

In practice this leads to a limitation of the possible priority hierarchy to 2 levels. Splitting up the thread pool into more subsets leaves even more threads idle if no requests of a certain priority occur.

This strategy assumes implicitly that all threads have equivalent needs for the remaining scarce resources. Although this assumption is met statistically, in some exceptional scenarios the strategy might not work.

3.4 Quality of Service by Priority Control of a Message Handling Component

This strategy uses an upgraded message handling system. As in usual message handling systems services subscribe to messages from certain applications. Subscription updates are done whenever a new client registers its applications at the server.

Incoming messages are published. As a difference to a common publish / subscribe system the upgraded message handling system prefers high priority requests. Low priority requests may be delayed or even thinned out. The criteria, which messages are to be preferred, are the same as for the dispatcher described in the previous chapter.

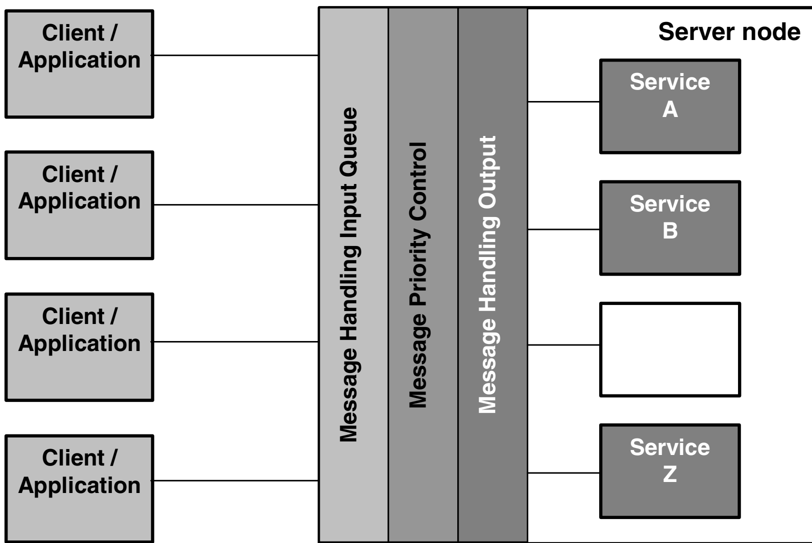


Fig. 3. Priority Control via Message Handling Component

Advantages. The message handling component has more opportunities than a dispatcher. With the expense of some more needed processing power the message handling component can carry out more sophisticated quality of service control with some more priority levels.

The overall performance loss caused by this strategy is still below 15%.

Disadvantages. A quite tight coupling between requesting clients / applications and services is needed. Configuration of this coupling is complex. For providers whose clients register and deregister very dynamically this strategy is not optimal.

3.5 Quality of Service by Priority Control of Intermediately Stored Messages

Accepting and acknowledging incoming requests is completely decoupled from processing. Incoming messages are collected in a permanent intermediate storage without any initial processing. All messages have a priority assigned; the criteria are the same as that of the other strategies. Background processes select waiting requests from the intermediate storage by their priority and by their age and execute them asynchronously.

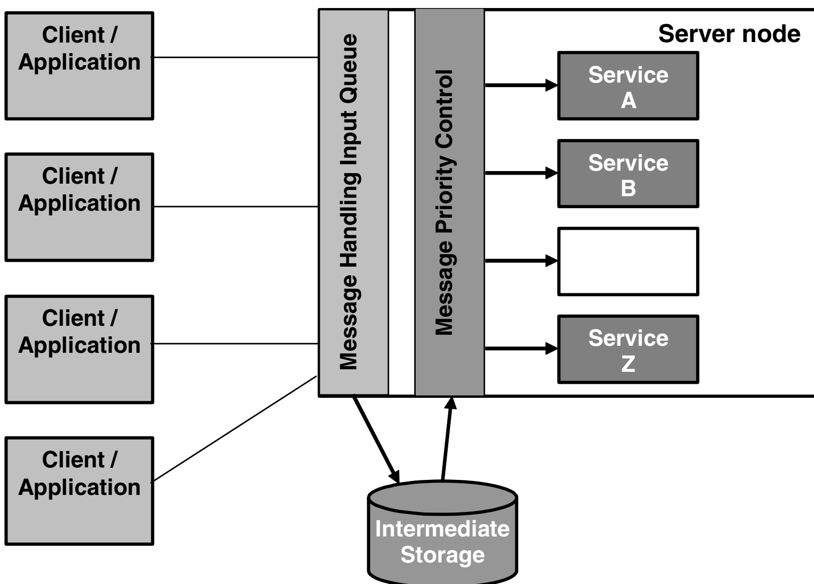


Fig. 4. Priority Control of Intermediately Stored Messages

Advantages. This strategy is especially suited for clients / applications that require that their waiting messages survive a system outage, but have no stringent answer[2] time requirements. The background processes have sufficient resources to undertake complex quality of service control. In addition to up to 5 priority levels the background processes can control sequential dependencies between distinct messages.

Disadvantages. This strategy needs a database for the intermediate storage of incoming requests. This may increase the server's costs. Because all incoming messages are first intermediately stored in the database the overall performance of the server is reduced considerably, we have experienced a performance decrease of up to 40%.

[2] Answer in this context means an indication of successful or unsuccessful processing, not just a receive acknowledgement.

For client / application requests needing very a short answer time this strategy is not suited. Optionally for messages with the highest priority this disadvantage can be overcome by exceptionally not storing these requests intermediately but processing them synchronously.

3.6 Handling of Low Priority Messages

Handling of low priority messages differs substantially due to the strategies of quality of service control described in the previous chapters.

3.7 Penalizing Low Priority Messages in Case of Resource Shortages

If quality of service control is carried out by reserved threads or by a message handling component, low priority messages are handled fully equally as ones with high priority as long as sufficient resources are available. Only if resources are lacking low priority messages are discriminated.

The best-proven strategy is to thin out messages gradually in case of resource shortages. Target is to adapt the number of actually processed requests smoothly to the amount of available resources. Oscillating should be avoided because this effect reduces the overall number of processed requests unnecessarily.

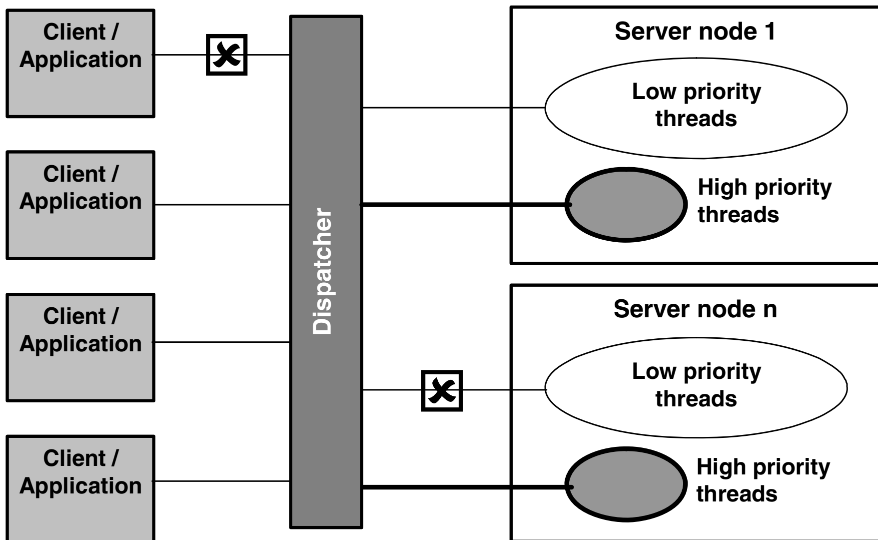


Fig. 5. Rejecting Thread Assignment to Low Priority Requests

A typical approach is to start to reject, e.g., every 4th low priority message if the server determines a resource shortage. As long as this resource shortage remains or gets even worse the ratio of rejected low priority messages rises. As soon as more resources become available again the ratio of rejected messages falls.

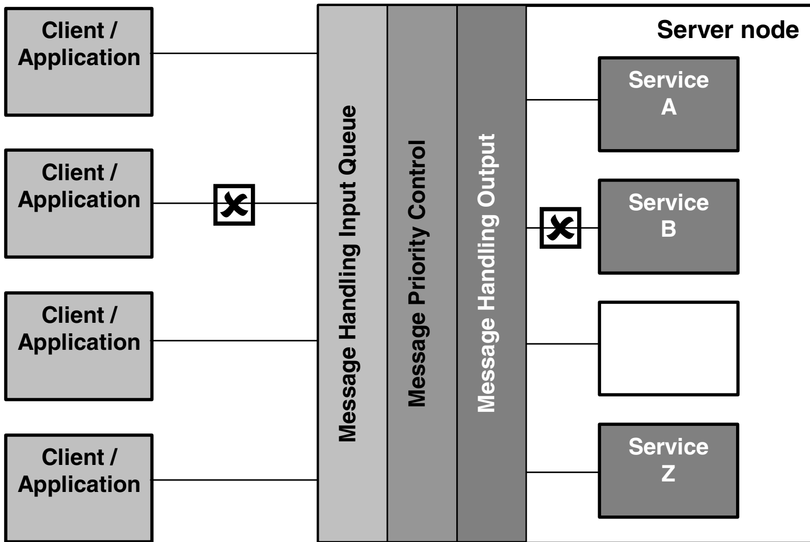


Fig. 6. Rejecting Delivery by Message Handling Component

This treatment of low priority messages is a commonly used overload handling pattern. Of course, if the server reaches a state in which 100% of low prioritymessages are blocked and needed resources are still not sufficiently available, the next escalation steps of overload handling also affect higher priority messages.

3.8 General Preference of High Priority Messages

If quality of service control is applied to intermediately stored messages low priority messages are always discriminated in favor of ones with higher priority. Whenever resources become available the server selects one of the intermediately stored requests to be processed. The priority of the request is the most important selection criterion.

Of course, it must be prevented that once accepted and stored low priority requests remain unprocessed forever as long as messages with higher priority are coming in. To this end every thread works according to a schedule, which controls the execution percentage of messages of each priority. Operators may adapt this schedule per administration.

In case of actual resource shortages a further thinning out of lower priority messages comes into effect, thus combining the strategies.

4 Client View

Usually Clients and their applications have a choice of different ways how to transmit their requests to a server. In a typical environment clients may select between

- public ISDNs,
- PLMNs (GSM, GPRS, UMTS),

- leased lines,
- IP networks,
- WLANs,
- etc.

These transport media differ in availability (dependent on the client's actual location), price - and quality of service (regarding bandwidth, latency, throughput, etc.). The main contribution clients can do themselves for the quality of the service they access is to select the proper transport medium. Clients need schedules, which control their preferred route dependent on

- their actual position,
- date and time,
- acceptable transmission costs and
- priority of a concrete request.

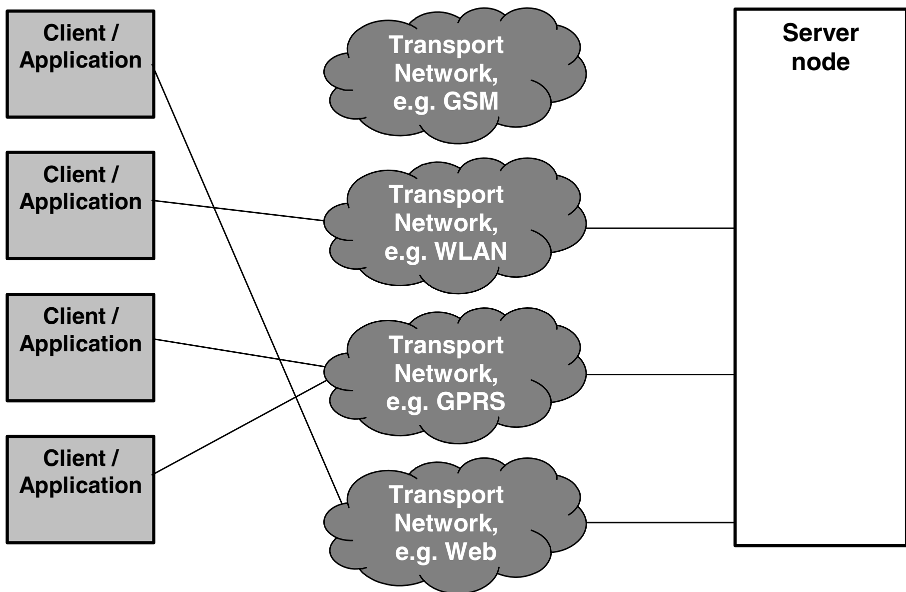


Fig. 7. Selection of Transport Medium

Such schedules are very dynamic in nature. Coverage of transport networks changes continuously, prices are very volatile due to permanent discount offers of various providers and even the quality of service offered by a provider changes. It is unrealistic that clients are able to hold their schedules up to date manually in order to retain their desired ratio between costs and quality of service.

It is up to the provider of telecommunication servers and services to provide their clients with regular or even event driven updates of optimal access routes. Schedules may be

- updated via Web serving explicit client requests;
- they may be loaded as applets into a client's browser
- or they may be delivered via mobile clients' air interfaces.

5 Summary

Assuring a convenient quality of service in a client-server session needs contributions of both sides. The server's middleware must provide efficient means of resource control and request-priority driven assignment of resources to clients / applications. The present contribution has discussed three basic strategies for this purpose:

- explicit reservation of resources for high priority requests;
- preferred delivery of high priority requests to the processing service instances;
- strict decoupling of request acceptance and request processing whereby processing is priority driven.

These strategies have different effects on the overall system performance and on the handling of low priority requests, which have been discussed thoroughly. Server middleware should be able to offer combinations of the presented strategies to meet the needs of heterogeneous clients and applications. Principles for proper combinations have been presented in the contribution.

The client must carry out an intelligent selection of available options for message transmission. It is up to the server to provide its registered clients with the needed intelligence.

The Communication Systems and Solutions branch of SIEMENS AG Österreich has issued its ServiceXpress product line for some years. ServiceXpress products consist of a generic middleware and a configurable set of services. Main areas of ServiceXpress products are

- administration and customizing of intelligent telecommunicating services,
- statistics warehouse,
- positioning solutions,
- mobile traffic services,
- mobile entertainment services.

The underlying common middleware of these services, the SX Framework, is capable of a powerful quality of service management and control based on prioritized assignment of processing resources. It is capable to provide the service's clients with stubs for intelligent transport medium selection.

6 Glossary and Abbreviations

Application: Software of a service provider or a 3rd party that accesses published services for its purposes.
Client: Clients are in a contract relation with service providers. Typically, but not necessarily, they have several applications, which are users of the provided services.

*GPRS:*General Packet Radio Service.
GSM: Global System for Mobile Communication.
*ISDN:*Integrated Services Digital Network.
Message: Used as synonym for service access.
*PLMN:*Public Land Mobile Network.
Registration: Clients and their applications must be known to a server before they are allowed to access services. During registration clients and applications specify their attributes. Relevant in the context of the present contribution are all static priority attributes.
Request: Used as synonym for service access.
Server: Computer system hosting a set of services offered by a service provider for access to registered applications.
Service access: One time use of functionality offered and published by a service provider.
Service enabler: Backend system accessed by a service to carry out its published functionality.
Service: Functionality hosted on a server being published for use by registered applications via open interfaces.
*SMS:*Short Message Service.
UMTS: Universal Mobile Telecommunications System.
*WLAN:*Wireless Local Area Network.

References

1. 3GPP TS 23.107, Quality of Service (QoS) concept and architecture;
2. 3GPP TS 23.207, End-to-end Quality of Service concept and architecture.

Benefit Evaluation of High-Availability Middleware

Jürgen Neises

Fujitsu Siemens Computers, Düsseldorf
juergen.neises@fujitsu-siemens.com

Abstract. This paper presents a benefit evaluation of high availability (HA) middleware based on a field survey. Various features which might be delivered by HA middleware have been evaluated by the participating companies. These companies consist of ISVs, solution providers, application developers, network equipment manufacturers, integrators and others. The survey is evaluated using the Kano approach. The Kano method helps to identify key features of HA middleware, which are relevant from a user's perspective: excitement factors, basic factors, and performance factors.

1 Introduction

RTP[4] Continuous Services™ (RTP4CS) [1,2,3,4] is a high availability (HA) middleware solution which conforms to the platform specification of the Service Availability™ Forum (SAForum) [5,6]. It addresses users who need to achieve a true, permanent availability of their applications.

A survey has been carried out to learn more about the customers' attitude towards the benefits of the functionalities offered by an HA middleware layer. RTP4CS has been used as reference point in this survey.

Through this survey, the following questions should be answered:

- what are the essential features of an HA middleware like RTP4CS from a customer's point of view,
- what can be the most important next steps of development according to the customer's priorities,
- who are the customers who are most interested in HA middleware features and are an optimal target group for marketing activities.

2 Approach of Survey

The survey is used as an analysis of customers' expectations regarding HA middleware. Among several models, the Kano model enables the analysis of the influence of product or service features on customer satisfaction [7,8]. The Kano model is used to determine the most valued requirements and customer expectations

---

[1] Results of a joint study with the Chair for Business and Administration with focus on Logistics (Prof. Dr. habil Horst Wildemann); special thanks to Dr. Monika Bauch for her commitment and outstanding work.

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 73–85, 2005.
© Springer-Verlag Berlin Heidelberg 2005

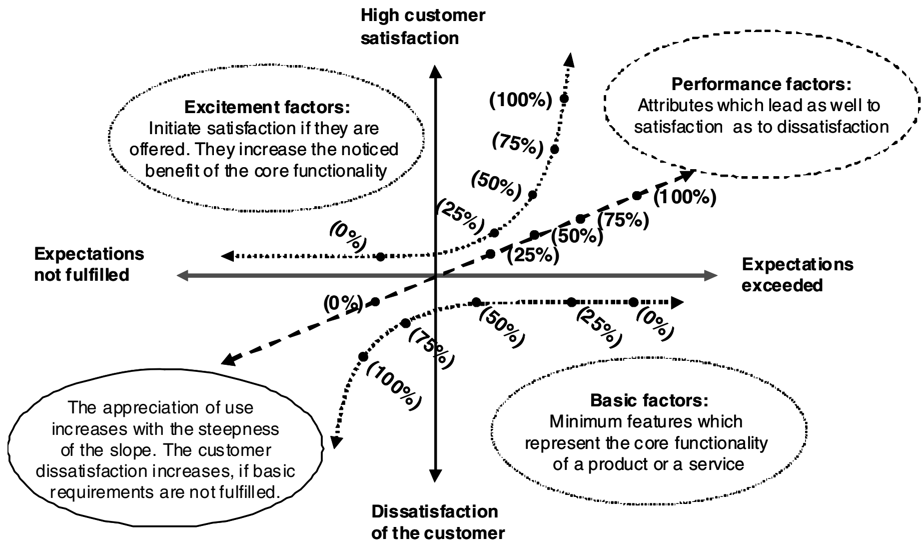


Fig. 1. Kano Model

within planning profiles of complex products and customer environments. This approach supports the goals of the survey in an optimal way.

Within the Kano model, customers' requirements on products or services are split into three classes of attributes (see figure 1). Features of these classes influence customer satisfaction in different ways:

- *Basic factors.* Basic features are essential for any customer. Lacking their fulfilment results in dissatisfaction. Hence basic features are a kind of gate to the market. Within an analysis of requirements, basic features are classified based upon their importance for the customer. A missing highly valued basic feature will result in strong dissatisfaction of the customer.
- *Perfomance factors*. Performance features are used to evaluate a product's or service's performance in comparison to a competing offer. These features evoke customer satisfaction, if requirements are fulfilled or even exceeded. However, lacking their fulfilment results in dissatisfaction. Depending on the relevance of these features, the performance of a product or service is rated within the analysis of the survey. The higher a feature is rated, the stronger is its influence on performance and satisfaction.
- *Excitement factors.* Excitement features are responsible for customer satisfaction. Missing excitement features do not result in dissatisfaction, since those features are not expected. Excitement features cannot replace basic features. However, they increase the perception of usefulness of an offered product or service. Excitement features increase differentiation and raise customer satisfaction. Analogously to basic features, excitement features are rated corresponding to their customer value.

The Kano analysis is performed in three steps. Firstly, the scope of the questionnaire has to be defined. This should cover

- Customers' expectations using product xyz
- Customers' dissatisfaction with usage of product xyz
- Customers' criteria buying product xyz
- Features which could fulfil customers' expectations in a better way
- Improvements the customers would prefer as next steps

The next step is the preparation of a Kano questionnaire. Two questions belong to each feature. The first question of each pair is a functional question referring to the customer's reaction on occurrence of a product feature. The dysfunctional second question refers to a possible non-occurrence of the same feature. There are five choices of response to each question. The questionnaire is the basis for the customer interviews. However, a good customer relationship is prerequisite to obtain valuable results.

Finally, in the third step the combination of answers to the functional and dysfunctional pair of questions are compared and put into an evaluation table to classify the customer's requirements. The results belonging to the various aspects are enlisted in a table enabling the allocation of the features to the 3 classes introduced above.

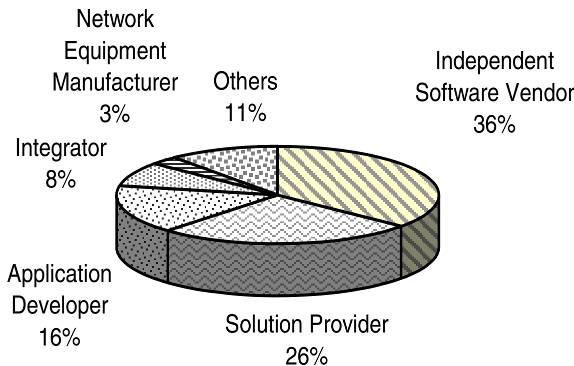


Fig. 2. Distribution of various participants of this survey

This survey took place within the project SOFTNET, part of the Bavarian research association for software engineering FORSOFT [9,10,11,12]. Within the cooperation of FORSOFT and Fujitsu Siemens Computers (FSC) the Kano method was used to assess the customer perception of RTP4CS. The FSC customers were selected at random. The results were anonymised by the University. The questionnaire consisted of 19 questions related to

- General questions
- Evaluation of usefulness of HA middleware
- Further important features
- Options on modularity

The questionnaire was filled in via Internet by the selected FSC customers. The 38 participants of the survey consisted to 37% of Independent Software Vendors, to 26% of Solution Providers, to 16% of Application Developers, to 11% of others (Software Technology Vendors, Independent Consultants, Consulting), to 8% of Integrators and to 3% of Network Equipment Manufacturers (figure 2).

With only a small number of companies responding to the poll the results of this survey has lead at least to a qualitative description of the requirements, which are important for the success of a high availability middleware. If a reliable confidence interval is required a much deeper survey with a larger number of participants is required.

3 Results of the Survey

3.1 Service Issues

Since the customer value of HA middleware should be assessed, customers' requirements on HA were a central issue within the survey. The implemented availability of the offered services is 98,46% on the average (figure 3). The availability required by the end-customer was estimated by the participants. In

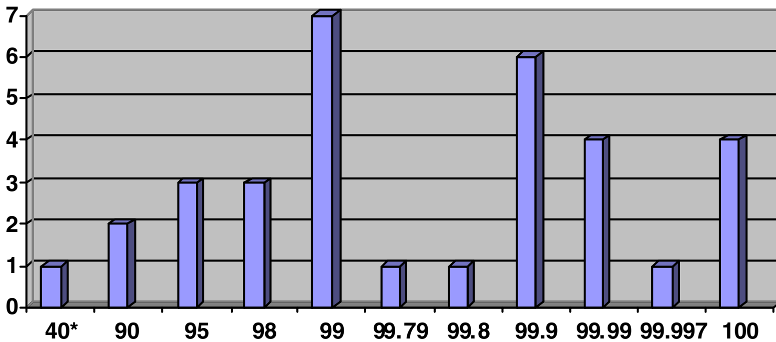


Fig. 3. Poll on realized service availability. An average service availability of 98,46% has been implemented by 32 of the polled companies. The response of 40% realized service availability is neglected due to discrepancies

this way, an average required service availability of 98.189 has been obtained (figure 4). 32% of the interviewed companies have customers who require an availability of 99.99% and more. Those companies are the main target group regarding HA middleware. These are mainly Independent Software Vendors, Solution Providers, Application Developers, and Integrators. The other 68% may give additional insight which features of HA middleware should be integrated into classical HA environments.

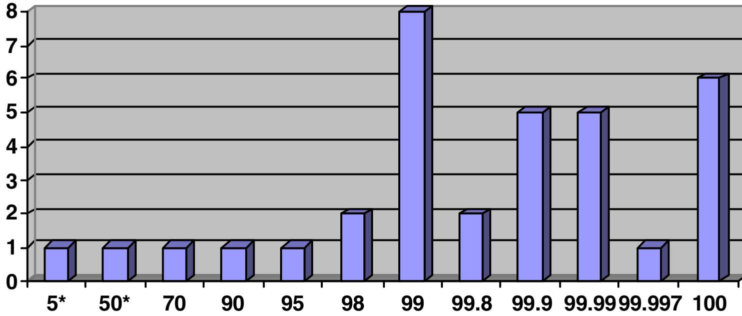


Fig. 4. Number of responses on required service availability. An average service availability of 98.189% is required by the end-customer of 34 of the polled companies. The responses 5% and 50% have been neglected in calculating the average due to discrepancies of realized and required service availability

More than 50% of the applications are developed for business processes, about 5% for consumers, and approximately 40% for business processes and consumers. High availability is required for both types of services. Nevertheless, the main issue regarding high availability is ensuring business continuity.

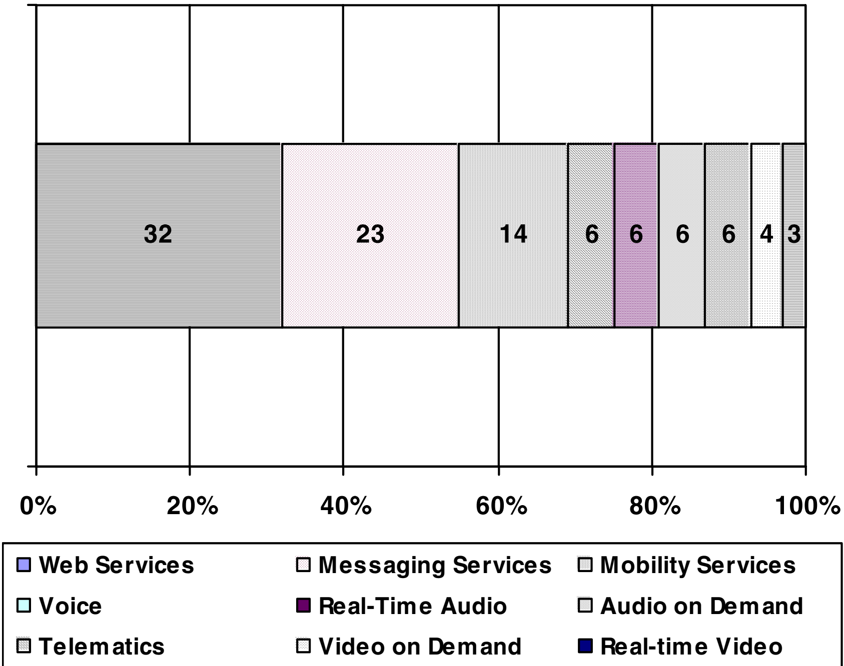


Fig. 5. Type of services

The services developed by the participating companies consist to 32% of Web Services, to 23% of Messaging Services, to 14% of Mobility Services, to 6% of Voice, Real-Time Audio, Audio on Demand and Telematics, to 4% of Video on Demand, and to 3% of Real-Time Video.

A few remarks on the questionnaire participants are in order. The number of respondents was very good (20% of all distributed questionnaires[2]). Despite a pre-selection to focus on availability aware participants, their expectations on the degree of availability varies widely. This means that the test sample becomes very small if only respondents with availability requirements beyond 99% are included in the analysis. The results of the analysis can therefore only be improved, if the survey can be repeated with more than thousand survey recipients.

3.2 Development Issues

The size of the development projects lies between one and more than 300 developers. No significant relation between project size and the required availability could be obtained. Hence, high availability issues have to be considered within the whole range of customers from SME to large enterprises.

The development projects are characterized by various other criteria. Figure 6 shows the core requirements which are relevant for most of the participating companies. Main criteria of the development projects are „Required scalability of service is high (number of users, data load)“, „Wide range of interfaces to other products, components, platform components“ as well as “We have implemented and deployed high availability applications before“.

Scalability of service is the most relevant issue in service development throughout all participating groups. However, scalability is especially important among the groups “Solution Providers” and “Others”. Furthermore, “Wide range of interfaces to other products/components/platform components” is an important issue. This feature is especially emphasized by the “Integrator” group, whereas it is of less importance to the “Application Developer” group. Experience in developing HA applications is ranked third in the characterization of development projects. All of these issues are of special relevance to the group “Network Equipment Manufacturer”.

Concerning the operating system, 47% of the interviewed persons use Windows, 26% Solaris, 24% Linux und HP-UX, 21% AIX, 11% MVS and 5% Tru64. In the future, 11% of the users of Windows intend to change to Linux. Regarding future use, the scenario changes. Linux and Windows will have the same share of usage of 42%. Hence Windows will have to share its dominance with Linux.

Due to the currently dominant market share of Solaris within the telecommunications market and the tremendous growth in Linux based applications, these two operating systems are the two most important UNIX based operating systems with a future market share of 66%. Therefore, any HA middleware should support them.

[2] Responses to the survey were motivated through prices for the fastest respondents and through a drawing between all others.

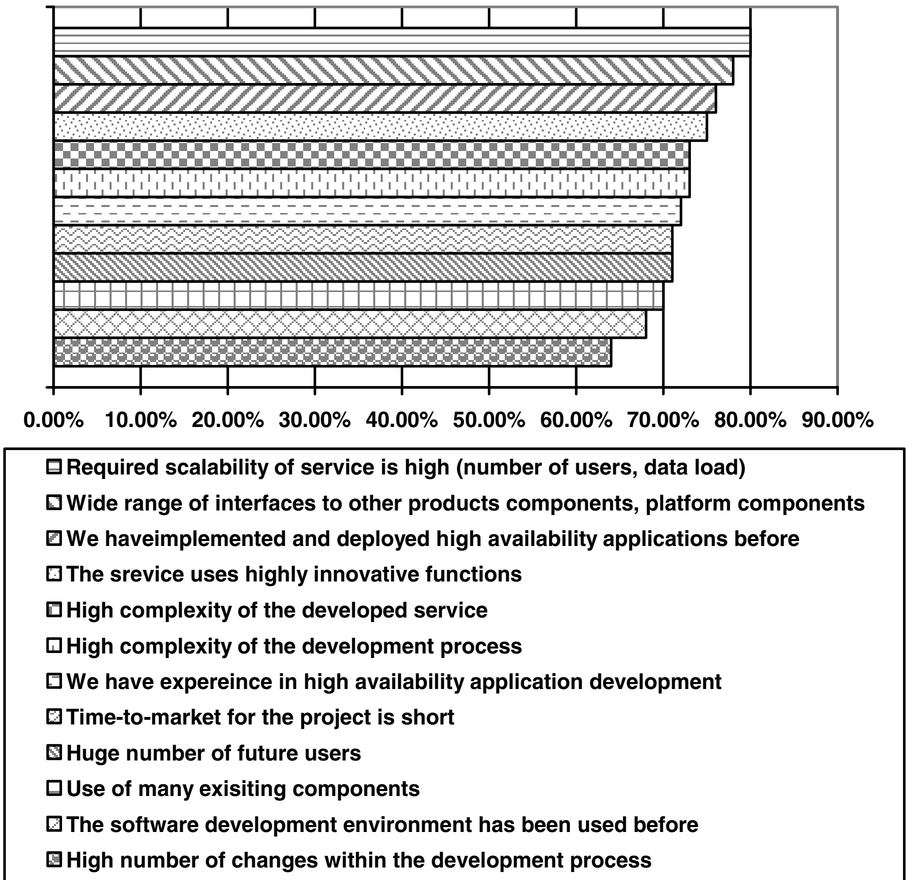


Fig. 6. Core criteria of development projects

3.3 Benefits of HA Middleware

General questions have given insight into the activities of the participating companies and hence on the environment in which an HA middleware has to be positioned. Besides that, the benefits which can be provided by HA middleware have been polled. This part of the questionnaire was divided into three categories:

- Product complexity
- Quality criteria
- Technical features

This way, the various features can be classified and their rating can be determined. The evaluation of the benefits regarding *product complexity* indicates that the most benefits are expected by five aspects:

- "Avoid unauthorised access to data",
- "Connection to external systems",
- "Execute data replication",
- "Application programming interface"

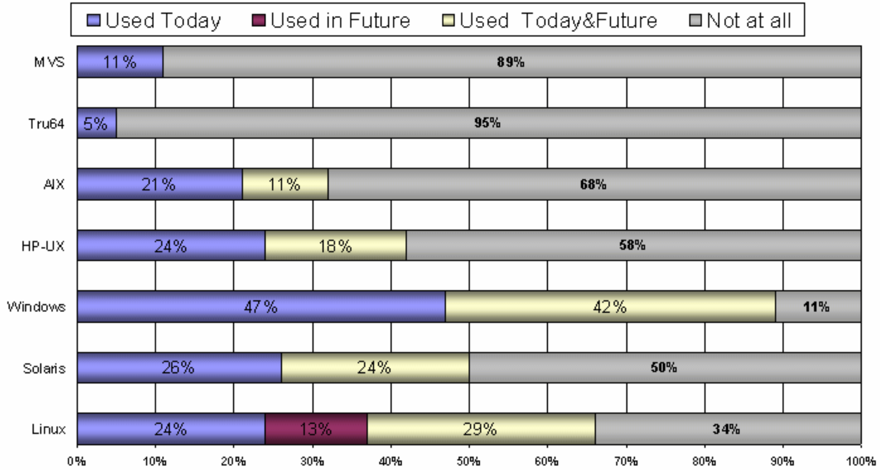


Fig. 7. Supported Operating Systems

Hence modules providing defined methods of data access and provision of specified interfaces for connectivity and data replication are highly valued. Usually such modules are laborious in programming and testing, but do not provide core functionality required for an application. Provision of such modules and a specified API is the main value proposition of any HA middleware. Thus, especially standardization of an API which defines interfaces and features will be welcomed by companies developing applications with more than 99.99% availability.

In addition to the benefits, which should be provided by HA middleware, *quality criteria* have been polled. The criteria

- "no loss of data",
- "stability",
- "short response times" and
- "support of continuous service availability"

were rated with more than 75% by the participants. Companies, whose customers require an availability equal to or more than 99,99%, set a high value on the „support of <=2 minutes down-time per year". Hence, when service availability is crucial, significantly more than five nines availability is an important quality criteria. Additionally, a combined offer of an HA middleware bundled either with a database and a protocol module or with a cluster framework is favoured.

The evaluation of *technical criteria* resulted in a high benefit for each criteria. The following criteria were used in the survey:

- Fault-resilient process communications,
- Transparent data replication,
- Online software update,
- Application programming interface,
- Graphical user interface,
- SNMP interface,
- Online hardware upgrade.

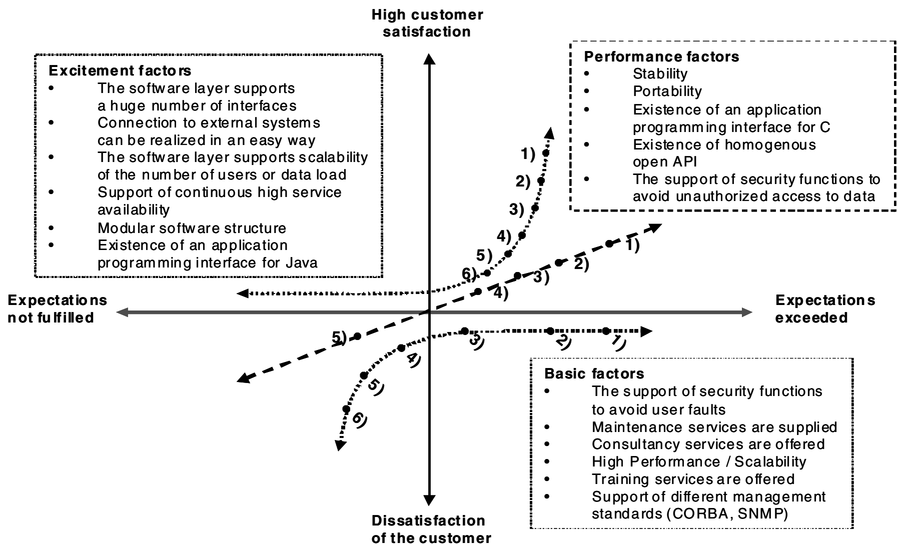


Fig. 8. Kano model of the group "Independent Software Vendor"

Also service and support functionalities deliver a high value to the customer. This means that all offered technical functionalities got the interest of the customers. Detailed evaluation using a classification of requirements will lead to a more precise understanding of the functions which really lead to customer satisfaction. For this purpose, the Kano model has been used.

4 Classification of Requirements

Therefore, a special technique – the Kano model – was used in the survey. On this basis, a profile for each interviewed person was created. The evaluation of the Kano

questions by averaging and comparison leads to a classification of features for each group of participants. Hence, for each type of company a typical Kano diagram can be obtained.

For example, the Kano model of the group “Independent Software Vendors” (ISVs) is presented in figure 8. It shows that they expect functions such as „The support of security functions to avoid user faults“. Performance factors are for e.g. stability and the existence of an application programming interface for C. Criteria such as „The software layer supports a huge number of interfaces“ and „Support of continuous high service availability“ lead to high satisfaction.

The expectations are evaluated using the Kano model according to the following two main customer groups:

- Companies with customers having availability requirements up to 99.99%. These companies are expected to put emphasis on more general features which are delivered by an HA middleware, i.e. number of supported interfaces.
- Service availability requirements of more than 99.99%: This group is the main target group of an HA middleware since the service availability targets are above five nines, i.e. service availability of less or equal 2 minutes per year (see above). Thus the participants belonging to these companies determine the features which are required at today’s market.

The companies facing service availability requirements of more than 99.99% are the main target group of an HA middleware. Hence focus of the Kano evaluation lies on this group, since the participants belonging to these companies determine the features which are required in today’s market.

The criteria „High performance / scalability“ is regarded as a performance factor by all solutions providers. On the other hand, 85% of the ISVs consider „High performance / scalability“ a basic factor. The residual 15% of ISVs see this as an excitement factor. “Application Developers” and “Others” consider this unanimously a basic factor. The “network equipment manufacturers” see „High performance / scalability“ as a clear performance factor (see table 1).

It has become clearly visible that the different groups have differing expectations and set priorities for the criteria according to their needs. It can also be stated that there are some similarities between the different user groups in their valuation.

The quality “The software layer supports a huge number of interfaces” is viewed consistently by some groups, but the view differs between the groups. “Application Developers” and “Others” see this as a basic factor while “Integrators” and “Network Equipment Manufacturers” value this a performance factor. “Solutions Providers” show internal differences: 25% see a huge number of interfaces as a basic factor respectively as a performance factor. The residual 50% consider this an excitement factor. Also, the “Independent Software Vendors” differ in their opinion. 50% consider “supports a huge number of interfaces” a basic factor, while 25% consider this a performance or an excitement factor. This means there are differences even within a “homogeneous” group.

Table 1. Kano model related to each group of participants

Requirement classes per participating Group

ISV: Independent Software Vendor

SP: Solution Provider

AD: Application Developer

Int: Integrator

NEM: Network Equipment Manufacturer

Oth: Others

B: Basic Feature

P: Performance Feature

E: Excitement Feature

Requirement		ISV	SP	AD	Int	NEM	Oth
Support of continuous high service availability.	B	X 40%	X				
	P						
	E	X 60%		X		X	
High Performance / Scalability	B	X 85%		X			X
	P		X			X	
	E	X 15%					
Stability	B	X 88%	X 33%	X 75%	X		X
	P	X 12%	X 66%	X 25%		X	
	E						
Portability	B	X 50%	X 50%			X	X
	P	X 50%	X 50%		X		
	E			X			
Modular software structure	B	X 33%					
	P						
	E	X 66%	X	X		X	
The software layer supports scalability of the number of users or data load.	B	X 70%	X 50%	X	X	X	X
	P	X 10%	X 50%				
	E	X 20%					
The software layer supports a huge number of interfaces	B	X 50%	X 25%	X			X
	P	X 25%	X 25%		X	X	
	E	X 25%	X 50%				
Existence of homogenous open API	B	X 70%	X 75%	X 66%		X	X
	P		X 25%	X 33%			
	E	X 30%					

5 Conclusion

A survey on the market requirements regarding HA middleware has been presented. HA issues are mainly driven by business continuity and a seamless operation of business processes.

Most of the participating companies require availability less than 99.99%. Thus they are only partially interested in HA middleware. These companies value the features

- “Avoid unauthorised access to data”,
- “Connection to external systems”,
- “Application programming interface”.

These features may be delivered as modules or should be integrated within a classical HA framework. The integration improves its value proposition especially compared to free cluster frameworks which e.g. are available for Linux.

The main target group of HA middleware are companies which need to fulfil service availability of more than 99.99%. These companies even require service availability of five nines and more. Those companies represent a share of 32% of all participants within this survey. Hence, the market seams to be ready for HA middleware today.

If service availability is important, the companies require the features

- Fault-resilient process communications,
- Transparent data replication,
- Online software update.

These features are essential. Furthermore, an HA middleware will be evaluated regarding the features

- Application programming interface,
- Graphical user interface,
- SNMP interface.

Based on the survey results and further research on the performance of the competitors and the expectations of the consumers, it is possible to improve the fulfilment of customer needs.

References

1. Konrad Wiesneth, Stefan Arntzen: “IT-Technology for High Available Solutions in the Telco Environment (RTP - Reliant Telco Platform).” Fujitsu Siemens Computers. 0-7803-6317-5/00/$10.00 (2000 IEEE).
2. Manfred Reitenspieß: "Providing Highly Available Computer Systems for Telecommunications Applications". In: Annual Review of Communications. Vol. 50, 1997. International Engineering Consortium. Chicago, 1997. ISBN: 0-933217-33-1
3. Resilient Telco Platform, the optimal way to implement zero downtime application. White Paper. Fujitsu Siemens Computers. 2002. http://www.fujitsu-siemens.com/rl/products/software/rtp4.html
4. B. Kellerer, M. Reitenspiess: High-Availability Middleware - Design Principles, Quality Requirements and Test Methods. Services Integrated Design and Process Technology, IDPT-2003

5. Timo Jokiaho, Fred Herrmann, Dave Penkler, Louise Moser: The Service Availability™ Forum Application Interface Specification (AIS 1.0). Service Availability Forum, 2003. http://www.saforum.org
6. Francis Tam: “On the Development of an Open Standard for Highly Available Telecommunication Infrastructure Systems”. Proceedings of the 28th Euromicro Conference, Milagros Fernandez Edt. Dortmund, Germany. September 2002. pp 347-350.
7. Berger, Charles; Blauth, Robert; Boger, David; Bolster, Christopher; Burchill, Gary; DuMouchel, William; Pouliot, Fred; Richter, Reinhard; Rubinoff, Allan; Shen, Diane; Timko, Mike; Walden, David. "Kano’s Methods for Understanding Customer-defined Quality", In: Center for Quality Management Journal, Vol. 4 (Fall 1993), pp. 3 - 36.
8. Matzler, K. u. H.H. Hinterhuber: How to make product development projects more successful by integrating Kano’s model of customer satisfaction into Quality Function Deployment, in: TECHNOVATION. The International Journal of Technology and Innovation Management, 1998, Vol 18, No. 1, pp. 25-38
9. Wildemann, H.: Kostenmanagement Software - Leitfaden zur Unterstützung einer marktorientierten Produkt- und Prozessgestaltung, 5. Aufl., München 2004
10. Wildemann, H.: Qualitätsmanagement in der Softwareentwicklung- Leitfaden zur Analyse und Verbesserung von Produkt- und Prozessqualität, 4. Aufl., München 2003
11. Wildemann, H.: Prozessgestaltung in der Softwareentwicklung- Leitfaden und Tools zur effizienten Entwicklungsprozessgestaltung in der Softwareentwicklung, 4. Aufl., München 2003
12. Wildemann, H.: Software-Projektmanagement - Leitfaden und Tools zur Planung und Abwicklung von Softwareentwicklungsprojekten, 4. Aufl., München 2003

A Measurement Study of the Interplay Between Application Level Restart and Transport Protocol

Philipp Reinecke[1], Aad van Moorsel[2], and Katinka Wolter[1]

[1] Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
{preineck, wolter}@informatik.hu-berlin.de
[2] University of Newcastle upon Tyne, School of Computing Science,
Newcastle upon Tyne, NE1 7RU, United Kingdom
aad.vanmoorsel@newcastle.ac.edu

Abstract. Restart is an application-level mechanism to speed up the completion of tasks that are subject to failures or unpredictable delays. In this paper we investigate if restart can be beneficial for Internet applications. For that reason we conduct and analyze a measurement study for restart applied to HTTP GET over TCP. Since application-level restart and TCP time-out mechanisms may interfere, we discuss in detail the relation between restart and transport protocol. The analysis shows that restart may especially be beneficial in the TCP set-up phase, in essence tuning TCP time-out values for the application at hand. In addition, we discuss the design of and experimentation with a proxy-based restart tool that includes a statistical oracle module to automatically adapt and optimize the restart time.

1 Introduction

Internet applications require effective ways to deal with unpredictable and highly fluctuating response times. Recently, restart has been proposed as a technique that may achieve that goal. Restart simply implies that an application retries an attempt if no result returns before some time-out. The applications researched range from Internet agent executions [4, 9] and web crawlers [9] to randomized database queries [13] and randomized algorithms [1, 8]. In this paper we investigate whether restart may work for straightforward HTTP GET, and how restart mechanisms interact with and relate to transport protocols (particularly TCP).

It is especially important to understand the inter-workings of application level restart with underlying transport protocols. An an example, consider a web browser's reload button, which most people routinely use to retry downloads that (seem to) halt. In essence, pushing the reload button is itself a restart mechanism, since it terminates the previous download and starts a new one [11]. Moreover, as one can read in detail in [7], pushing the reload button corresponds to 'overruling' a time-out value within the TCP protocol (namely the Retransmission Timeout

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 86–100, 2005.
© Springer-Verlag Berlin Heidelberg 2005

(RTO)). Since the TCP protocol is core to the well-functioning of the Internet, interfering with TCP may be a dangerous play. In the worst case, if restart is used wide-spread by Internet users, overall performance may deteriorate, as discussed in [9] and in a broader context in for instance [2, 6].

This paper provides a measurement study to investigate at what stages of HTTP downloads one could apply restart. Restart decreases completion time of a job if the completion time after restart is less than the remaining completion time without restart. To find out whether this is the case, we first examine if consecutive attempts are independent and identically distributed (i.i.d.). If attempts are i.i.d, it is likely that they better respond to restarts than if they are positively correlated. In addition, if attempts are i.i.d., we can compute the restart time that minimizes the expected completion time. If the optimal restart time is finite, we conclude that restart reduces the completion time of the job.

From the analysis in this paper we conclude that successive downloads to different URLs are i.i.d, and that downloads to the same URL are i.i.d. in a majority of the cases. The fact that successive completion times are largely independent is a somewhat surprising and very Internet-specific fact, but supported by other measurement studies [7]. It also turns out that during all phases of HTTP GET restart improves completion time. However, if we consider retries to the same URL, restart is primarily beneficial because attempts fail with a certain probability. If we separate out failures, the completion time distribution of TCP connection set-up benefits from restart in only a small minority of the cases. We therefore conclude that (when applied to a single HTTP GET application) restart is especially useful to avoid very long connection set-up times that arise because the RTO time-out value is not optimal for HTTP GET.

We note that the above conclusions apply to the case that only one application executes restart. All other active Internet applications are assumed to not change their behavior. We refer to [9] for a simulation study of the case that multiple or all applications apply restart. Note also that the restart approach is a black-box approach, as is our measurement study. We have no data about the particulars of network or server status, and thus cannot draw conclusions regarding the relation between completion times and server or network load.

Applying restart would be greatly helped by a software module that can be used for various applications, and flexibly adapts the restart time based on the application at hand and the performance measured. To that end, we designed an on-line oracle with algorithms that dynamically adapt and optimize the restart time. The results we outlined above have all been obtained using this restart tool.

The work in this paper is similar in purpose to the experiments in [14]. We subscribe to the conclusion from [14] that restart can substantially improve connection set-up, especially for the tail of its completion time distribution. We add to that a to-the-point discussion of the inter-working between TCP and restart, computation of optimal restart times for our specific experiments, and the design of an adaptive restart software tool.

2 Experiment Design

In our experiment we gather a large set of data reflecting performance of an application that uses HTTP GET, the most obvious example being the web browser. For each individual web page, a three-stage process is to be considered: (1) IP address resolving, (2) connection establishment and downloading of data comprising the page, and (3) page rendering by the browser. In our experiment we focus on step (2), arguing that the first step usually happens only once for each host, and the time taken by the third can in most cases be neglected.

Within step (2), we concentrate on three important aspects:

1. TCP connection set-up time (hereafter referred to as CST).
2. Download time of a single object in a web page, e.g. a picture (object download time, ODT).
3. Time from beginning to end of a page download (total download time, TDT).

Note that the total download time includes connection set-up as well as a number of object downloads. Each object download may include a TCP connection set-up. If HTTP/1.1 is being used, downloads from the same IP address can use the same TCP connection, thus alleviating the need for multiple TCP connection set-up phases.

Our experiments consisted of two stages, where the point of the first one was an investigation into the nature of time data encountered downloading web pages, while the second aimed at examining a possible improvement doing restarts.

2.1 First Stage

We carried out the experiments of the first stage in two phases, first downloading a large number of pages and afterwards concentrating on a few interesting examples.

Host List Construction. To obtain a large number of samples for the first step we built a list of web page URLs to be downloaded. We repeatedly fed entries of a word list into the Google search engine (http://www.google.com), requesting to be shown the first 100 hits in the reply. From this page, all linked URLs were extracted and sorted in lexicographic order. We then isolated the host name components and removed duplicate entries. As far as possible, we also checked by informal means that the selection of hosts is not limited to any geographical area or part of the Internet. Altogether we hope to have created a more or less random set of URLs with which we conducted our experiment.

We used three machines running Linux as clients. Two of these were connected to the Internet by 768/128 kbit ADSL, the third one by 100Mbit Ethernet. Iterating through the list of hosts a client downloads each server's index page exactly once. This yields three sets of data, with characteristics as in Table 1. Note that the number of objects listed under ODT is larger than the number of TCP connections listed under CST, since HTTP/1.1 allows that TCP connections can be reused for multiple objects from the same IP address.

Table 1. Constructed data sets of phase 1

Data set	# of CST $\neq 0$	# of ODT samples	# of TDT samples
I	234848	799265	56117
II	231793	794324	55397
III	232525	795986	55558

As the second phase we then selected 309 URLs to be examined further. These URLs were chosen based on the findings of the first run; we selected those that (1) consisted of a minimum of 50 objects and (2) provided several connection setups (i.e., probably did not use HTTP/1.1). These criteria were based on our decision to concentrate on CSTs. Hence, to draw reliable conclusions, we needed a large number of CST samples. URLs from this list were then repeatedly downloaded in batches of 25, each URL 10 times in a row, and the whole batch 10 times as well. Entries that failed as well as those that had already provided at least 1000 CSTs were removed from subsequent iterations in order to speed up the experiment. This phase was run on one Linux system using ADSL.

2.2 Second Stage

In the second stage we compared performance of connections through our proxy with and without doing restarts at connection setup. Here we used the same method as described in the second step above, but employing the proxy in-between. This stage was conducted in two ways, (1) using a list of 17 URLs that exhibited a relatively large number of CSTs above 3 seconds and (2) using 25 URLs without CSTs in this range, but with a high standard deviation of samples, both based on observations in the second phase of the first stage.

2.3 Tools Used

For all our experiments we relied on a modified version of the GNU *wget* utility. We modified the program's usual output to display time stamps consisting of seconds and microseconds (as given by the *gettimeofday()* C library function), at the start and end of (1) connection set-up, (2) download of a page component (possibly including more than one connection set-up), and (3) download of a whole page. To obtain such detailed data, we ran *wget* with the *'-p'* option, thereby downloading all elements necessary to render the page, i.e., requesting the HTML code as well as any objects referred to.

3 Analysis for Multiple URLs

For some Internet applications restart implies connecting with a different URL at each attempt. This can for instance be the case with web crawlers [9] or certain agent applications that find quotes from e-commerce web sites. For such

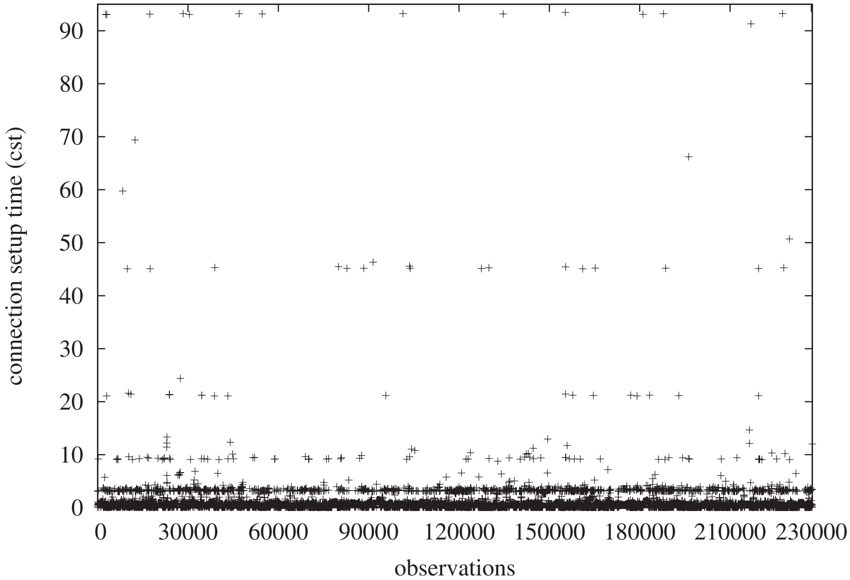


Fig. 1. All sampled connection set-up times

applications, one needs to analyze data for varying URLs. We focus mainly on data set II of Table 1, but the results for the other data sets are similar. Fig. 1 shows the time for connection set-up of the close to 240 thousand samples of data set II.[1] The plot nicely visualizes the role of the retransmission time-out in TCP connection set-up. The RTO is used to trigger a retransmission of a connection attempt if no acknowledgment has been received in time. In Fig. 1 one recognizes the RTO values through the 'stripes' formed a little above RTO values: first after 3 seconds, upon the second trial after 9 seconds, then 21, 45 and 93 seconds, exactly according to specification [7].

Importantly, almost all connection set-up times are less than 1.0 seconds. That is, either the connection set-up succeeds quickly, or one waits for the RTO timer to trigger a new attempt. Very rarely (roughly 1 in 5000 attempts) does a connection get established after one second, but before the next RTO time-out. In addition, out of the roughly 230 000 observations, about 1200 'fail' (i.e. they take longer than 3.0), thus leading to retries. Hence, the CST distribution is with probability 0.995 distributed as given by the samples, and with probability 0.005 it fails. If we compute the optimal restart time for this data set (see Appendix A and [11]), we obtain that restarts should be done every 0.43 seconds. Such computation, however, assumes independence of consecutive tries, a fact we investigate now using the statistical package R [12].

[1] For practical reasons we plot in several figures throughout this paper only a subset of the thousands of samples. In none of these cases this changes the visual experience when interpreting the figures.

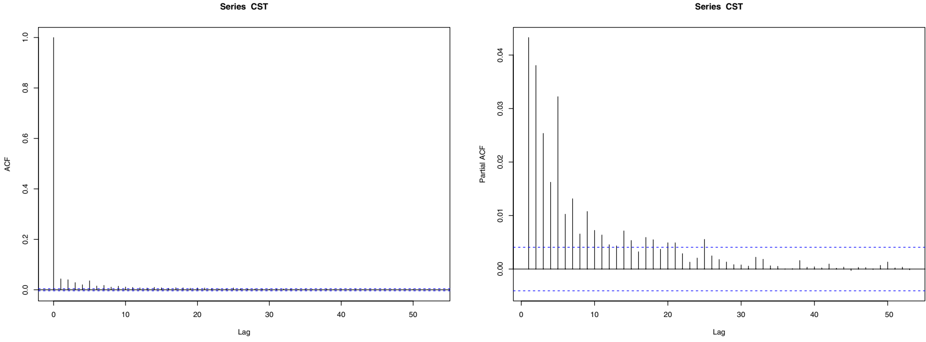


Fig. 2. Lag versus autocorrelation and partial autocorrelation function for CSTs to multiple URLs

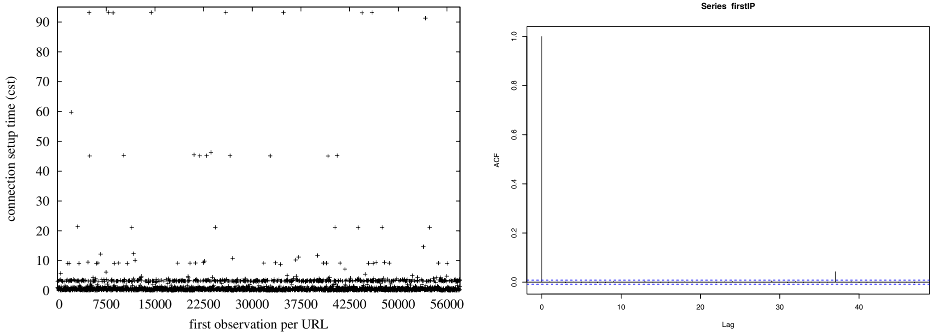


Fig. 3. Connection set-up times for the first connection per URL download, and its autocorrelation function over the lag

We compute the autocorrelation function and, since it provides better visual evidence, the partial autocorrelation function (see Appendix B for statistical background information), as shown in Fig. 2. Both figures show that the series exhibits correlation, since the values of the autocorrelation function are not close to zero for lower lags.

We believe that part of the explanation of the correlation between consecutive connection set ups in data set II is the clustering effect introduced by the different URLs: URLs are accessed sequentially and connections to the same URL tend to take times similar to each other. From how it is assumed to occur one expects it to disappear if for each download in data set II only the first CST is selected. Fig. 3 shows the results (roughly 56000 samples). We see from Fig. 3 on the left side that the first set-up time to each server has similar characteristics as in the overall sampling of connection set-ups in Fig. 1. Even though the pattern in the CST is the same for all sampled CSTs as for the first connection set-up to a location, the dependence characteristics are not. The autocorrelation, shown in Fig. 3 on the right side, is very close to zero. Note the outlier at lag 37, which we

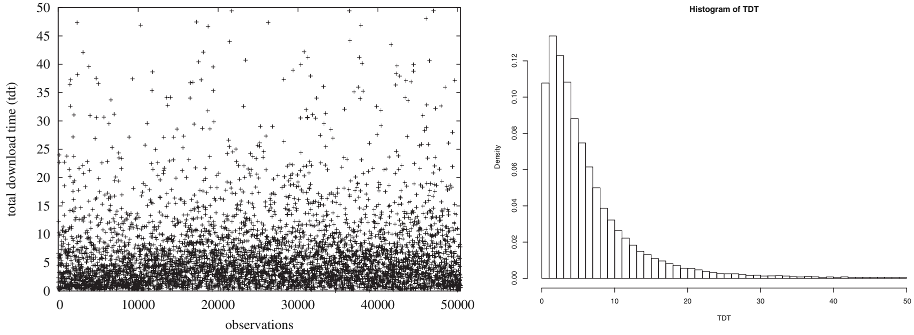


Fig. 4. Scatter plot and histogram of the total download time

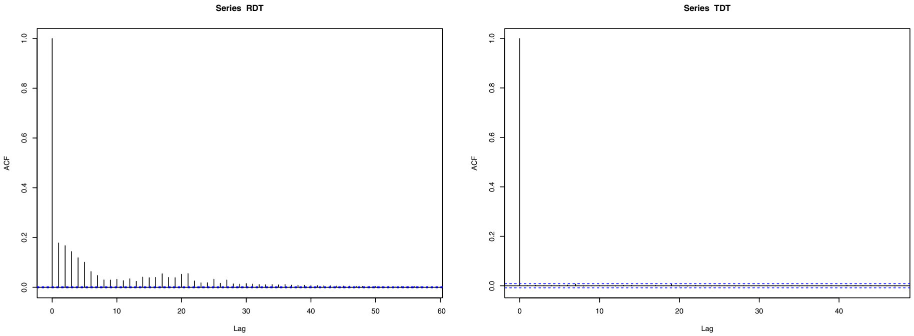


Fig. 5. Autocorrelation function for the object download time and the total download time

cannot explain – each of the other two data sets also exhibits one such outlier. In the first data set it is for lag 2, in the third for lag 28. In general, however, we conclude that this data is uncorrelated and hence is more likely to be amenable to restart.

The connection set-up is only the first step in the download of a page. The final objective of restarting a download is to receive the whole page faster. Therefore we are interested in the total download times. In this case, the results we obtain show fewer obvious patterns. This is because the file sizes vary greatly, and also the number of objects belonging to one page is quite diverse. To give a picture of the total download times we sampled, Fig. 4 shows TDTs for our data set II. In addition, the right side of Fig. 4 shows the probability density of the data in the left plot. Note that both figures display samples with a download time of at most 50 seconds, thus omitting the 726 largest values. It is interesting to note that if one applies the expressions from Appendix A to this data, one finds that the optimal restart time is about 8 seconds.

Of course, it is not at all clear if the assumptions under which the optimal restart time can be computed apply to the data in Fig. 4, and we therefore study the correlation of consecutive downloads. From Fig. 5 we see that there is quite strong autocorrelation in the object download time, probably since many objects belong to the same index page (up to 200 objects on one page). However, the total download times of consecutive attempts seems highly uncorrelated.

4 Analysis of Individual URLs

Until now we discussed the situation of multiple URLs. Restart resulting in HTTP GET actions to rather randomly selected URLs may be useful for some Internet applications, but, obviously, for the most common, a restart would use the same URL as the previous one. This is certainly true for web browsing. Therefore, we must do the statistical analysis for consecutive requests to the *same* URL, which we did in the second phase of this stage.

Drawing conclusions for the correlation of individual URLs is much more complicated than that for all URLs together. If we consider connection set-up time, about two-thirds of the URLs can be said to be uncorrelated. Of this some exhibit no correlation at all. The majority shows some lags with positive correlation, but we do not think this indicates serious dependence, as we discuss below for one typical example. The correlation exhibited by about one-third of the URLs can be explained in various ways. For some URLs something drastically changed during the experiment, resulting in a shift up or down of all CST values – in such cases correlation is extremely high (this accounts for about one third of correlated URLs). We also identified URLs for which connection set-up times show multi-modality. That is, CSTs are roughly equal to one of several values, possibly because of servers in a server pool with different speed. Finally, we noticed some URLs with periodicity in the results.

As an example, we discuss a typical URL, namely http://www.jp.arm.com/. We refer to this URL as 'URL 29'. Fig. 6 shows the CSTs for URL 29 as well as the autocorrelation function. Note that the confidence interval here is much larger than for earlier shown data, because the number of observations is comparatively low. We see from Fig. 6 that consecutive CSTs to the same URL are surprisingly independent.

Fig. 7 shows the object download times and the total download times for requests for URL 29. The autocorrelation function for ODT is similar to that for CST in Fig. 6. We can compute for URL 29 what the optimal restart times are based on the data for CST, ODT and TDT. We find that for connection set up, one would restart every 0.5 seconds, for objects one would also restart every 0.5 seconds, and for total download of URL 29 one would never restart.

If we consider CSTs, it turns out that the dominant reason for which one would do restart is when connection set up fails. For all our URLs an optimal restart time with a value of less than three seconds exists, if the CST includes RTO expirations. In the examples for which the RTO timer never expires, restart pays off far less frequently. In particular, for only about 10 percent of URLs

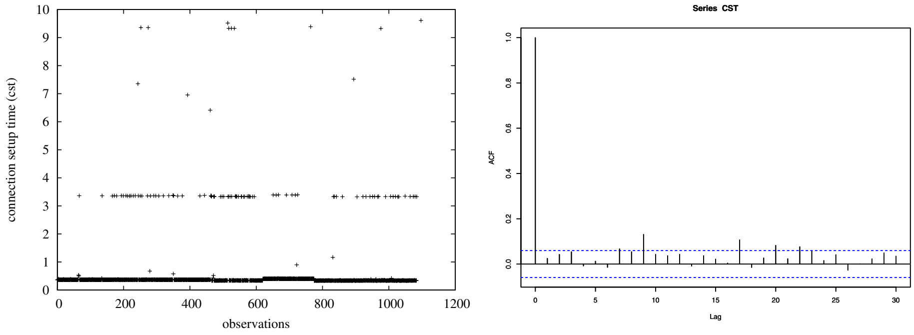


Fig. 6. Connection set-up times and autocorrelation function for URL 29

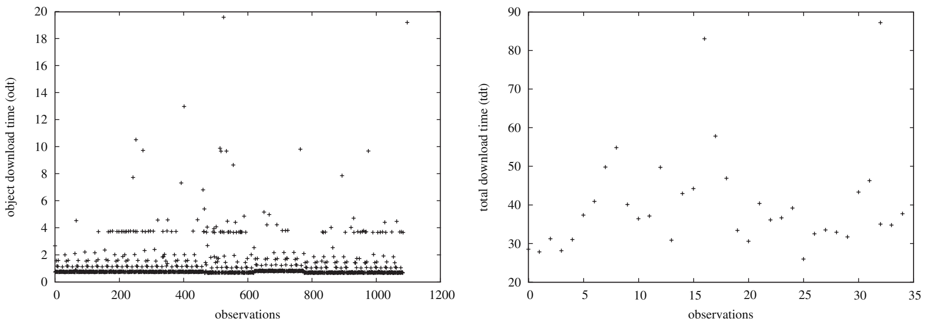


Fig. 7. Object download time and total download time for URL 29

without major correlation, restart is beneficial if no RTO timers expire. We note, however, that these results are very sensitive to the tail of the completion time distribution, and are therefore hard to estimate accurately.

5 On-line Optimization of Restart Times

To determine if restart indeed improves the set up and/or download times for HTTP GET, we need to implement restart and apply it to the URLs of our test set. Therefore, we designed and implemented a client-side HTTP-based proxy capable of dynamically adapting restart times.

5.1 Proxy Design

As depicted in Fig. 8, the restart proxy consists of the core proxy part and a general sample data collection/computation part, which we call the Timeout Oracle. When the proxy, after accepting a request from the client, connects to the appointed server, it measures the time elapsed in the set-up of this TCP connection. Every connection set-up duration is limited by a timeout. In case an attempt takes longer than the timeout, it is aborted and retried, possibly

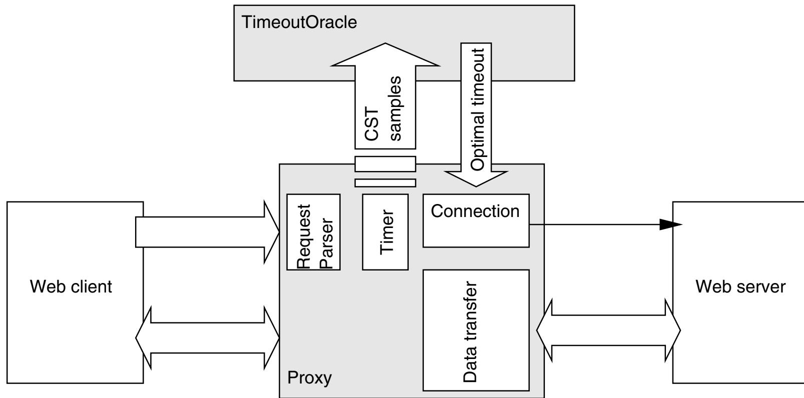


Fig. 8. Proxy design

with another timeout. This may repeat up to ten times, after which the proxy gives up.

The CST samples obtained by measuring successful as well as timed-out connection attempts are fed into the Oracle, which in turn provides a new recommendation on an optimal timeout for later connection set-up attempts. It does this based on the sample-based estimators given in Appendix A, which use results from [10, 11]. The optimization target is the first moment of connection set up time, and in the formulas we assume unlimited number of retries (we could make this more precise for the limit of 10 retries, but as some results in [11] indicate, one would not expect this to make an important difference). After connection set-up, the proxy simply forwards HTTP requests/replies between server and client. For the moment, we do not implement restart during data transfer, but we hope to do this in the future.

Parameters of the Oracle. The Timeout Oracle implements the on-line algorithm as outlined in the appendix. In principle, the Timeout Oracle can be used by any application, thus providing a general optimal timeout computation facility to applications. Its parameters are the maximal timeout (set to 24,000 ms in our experiment), the number of buckets ($H = 100$, see Appendix A) and the penalty, i.e., the time needed for a restart (somewhat arbitrarily chosen to be $c = 100$ms). When computing a new timeout value, the Oracle always uses all samples collected so far. It can be argued that this could potentially lead to slower adaptation of optimal timeout to current samples after a high number of samples have been entered. Tuning the restart time needs to be studied further but is not the topic of this paper.

We tested the usefulness of the proxy in the second stage of our experiments. Therein, we use CST, ODT and TDT as performance metrics: ODT and TDT are measured through *wget* as before, but CST samples are now taken from the proxy. In the case of restarts, all consecutive CSTs during the establishment of a TCP connection are summed to reflect the actually elapsed time.

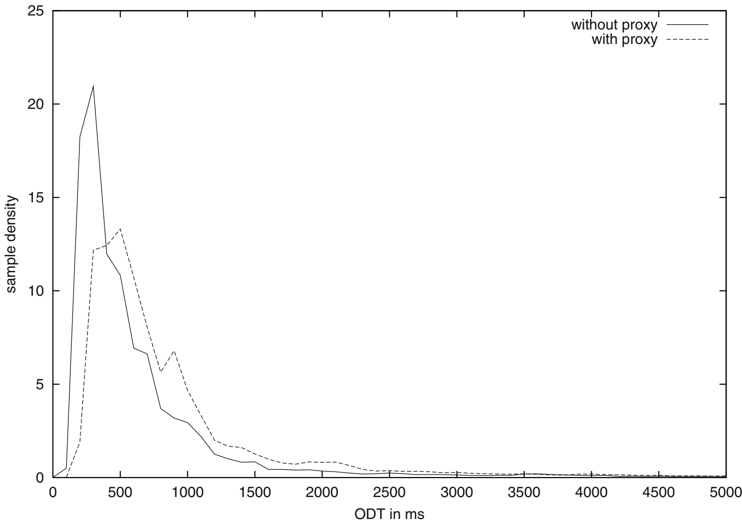


Fig. 9. Comparison of ODT samples with and without the proxy

Before discussing restart for connection set-ups, we look at the overhead the proxy introduces when downloading web pages. The proxy requires setting up an additional connection to the proxy and the proxy also uses CPU cycles to compute the optimal restart time and to handle the request. Fig. 9 gives the overhead for ODT, and the same can be demonstrated for TDT. (Note that the values on the y-axis of Fig. 9 are an artifact of the way we plotted the densities, and have no physical interpretation.) One sees that the overhead is not that drastic, so we hope that our experiments with the proxy will be representative for other future implementations of restart, for instance within the web browser.

5.2 Results

Table 2 shows results for the adaptive restart proxy, for CSTs of 16 URLs (URL 8 is omitted in the table since we no longer could connect to it halfway through our experiments). For each URL we have an equal number of samples with and without restart (about 1000 samples for each URL). We see that restart gives clear advantages for the tail of the connection set-up time distribution. In the two middle columns there are many more set-up times of more than 3 seconds without restart than with restart (average of 4.3 per thousand CSTs and 0.6 per thousand CSTs, respectively). In other words, the TCP RTO timer is set too high (at 3 seconds) to be efficient, and our restart mechanism improves considerably on the results with RTO. If we consider the mean set-up time, the difference is much less clear. Interesting are the URLs without RTO timeouts (URLs 2, 3, 7, 12, 16 and 17). We see that for these URLs set-up times with

Table 2. Restart results for CST

URL	Mean CST w/o restart	Mean CST w. restart	# > 3 sec w/o restart	# > 3 sec w. restarts	# RTOs	# Restarts
1	353	320	11	0	11	1
2	233	254	0	0	0	1
3	246	242	0	0	0	1
4	244	232	3	0	3	0
5	376	370	2	0	2	1
6	552	513	16	1	16	5
7	291	290	0	0	0	0
9	729	109	7	1	7	16
10	115	104	9	3	9	11
11	200	197	1	0	1	1
12	193	194	0	0	0	1
13	116	113	9	1	9	19
14	107	115	6	1	6	19
15	113	107	5	0	5	6
16	88	117	0	2	0	12
17	424	438	0	0	0	0
average	274	232	4.3	0.6	4.3	5.9

and without restart are roughly equal. The most noticeable exception is URL 16, for which no RTO expires, but for which restart triggers reconnection 12 times. It seems that for this example, the restart time was set too tight, because the average CST with restart is much higher than the average CST without restart.

For URL 9, Table 2 shows that the mean CST without restart is far worse than with restart. This happens when the TCP RTO timer expires several times in a row, increasing to 9, 21, 45 or 93 seconds, subsequently. Our restart mechanism does not increase so drastically (the algorithm discussed in Appendix A does in fact lead to an increase of the restart time, but at a far slower pace than the RTO increase). Of course, we have to realize that in such cases the network might have been down temporarily, also rendering quicker restarts unsuccessful (or even harmful if overload conditions caused the network problems [9]). Such effects can not be traced back using our data sets. Note furthermore that our data comes from the case that only a single application applies restart, leaving the rest of the Internet unchanged. In [9] the authors analyze network effects if multiple agents apply restart simultaneously. Moreover, using our on-line adaptive algorithm, there is more analysis to be done to assure the stability of such a control algorithm. This is all for future work.

In conclusion we see that our experiments indicate that the average connection set-up time changes little with or without restart, but that very long TCP connection set-up times can be avoided using restart.

References

1. H. Alt, L. Guibas, K. Mehlhorn, R. Karp and A. Wigderson, A Method for Obtaining Randomized Algorithms with Small Tail Probabilities, *Algorithmica,* Vol. 16, Nr. 4/5, pp. 543–547, 1996.
2. D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker, "Dynamic Behavior of Slowly-Responsive Congestion Control Algorithms," in *Proceedings ACM SIGCOMM 2001*, San Diego, CA, USA, Aug. 2001.
3. P. Brockwell and R. Davis, *Time Series: Theory and Methods*, 2nd Edition, Springer Verlag, New York, 1991.
4. P. Chalasani, S. Jha, O. Shehory and K. Sycara, "Query Restart Strategies for Web Agents," in *Proceedings of Agents98,* AAAI Press, 1998.
5. W. Cochran, *Sampling Techniques*, John Wiley, New York, 1977.
6. S. Floyd and K. Fall, "Promoting the Use of End-to-End Congestion Control in the Internet," in *IEEE/ACM Transactions on Networking,* Vol. 7, No. 4, pp. 458–472, 1999.
7. B. Krishnamurthy and J. Rexford, *Web Protocols and Practice,* Addison Wesley, 2001.
8. M. Luby, A. Sinclair and D. Zuckerman, "Optimal Speedup of Las Vegas Algorithms," *Israel Symposium on Theory of Computing Systems,* pp. 128–133, 1993.
9. S. M. Maurer and B. A. Huberman, "Restart strategies and Internet congestion," in *Journal of Economic Dynamics and Control,* vol. 25, pp. 641–654, 2001.
10. A. van Moorsel and K. Wolter, "Optimization of Failure Detection Retry Times," in *Performability workshop*, Monticello, IL, Oct. 2003.
11. A. P. A. van Moorsel, K. Wolter, "Analysis and Algorithms for Restart," in Proceedings of *Quantitative Evaluation of Systems,* Twente, The Netherlands, pp. 195–204, Sep. 27–30, 2004.
12. R Development Core Team, *R: A Language and Environment for Statistical Computing,* R Foundation for Statistical Computing, Vienna, Austria, http://www.r-project.org, 2003.
13. Y. Ruan, E. Horvitz and H. Kautz, "Restart Policies with Dependence among Runs: A Dynamic Programming Approach," in *Proceedings of the Eight International Conference on Principles and Practice of Constraint Programming,* Ithaca, NY, Sept. 2002.
14. M. Schroeder and L. Buro, "Does the Restart Method Work? Preliminary Results on Efficiency Improvements for Interactions of Web-Agents," in T. Wagner and O. Rana, editors, *Proceedings of the Workshop on Infrastructure for Agents, MAS, and Scalable MAS at the Conference Autonomous Agents 2001,* Springer Verlag, Montreal, Canada, 2001.

Appendix A. On-line Determination of Restart Time

Our on-line algorithms attempt to minimize the expected time it takes to set up a TCP connection, by using restart. In terms of job completion time, as used in [11], the 'job' thus corresponds to connection set up. If $f(t)$ is the probability density function for the job completion time, then to minimize the expected job completion time E_τ, with restart every τ time units (for as long as a job does not complete), we have the following result [11]:

$$E_\tau = \frac{M(\tau)}{F(\tau)} + \frac{1 - F(\tau)}{F(\tau)}(\tau + c), \tag{1}$$

where F denotes the probability distribution for the time a job can take, that is,

$$F(\tau) = \int_0^\tau f(t)dt, \tag{2}$$

and M denotes the first partial moment of download times:

$$M(\tau) = \int_0^\tau tf(t)dt. \tag{3}$$

Finally, c denotes the time it takes to execute a restart.

In our scalable on-line algorithm, we assume we collect data for a system with some restart time τ set beforehand, out of our control. Based on the collected data, we adapt τ to improve the expectation of the completion time. Of course, if one continues collecting data, the amount of data eventually gets prohibitively large. We therefore keep track of results per 'bucket,' that is, we divide the observations over H buckets, each of size $h = \tau/H$, and only keep track of the average return time M_i and number of samples N_i within each interval $(i = 1, 2, \ldots, H)$. In the i-th bucket, we thus consider the observations with values in the interval $[(i-1) \cdot h, i \cdot h)$. If we label the observations $t_{i,1} \ldots t_{i,N_i}$, M_i is estimated by:

$$\hat{M}_i = \frac{1}{N_i} \sum_{j=1}^{N_i} t_{i,j}. \tag{4}$$

We also keep count of N_τ, the total number of observations that take at least τ time units. (Note that for these observations a restarts is initiated.) For candidate retry time $\tau_i = i \cdot h$, we then obtain the following estimators for (2) and (3):

$$\hat{F}(\tau_i) = \frac{\sum_{k=1}^{i} N_k}{\sum_{k=1}^{H} N_k + N_\tau}, \tag{5}$$

$$\hat{M}(\tau_i) = \frac{\sum_{k=1}^{i} N_k \cdot \hat{M}_k}{\sum_{k=1}^{i} N_k}. \tag{6}$$

Ultimately, we thus estimate the expected diagnosis time E_{τ_i} by the asymptotically unbiased ratio estimator [5]

$$\hat{E}_{\tau_i} = \frac{\hat{M}(\tau_i)}{\hat{F}(\tau_i)} + \frac{1 - \hat{F}(\tau_i)}{\hat{F}(\tau_i)} \cdot (\tau_i + c). \tag{7}$$

The optimal restart time is then obtained by selecting the value of $\tau_i = i \cdot h, i = 1, 2, \ldots, H$, which minimizes (7).

Appendix B. Used Statistics

Since our data did not suggest noteworthy trends or seasonal indications (with exception of the clustering effect discussed in Section 3), we may assume we deal with (weakly) stationary time series, see Paragraph 1.4 of [3]. Stationarity implies that for a time series $\{X_i, i \in \aleph\}$, the second moment of all X_i is finite, the mean value identical for all X_i, and the correlation between X_i and X_{i+k} is independent of i (see Definition 1.3.2 in [3] for a mathematically precise definition.)

The figures in this paper plot the 'ACF' or auto-correlation function, as well as the 'PACF' or partial auto-correlation function. These can directly be obtained using the statistical package 'R' [12], using standard sample-based estimators corresponding to the following definitions of (partial) autocorrelation. Auto-correlation $Corr(X_n, X_{n+k})$ of lag k of a stationary time series $\{X_i, i \in \aleph\}$ is defined as:

$$Corr(k) = \frac{Cov(X_n, X_{n+k})}{Cov(X_n, X_n)} = \frac{E[X_n X_{n+k}] - E[X_n]E[X_{n+k}]}{Var[X_n]}, \tag{8}$$

where, as usual, E is used to denote the expectation of a random variable, while Var denotes variance. Partial autocorrelation can be said to adjust the autocorrelation with lag k for the intervening observations (that is, for those with lesser lag). Compared to the ACF, the PACF indicates if auto-correlation can be explained by the information in the intermediate variables. For the precise definition of PACF, we refer to Definition 3.4.1 of [3]. For consecutive random variables in a time series to be independent, the autocorrelation function must be zero. For the sample-based computation, this translates into the autocorrelation function to be within a confidence interval around zero, as depicted in the various graphs in this paper.

Service-Level Management of Adaptive Distributed Network Applications

K. Ravindran[1] and Xiliang Liu[2]

[1] City College of CUNY and Graduate Center, Department of Computer Science, Convent Avenue at 138th Street, New York, NY 10031, USA
ravi@cs.ccny.cuny.edu
[2] CUNY Graduate Center, Computer Science, 365 Fifth Avenue, New York, NY 10016, USA
xliu@gc.cuny.edu

Abstract. The paper is on generic service-level management tools that enable the reconfiguration of a distributed network application whenever there are resource-level changes or failures in the underlying network sub-systems. A network service is provided to client applications through a protocol module, with the latter exercising network infrastructure resources in a manner to meet the client requirements. Client requests for a network service instantiate the underlying protocol module with parameters specified at the service interface level, along with a prescription of critical properties to be enforced therein. At run-time, a management module may automatically monitor the service compliance to client-prescribed requirements, and notify the client whenever a service quality degradation is detected. The paper proposes a 'function'-based model of service provisioning to realize our management approach. In this model, a service prescription conforms to generic interface templates based on an enumeration of the service attributes visible to clients, and how these attributes logically relate to one another in composing a client-level quality expectation. Our management model is independent of the specifics of problem-domain, which simplifies the development of distributed adaptive applications through a 'software reuse' of the management module. The paper presents the case study of an application: CDN, to demonstrate the usefulness of our model.

1 Introduction

Network-centric distributed applications are evolving in the form of requiring far diverse and widely varying service capabilities from the network infrastructure (such as electronic banking, tele-medicine, and tele-shopping). Network infrastructures are also evolving to augment their capabilities in terms of diverse criteria and to offer these capabilities in a form usable by applications [1]. To balance application evolutions on one hand and infrastructure evolutions on the other without constraining either, comprehensive network management tools and paradigms are required that can allow applications to interwork with network

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 101–117, 2005.
© Springer-Verlag Berlin Heidelberg 2005

infrastructures in a flexible and extensible manner. A part of these challenges arises due to infrastructure outages and/or changes that may occur dynamically during an application execution (e.g., increase in the access latency on a web page due to the crash of a 'content distribution' agent node). Our paper identifies a suitable *network service model* to support the development of applications that can adapt to the capabilities offered by network infrastructures.

The current management standards, TINA and DCOM [2, 3], advocate the partitioning of a distributed network application into two types of entities: *service provider* (SP) and *service user* (or client). The SP maintains a repertoire of protocol mechanisms that are capable of offering network services to clients. For service delivery, a protocol mechanism exercises the infrastructure *resource* components placed at different sites of the network. A client application controls the extent to which network infrastructure resources are exercised, based on its service-level requirements. Our work on network service models purports to 'liaison' these two aspects of network application development. See Figure 1.

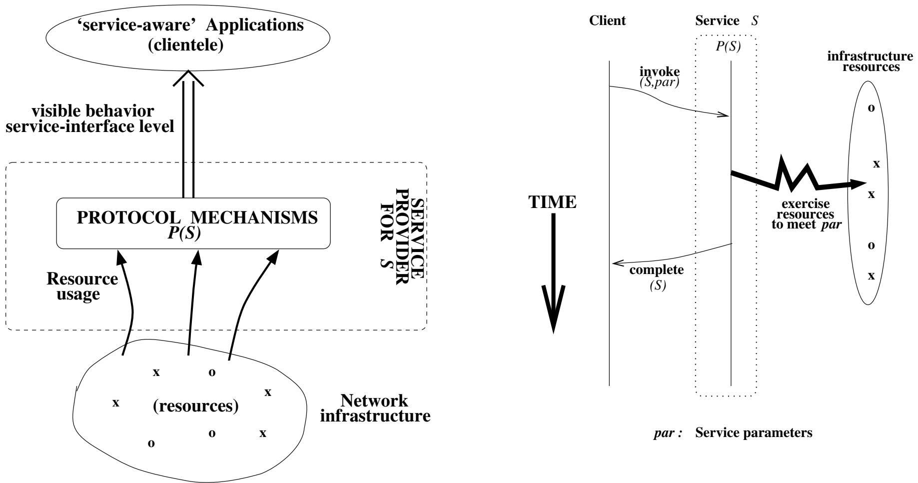


Fig. 1. Layers of functions in a service provisioning system

A protocol $P(S)$ exercises the infrastructure resources in delivering a service S to clientele. The internal state maintained by $P(S)$, which abstracts the resource usage, may be mapped onto service-level parameters visible to clients. A client may determine the feasible service behavior therefrom, to compare with the service obligations expected of S. Any deviation in the actual service behavior from the expectations is symptomatic of resource-level failures (e.g., excessive packet loss seen by end-users indicates a heavy bandwidth congestion along the network path). Thus, *behavioral monitoring* of network services is a necessary element of client-level reconfiguration mechanisms.

From a service-level programming standpoint, the monitoring activity may be structured as generic functions that can be instantiated with problem-specific

parameters. For instance, how often the state variables characterizing a service behavior are sampled is application-dependent, while the distributed algorithm to sample the state at various network nodes can itself be generic. Our model of network services allows developing the monitor and control functionalities in a *service-neutral* manner, making them usable across a wide range of applications. The service-neutrality of our model arises from its 'functional' orientation, where a service prescription conforms to a generic template based on enumeration of service attributes and how they logically relate to one another. A management module interworks with the service and client modules through a generic schema, for monitoring service compliance to client-prescribed critical properties. The monitor is realizable by a set of software agents that can be implanted in the target client and service modules, and be projected onto the problem-specific parameter space to facilitate their interworking with the target modules. A re-use of the management agents across different applications may significantly simplify the development of distributed networking software.

In the paper, service-level critical properties may be prescribed in the form of logical relations on the service attributes exported through a client-visible interface. Clients may instantiate the attributes of a network service with parameters, along with a prescription of what service property should hold. This prescription in turn allows the monitoring of service compliance to client-level requirements by the management module at run-time. Dynamic reconfiguration of clients may be triggered by notifications of service quality degradations from the management module. The paper describes case studies of network applications to bring out the usefulness of our management model.

The paper is organized as follows. Section 2 provides a management-oriented view of network service offerings. Section 3 describes our application-oriented function-based model of network management. Section 4 positions our model in the light of existing management paradigms. Section 5 provides the case study of a network application: 'content distribution network' (CDN), in terms of our model. Section 6 concludes the paper.

2 Management-Oriented View of Service Offerings

The functions of a network sub-system may be made available to client applications through an abstract service interface that prescribes a set of well-defined service features and capabilities, i.e., *service attributes*. A client may exercise one or more features to obtain a desired quality of service delivery. For example, the 'accuracy level' of time information is a feature of Network Time Service (NTS) exercisable by a client-prescribed 'minimum accuracy' parameter — say, to time-stamp banking transactions. The protocol employed at network infrastructure level to provide a given service (e.g., 'sync' message exchanges between NTS nodes) is itself hidden from clients.

2.1 Service Behaviors Visible to Clients

How a service 'quality' as seen by clients is affected by the changes in infrastructure resources constitutes a service behavior. Given a client-prescribed quality expected of a service S, different protocol modules $P(S), P'(S), \cdots$ may exhibit distinct behavioral profiles, i.e., offer different levels of service S for a given amount of infrastructure resources.

From a client's perspective, two services are identical if their externally visible behaviors for a given a set of parameters are the same. As example, consider a 'circuit switched' service offered in POTS-style telephone switching networks [5]. A management-oriented view of 'circuit switched' service is one of a 'network link' that offers transfer delay with variance ≈ 0 around a mean d for each data unit (where $d = \frac{\text{size of data unit}}{\text{link bandwidth}}$). This view, while subsuming traditional POTS-style implementations that dedicates a 'copper wire' between end-points to realize voice 'links' (with implicit guarantees of bandwidth), also encompasses 'packet switching' implementations that exhibit a similar delay behavior — at least over an operating range of interest. In general, service differentiation may be in terms of measurable parameters of externally visible behaviors.

Changes in the parameters of a service offering depict macroscopic behavior indices (or symptoms) at the service interface, reflected onto from infrastructure changes. For example, a symptom of network congestion is the end-to-end data transfer delay over the network exceeding a limit [6, 7]. In general, a service behavior can be dynamic in nature, i.e., service features can change in the midst of service provisioning due to changes in the underlying infrastructure elements.

2.2 Programmability of Service Offerings

The service attributes exported by the SP may pertain to performance, availability, functionality, and the like. From a client standpoint, the quality expectation may be some combination of these attributes. With such service-aware clients, any degradation in the level of service offering should be within the tolerance limits of attributes prescribed by clients.

The client invokes a network service S by prescribing a set of parameters. The SP instantiates a network protocol $P(S)$ by mapping the client prescriptions to protocol-level parameters [8]. The latter allow exercising the infrastructure resources by the functions that compose of $P(S)$. In general, a higher level of service obligation to clients requires an increased usage of resources; likewise, a reduced resource availability lowers the service quality. The parameterizable execution of $P(S)$ allows a macroscopic control of infrastructure resource usage to meet the client-prescribed service parameters. In the NTS example, the client-specified resolution and accuracy levels of time information influence the frequency of 'time sync' message exchanges between NTS nodes: higher accuracy requiring more frequent messages. The relationship between resource usage and service quality is embodied into the functions realizing $P(S)$.

Dynamically occurring infrastructure resource outages and/or failures (such as component upgrades or removals) may manifest as observable behavioral changes in a service offering. Based on a quantitative assessment of these changes,

an *adaptive* client application may adjust its service parameters to match the available network resources: relax service expectations when resources are limited, and tighten them otherwise. In some cases, even with sufficient resources, a client may wish to relax its service expectations — for reasons such as usage-sensitive service tariffs.

Guaranteeing a minimal service obligation to clientele against severe failure conditions in the infrastructure depicts 'service availability'. If availability takes precedence over performance, the SP may employ an operationally safe protocol despite it being heavy-weight in terms of resource usage. Such a protocol incurs a bounded recovery time when failures occur. On the other hand, if performance is more important, a light-weight protocol may be employed — though it is unsafe. For example, a 'nack'-based protocol may be employed for a 'end-to-end data transfer' service when packet loss in the network is low, and an 'ack'-based protocol otherwise. The latter guarantees a correct 'data transfer' between users but at a lower transfer rate. In general, the client-level prescriptions of a service should capture the tradeoffs between availability and performance.

2.3 Time-Scales of Service Monitoring

One may consider the possibility of a client detecting service degradations through symptoms in the problem-domain that may manifest therefrom. Due to slower dynamics of the problem-specific state analyzed by clients, the latency in such a detection of symptoms may be higher. On the other hand, an automated detection of the symptoms of network service degradation involves simply comparing the client-prescriptions of expected service behaviors and the parameters of actual service offering from the network infrastructure. Such a detection can be realized over a much faster time-scale. Figure 2 shows our experimental studies on the relative time-scales of congestion detection at various layers of a video distribution network. In these studies, the human-perceived quality degradation at the video receivers is on a slower time-scale, when compared to the receiver agent-level detection based on the fluctuations in observed frame loss rates. Accordingly, a recovery triggered by user-level notifications (say, through a GUI on receiver windows) incurs higher latency, and is less responsive to network congestions. In general, program-level symptoms can be detected much sooner than the corresponding problem-domain anamolies.

A service degradation can be detected by identifying specific patterns of state changes in a computational projection of the problem-domain. Accordingly, adaptive service provisioning can be realized by distributed algorithms that interpret the monitored program-level state variables at computational time-scales (such as how frequent the state variables are sampled). This allows recovery from, say, an infrastructure resource failure in a timely manner.

2.4 Management Module for Reconfigurable Services (RSMM)

In a basic form, the RSMM maps client-initiated service requests onto a specific protocol at network infrastructure level. For this purpose, the RSMM maintains

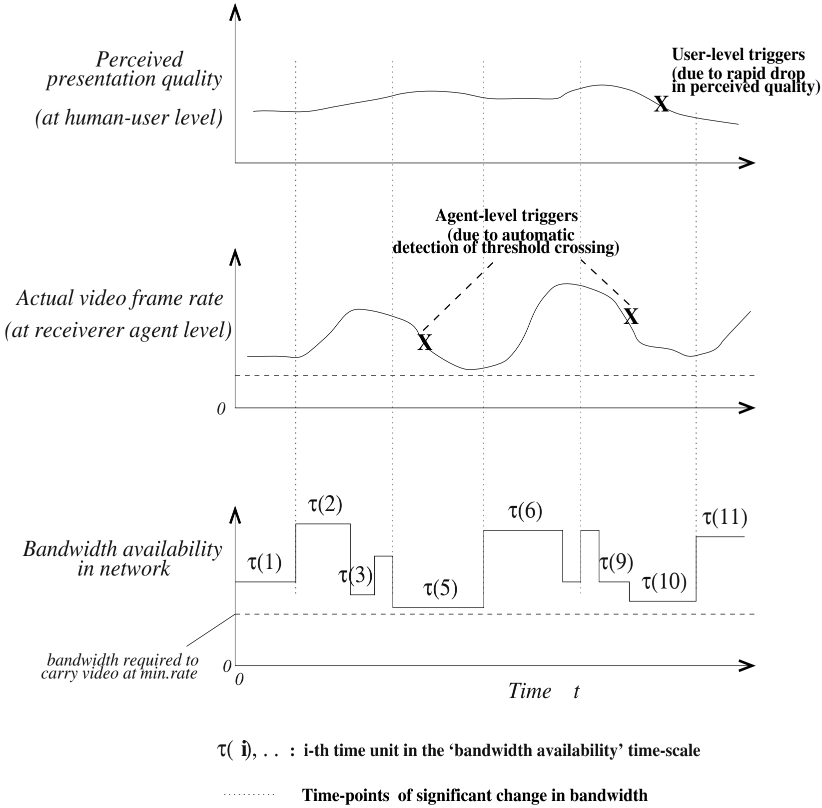


Fig. 2. Time-scales of parameters at various layers of video distribution network

the binding information for various services provided by the network infrastructure. The focus of our research is on additional functionalities desired of the RSMM: service monitoring and coordination of client adaptation, to support dynamic settings where changes and/or outages in infrastructure resources may occur at various points in time[1]. See Figure 3. The RSMM functionalities should be independent of the problem-domain a service offering may pertain to.

If the RSMM does not provide for service monitoring, detection of service degradation by clients (and a subsequent adaptation) is possible only over problem-specific time-scales. On the other hand, RSMM support for service monitoring enables client adaptations to occur at computational time-scales. These cases are illustrated by scenario-A and scenario-B respectively in Figure 3.

Suppose the client actions to deal with service degradations are expressible in a closed-form involving application-supplied functions. Here, the RSMM agents

[1] In comparison with isolating the network subsystems based on observed outages in aggregated network elements [9], our work focuses on application-level adaptations to deal with such outages.

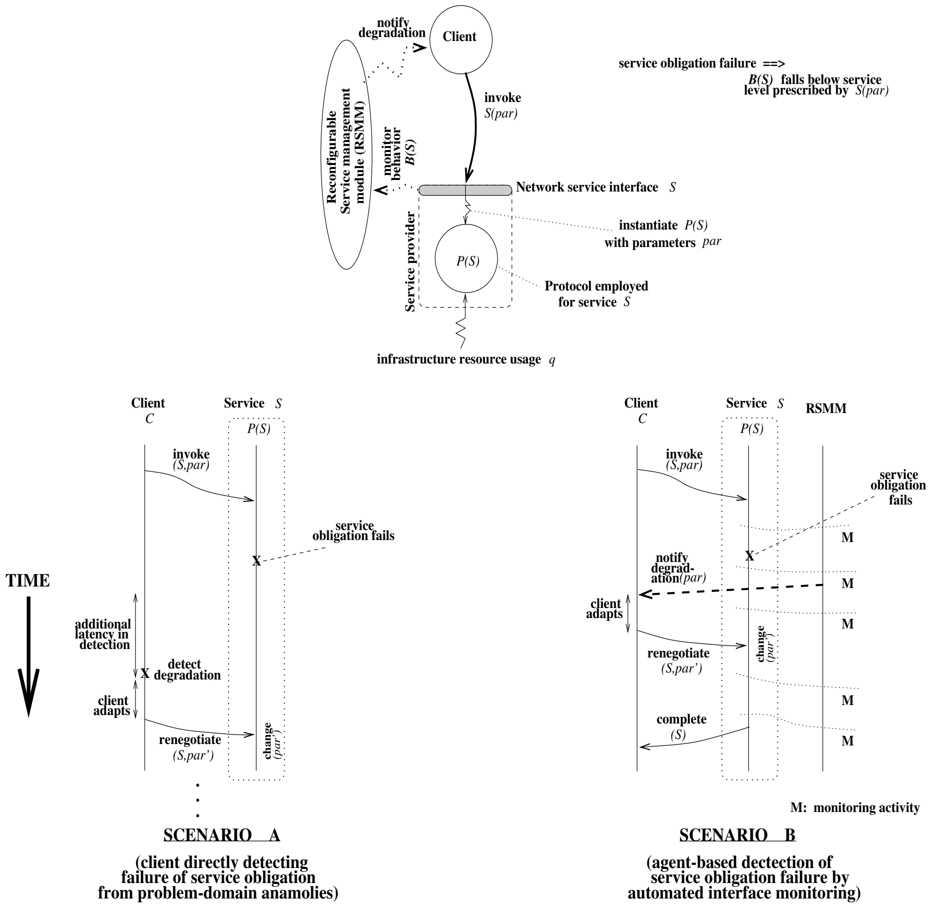


Fig. 3. A high level view of reconfigurable network service offering

at the client and service sites can coordinate with one another to reconfigure the application over computational time scales (e.g., reducing the video send rate in a 'multiplicative decrease' form during network congestion). Otherwise, the RSMM may signal an exception that triggers client-level recovery over problem-specific time scales (e.g., human intervention to reduce the video window size and/or switch to a low quality encoding [6]).

For monitoring service-level compliance checks with respect to client expectations, the RSMM instantiates service-level meta-data dissemination protocols with parameters pertaining to the problem-domain. The monitoring and control roles of RSMM should themselves be independent of the service-specifics (at meta-level), so that they can be instantiated across diverse problem-domains[2].

[2] The service description language should allow capturing the 'flow of time' — so that 'time scale' can be expressed as a parameter [10].

2.5 Management View of Sample Network Application: CDN

We delineate the management aspects of network service provisioning from the details of how the underlying protocols function in providing the service. In this light, we examine a 'content distribution network' (CDN).

A server hosts the content pages of an information base — e.g., an electronic shopping catalogue. With a large number of clients trying to access various parts of the information (or, pages), the server maintains copies of pages at multiple nodes of the distribution network (such as in AKAMAI) [11]. When a client (say, a web browser) accesses the CDN server for a page, the request is forwarded to the nearest node containing this page for downloading. Keeping copies of the pages at multiple nodes in the network (i.e., replica nodes or proxies) increases access performance and offers higher availability of the information base. The attributes prescribed by CDN clients may include the tolerances to page access latency and miss rate on page delivery deadlines.

There are two elements of the protocol mechanisms: i) placement of replicas in the network, and ii) update policies for keeping the replicas consistent[3]. See Figure 4 for illustration. The geographic spread-out of replicas relative to the location of clients in the network and the page update policies to keep the copies consistent determine the latency in fetching pages by a client. The policies for replica placement and page updates are chosen by the CDN protocol[4].

The protocol mechanisms (i) and (ii) constitute the 'resources' in terms of our service management model. The agents placed at client nodes continuously monitor the access latency, so that the client nodes can adapt their access behavior — such as removing a page from their access set if it incurs excessive latency and/or stipulating tighter controls on (i) and (ii).

As can be seen, a management view delineates the protocol mechanisms employed in the problem-domain and the generic monitoring and control tools that interwork with these mechanisms. In terms of this view, we now describe our 'function'-based service model and the underlying management techniques.

3 Our 'Function'-Based Model of Services

We employ an *application-oriented functional* approach to managing network services. It underscores the theme: reconfigurability of network services.

---

[3] Page update policies include correcting the copy at a node when a client attempts to access this copy (client-driven updates) and correcting the copies at various nodes whenever the server changes its master copy (server-driven updates).

[4] In the management view, a CDN client need not be an end-user of content pages. The client may possibly represent an access box feeding pages to a community of customers in a geographic area. In the latter case, the access box may derive its 'access latency' prescription to the SP from an aggregation analysis of the individual customer needs. The QOS composition mechanisms employed by the access box (say, for customer pricing) are however outside the purview of a management model.

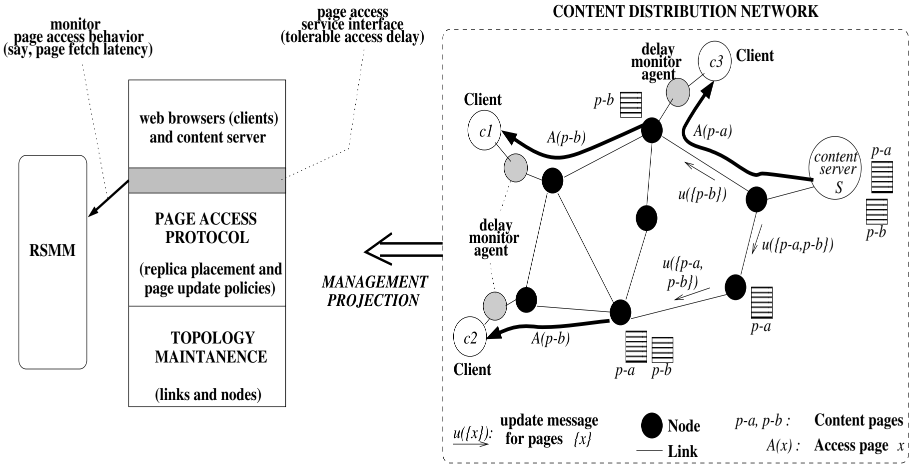


Fig. 4. Management view of a content distribution network

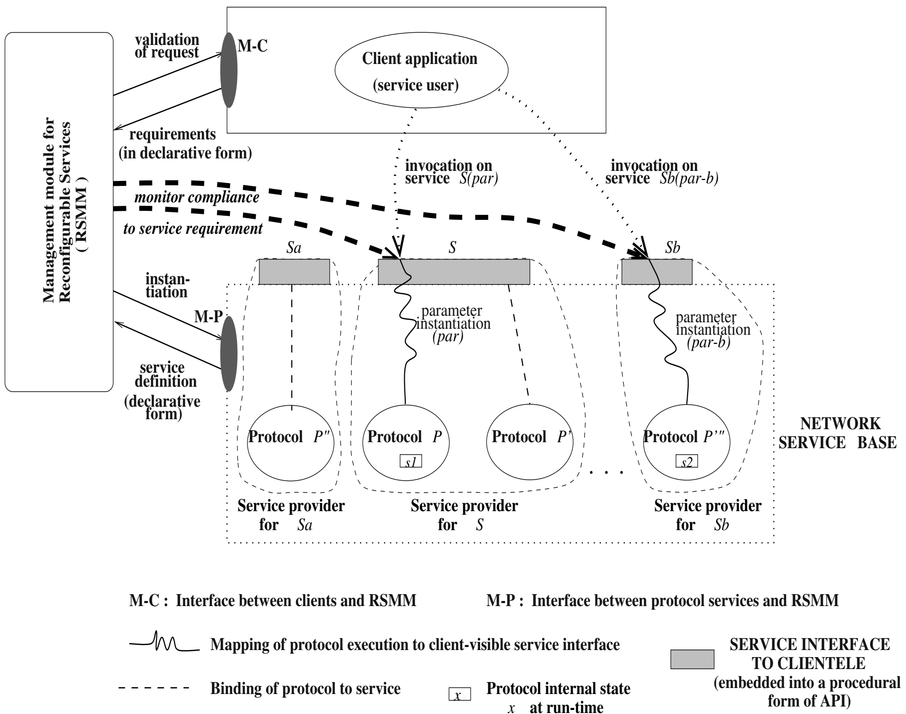


Fig. 5. Architectural view of our service management system

3.1 Service Normalization

Figure 5 shows an architectural view of how different types of services can be supported through a general-purpose service management system.

First, when different protocol modules $P(S), P'(S), \cdots$ are capable of providing a given network service S, the externally visible behaviors of $P(S), P'(S), \cdots$ are represented through a set of features common to S. This ensures that the choice of a protocol module by the SP to handle a given client request on S and the use of service-specific functions to manage S are transparent to the RSMM. At the client-service interface level, the differences in internal mechanisms employed by $P(S), P'(S), \cdots$ are not visible to the clients. Second, with a multitude of network service offerings $S_1, S_2, \cdots$ to be managed by the RSMM, it is desirable that the behavior descriptions of $S_1, S_2, \cdots$ be captured through a uniform meta-language. This allows the RSMM to employ the same set of management tools across various types of network services, by simply instantiating the tools with service-specific parameters.

The above goals require that service behaviors are representable in a canonical form, in terms of the notations allowed by a generic service description language. Towards this end, the protocols $P(S), P'(S), \cdots$ expose a common *interface state* $\Psi(S)$ to clients through the SP, denoted as:

$$\Psi(S) = \{\alpha_i\}_{i=1,\cdots,k \text{ for } k \geq 1},$$

where $\alpha_1, \cdots, \alpha_k$ are state variables. From a client's perspective, the α_i's are the service attributes exported by S that can be compared against 'values' supplied by the client to determine if S meets the behavioral expectations.

3.2 Protocol-to-Service Mapping

A protocol $P(S)$ realizing the service S maintains a state s, which is a reflection of the infrastructure resource usage by the internal mechanisms of $P(S)$. At the service level, the interface state $\Psi(S)$ is a mapping of the form:

$$\Psi(S) = \gamma_{(P,S)}(s)$$

prescribed by the SP. Consider the example of a 'virtual circuit' service implemented by an ack-based window protocol over a network, referred to as WIN(VIRT). A range of achievable data transfer rates over the 'virtual circuit' can be prescribed by a client (i.e., data sender/receiver). Infrastructure resources are the send/receive window w, and the link capacity c and error probability e. A client-visible interface state is the link utilization, given as:

$$\Psi(\text{VIRT}) = \gamma_{(\text{WIN,VIRT})}(\{w, c, e\}),$$

where $\gamma_{(\text{WIN,VIRT})}$ may be an analytical formula for computing the link utilization α_1 in terms of the infrastructure parameters w, c and e (among others)[5]

[5] The achievable link utilization is given by: $\alpha_1 = \frac{1}{1+\frac{1}{w}+\frac{(\bar{l}-1).\text{T}.R}{w.C}}$, where $\bar{l}$ is the mean number of transmissions per packet (expressed in terms of e, the timeout period T for retransmissions, and the packet size R).

[12]. The $\gamma_{(\mathrm{WIN,VIRT})}$ is encapsulated into an 'applicative' function supplied by the 'circuit' provider. In general, $\gamma_{(P,S)}$ depicts a static mapping relation between the protocol internal state s and the client-visible interface state $\Psi(S)$.

In legacy systems however, an explicit representation of the protocol internal state s may not be feasible (e.g., 'available bandwidth' on a TCP connection). In such cases, the SP may declare one or more of the service attributes as *unspecified*, i.e., these attributes can neither be assigned 'values' by clients nor be exported as 'indicators' of network conditions by the SP. Here, the $\gamma_{(P,S)}$ is simply a stub library to implement the procedural hooks to the service module.

Given our state-oriented interface characterizations, a language like JAVA is more suitable for service-independent interface descriptions (in comparison to data-oriented languages like XML [13]).

3.3 Service-Level Behavioral Descriptions

The requirement of a generic behavioral interface can be met with a 'function'-based model of service provisioning that involves the RSMM and the client and service modules. We use modular decomposition principles to structure these functions and the flow of meta-information between them. Refer to Figure 5.

The externally visible behavior of S can be described as a set of generic logical relations on the interface state components $\alpha_1, \cdots, \alpha_k$:

$$L(\Psi(S)) \equiv \Phi(\{o_1(\alpha_1), \ \cdots, \ o_k(\alpha_k)\})$$

where $o_1, \cdots, o_k$ are 'applicative functions' that produce boolean results based on the values of $\alpha_1, \cdots, \alpha_k$ respectively and $\Phi(\cdots)$ depicts logical relations: AND, OR, and NOR. Such a behavior description allows the RSMM tools to be independent of the problem-domain S pertains to.

Consider the example of a video distribution service (VDIST). The sustainable frame rate and the inter-frame delay are the interface state components. $L(\Psi(\mathrm{VDIST}))$ may capture a condition: 'frame rate is no less than *vrate*' and 'frame-delay is no more than *vdel*'. This condition may be represented as:

$$L(\Psi(\mathrm{VDIST}) \equiv [> (\mathrm{FRATE}, \ vrate) \wedge \ < (\mathrm{FDEL}, \ vdel)],$$

with '>' and '<' being 'applicative' functions to compare with the values *vrate* and *vdel* respectively, as prescribed by the client (for use by the RSMM); FRATE and FDEL are the names of service features that allow client-level handle on the frame rate and frame delay respectively.

As can be seen, the service interface state may be derived through static mappings of protocol internal states, whereupon service behaviors may be determined by exercising logical relations among the interface state components. The extraction of service behaviors from the interface state forms a core part of the monitoring functionality of RSMM.

Largely, our RSMM acts as a *broker*, liaisoning clients with services, based on client-specified requirements and published service capabilities. The actual

provisioning of a requested service is itself done directly by the SP. This is different from the 'contract-bidding' based management model suggested in [18] which requires service-specific procedures to be rigidly integrated into the model.

4 Related Paradigms for Managing Distributed Networks

We project our service management function in the light of currently prevalent management approaches: i) resource-level monitoring, and ii) 'SNMP'-based functional management. We focus the comparison on the programming model of service management, i.e., how a service management can be built into a programming environment for constructing distributed network subsystems.

4.1 Resource-Level Monitoring

The ReMoS system [14] provides a query-based interface to obtain information about resource availability in a form meaningful for the SP to support application-level information flows. Likewise, the QOS Broker System (QBS) [15] provides for QOS specification, resource reservation, and resource monitoring. The QBS allows prescribing QOS violations that can be detected by 'sensor' objects (e.g., video frame delay jitter threshold of .15 *msec*). The scope of our work, when cast in the above lights, is in the SP-level mapping of flow specs onto network resource parameters.

The Rutgers Environment Aware API [16] provides for application adaptation through asynchronous event delivery, where an event is prescribed at an appropriate abstraction level along with a handler for that event type. Though the Rutgers API caters to a variety of network applications, its usefulness has been studied mainly to provide adaptation in relation to network-centered parameters (such as link bandwidth and security). In contrast, our model unifies the management of both information networks and data networks through a canonical and meta-level service-oriented interface. We extend the notion of infrastructure resources to cover beyond the traditional network parameters.

The RSMM part of SP may employ, say, the ReMoS primitives or the Rutgers API to monitor the symptoms of resource failures over selected time-scales. For instance, the ReMoS procedures for statistical estimations can be employed by the RSMM for smoothing and/or filtering of monitored parameters. Likewise, the QOS specs from applications can be signaled to the SP using the ReMoS primitives. The RSMM may also adopt the QBS methods to prescribe when an increase or a decrease in resource usage should take place. In contrast with such QOS specifications and mapping, the control actions of SP for QOS management are problem-specific (e.g., determining the proxy placement in CDNs). So, the SP consists more of 'resource adaptation' functionalities mediated through the RSMM. Our service-level management model reflects this emphasis.

4.2 SNMP-Based Network Monitoring Approach

SNMP defines a set of APIs that can be invoked at a network management station, and also the underlying (signaling) message exchanges between the management station and the agents executing at target protocol sites [17]. A 'sniffer' tool is provided that examines all packets exchanged through the protocol being monitored, without disrupting their flow. One useful SNMP feature is a prescription of packet filters based on 'logical expressions' that select which packets are captured for further analysis (e.g., all packets from an IP host to a specific 'ethernet interface' address). An event table prescribes when a notification or an alarm, is to be sent to the management station (e.g., when the number of packets destined to a given host exceeding a limit).

Packet selection criteria and the functions necessary to prescribe events therefrom are oriented towards traffic and fault analysis at various network elements. For example, the number of ICMP packet losses suffered at a router node exceeding a limit may be treated as indicative of a crash of this node. Arbitrary packet selection criteria can be loaded into 'filter tables' maintained by the network management system, along with a provision of basic functions to operate on the filtered streams of packets (e.g., computing the probability of packet loss). In contrast, our approach is towards behavior analysis of network subsystems as seen by applications through service interfaces.

In SNMP, the 'network management' user basically selects a function to operate on a packet stream from a menu of functions (possibly, through a GUI) hardwired into the management system. From the perspective of service programming, the presence of human user in the monitoring-reconfiguration loop makes pushes the SNMP model to a lower level, in comparison to our approach.

We now provide, as a case study, a management-oriented description of how service-level monitoring and control can be realized in CDNs — c.f. section 2.5.

5 Case Study of Reconfiguration Management in CDNs

We first capture the service-level behavior, i.e., changes in page access latency with respect to the degree of page replication — as shown in Figure 6. We then describe the meta-activities of RSMM to adjust the page replication at run-time.

The page access latency PLAT, which is a client-visible attribute, is influenced by the replication mechanism that allows the nearest and most-recent copy of a page in the network to be accessed. The protocol-level mechanism depicts the exercising of infrastructure resources, namely, 'sync' message exchanges and traversal of data over network links.

The internal state of a protocol that realizes the CDN service is the current placement of replicas $repset_{cur}$ and the link delays between nodes $ldel_{cur}$. The SP may provide a mapping function:

$$plat(i) = \gamma_{(U,\mathrm{CDN})}(repset_{cur}, ldel_{cur}),$$

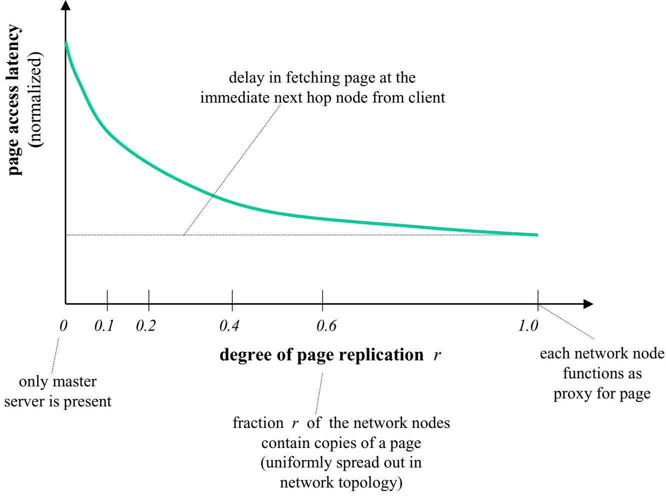


Fig. 6. Quantitative behavior-oriented study of CDN service

to yield the value of PLAT in an ith sampling interval. The function may be specific to an update policy U on pages employed by the CDN protocol (i.e., client-driven or server-driven or page lifetime-based).

We consider four types of operations provided by the CDN SP for use by the RSMM to control the replica placement:

$$
\begin{aligned}
&plat(obs) = \text{smooth_latency_samples}(\{plat(i)\}_{i=1,2,\cdots});\\
&plat_{new} = \text{set_latency}(plat'_c, plat(obs));\\
&repset_{new} = \text{get_replica_placement}(plat(obs), plat_{new});\\
&\text{change_replica_placement}(repset_{new});
\end{aligned}
$$

where $\{plat(1), plat(2), \cdots\}$ are the time-chronological sequence of samples of PLAT from which a smoothed observation $plat(obs)$ can be extracted and $repset_{new}$ depicts a new placement of replicas on the CDN nodes to bound PLAT to less than $plat_{new}$. The 'smooth_latency_samples' operation may simply be an averaging mechanism over a time-scale meaningful to the 'content distribution' application. The condition $> (plat(obs), plat_c)$, i.e., the monitored 'value' $plat_{obs}$ exceeding $plat_c$, may cause the RSMM to trigger a change in replica placements, where $plat_c$ is the currently prescribed value for PLAT.

The 'set_latency' operation may compute $plat_{new}$ as, say, $[plat'_c - \Delta \times (plat(obs) - plat'_c))]$, where $plat'_c$ is a modified limit on latency prescribed by the client and Δ is a parameter used by the SP for 'control-theoretic' stability of the changes in latency. The 'get_replica_placement' function may employ the $\gamma_{(U,\text{CDN})}$ function in determining $repset_{new}$ that will cause the latency to not exceed $plat_{new}$. The 'change_replica_placement' function is specific to the update policy employed

by the CDN protocol. Suppose a server-driven update policy is employed. If $repset_{new} \supset repset_{cur}$, the replicas $(repset_{new} - repset_{cur})$ are initialized to the current page content, and then installed at the designated CDN nodes. If $repset_{new} \subset repset_{cur}$, the replicas $(repset_{cur} - repset_{new})$ are removed from the concerned CDN nodes. Other cases can be a combination of these mechanisms.

The operations listed for a CDN service are supplied by the SP as 'applicative' functions that can be invoked by the RSMM. The client prescribes the monitoring conditions through 'applicative' functions: the '>' relation on latency values, for use by the RSMM[6].

6 Conclusions

When developing distributed network services, challenges arise due to sub-system level failures and/or changes that may dynamically occur and can, in turn, affect the application functionality. How such sub-system level events impact applications is specific to the problem-domain. In this paper, we described an automated and generic management tool that enables the client application to reconfigure whenever there is a service change or failure.

To support the integration of service management tools into distributed network development environments, we employed a paradigm founded on modular decomposition principles. In this paradigm, a service may be provided by a protocol module through a generic interface, with the client application instantiating this module with a desired set of parameters. The service-level management module (RSMM) maintains the binding information for various network services that can be provided through the infrastructure resource usage. The RSMM supports a highly dynamic setting, such as changes and/or outages in service provisioning occurring at various points in time and in different time-scales. The RSMM effectively 'brokers' between clients and services to coordinate their interactions in a flexible and configurable manner.

Our paper described a new model of service provisioning: a 'function'-based decomposition of service components. The model separates a service interface to clients from the details of protocol modules that actually provide the service. Critical properties may be associated with a service interface, prescribable in the form of logical relations on service attributes. Client invocations on a service may instantiate these attributes with a set of parameters, along with a prescription of what service property should hold. The model allows dynamic reconfiguration from one set of parameters to another, in case a service property violation occurs at run-time. For this purpose, the RSMM incorporates a monitoring functionality that checks for compliance to prescribed critical properties at run-time. The paper also described the case study of a network application: CDN.

---

[6] [19] provides architectural frameworks to encapsulate 'policy' functions in systems for managing service-level infrastructures.

The 'function'-based structuring and the service-neutrality that underscore our service model offer the flexibility of service growths and the extensibility of supporting diverse client requirements. This is much broader in scope, when compared to the existing management models based on SNMP and OSI-TMN. Our model can be implemented in an appropriate object-oriented programming language that offers the capability of dynamically dispatchable protocol code (such as JAVA) [20]. It can thus facilitate the rapid development and deployment of network services.

References

1. S. Erfani, V. B. Lawrence, and M. Malek. **The Management Paradigm Shift: Challenges from Element Management to Service Management**. In *Applications, Platforms, and Services*, Bell Labs Technical Journal, vol.5, no.3, pp.3-20, July-Sept. 2000.
2. M. Subramanian. **Telecommunications Management Network**. Chap. 11, *Network Management: Principles and Practice*, Addison-Wesley Publ. Co., 2000.
3. M. Horstmann and M. Kirtland. **DCOM Architecture**. MSDN Library, *http://www.msdn.microsoft.com/library*, July 1997.
4. Object Management Group. **The Common Object Request Broker: Architecture and Specification**. Rev.2.0, OMG, Framingham (MA), 1995.
5. A. S. Tanenbaum. In **Switching: The Telephone System**. Chap. 2., *Computer Networks*, Prentice-Hall Publ. Co., pp.130-134, 1996.
6. C. Diot, C. Huitema, T. Turletti. **Multimedia Applications Should be Adaptive**. In Proc. *HPCS'95*, Mystic (CN), Aug. 1995.
7. K. Ravindran and R. Steinmetz. **Object-oriented Communication Structures for Multimedia Data Transport**. In *IEEE Journal on Selected Areas in Communications*, Special Issue on *Distributed Multimedia Systems and Technology*, 14(7), pp.1360-1375, Sept. 1996.
8. K. Ravindran and K. K. Ramakrishnan. **Feature-based Service Specification in Distributed Systems**. In proc. *Intl. conf. on Distributed Computing Systems*, IEEE-CS, Arlington (TX), May 1991.
9. V. Sethaput, A. Onart, and F. Travostino. **Regatta: A Framework for Automated Supervision of Network Clouds**. In Proc. *OPENARCH 2001*, Anchorage (AK), pp. 104-114, April 2001.
10. L. Yilmaz and S. H. Edwards. **Specifying and Verifying Collaborative Behavior in Component-based Systems**. In proc. *RESOLVE Workshop 2002*, Columbus (OH), June 2002.
11. J. Chase, S. Gadde and M. Rabinovich. **Web Caching and Content Distribution: a View from the Interior**. In 5th Intl. Workshop on *Web Caching and Content Delivery*, Lisboa (Portugal), 2000.
12. M. Schwartz. **Telecommunication Networks: Protocols, Modeling, and Analysis**. Addison-Wesley Publ. Co., Nov. 1988.
13. M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster. **UIML: An Appliance-Independent XML User Interface Language**. In proc. *Eighth International World Wide Web Conference*, Toronto (Canada), 1999.
14. T. Dewitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, and D. Sutherland. **ReMoS: A Resource Monitoring System for Network-Aware Applications**. In *Tech. Report*, CMU-CS-97-194, Dec. 1997.

15. M. Katchbaw, H. Lutfiyya, and M. Bauer. **Driving Resource Management with Application-level Quality of Service Specifications**. In Proc. *1st Intl. Conf. on Information and Computation Economies*, ICE98, Oct. 1998.
16. B. Badrinath, A. Fox, L. Kleinrock, G. Popek, P. Reiher, and M. Satyanaranyanan. **A Conceptual Framework for Network and Client Adaptation**. In *IEEE Mobile Networks and Applications*, 2000.
17. M. Subramanian. **SNMP Management RMON**. Chap. 8, *Network Management: Principles and Practice*, Addison-Wesley Publ. Co., 2000.
18. G. Kar and A. Keller. **An Architecture for Managing Application Services over Global Networks**. In proc. *INFOCOM'01*, IEEE-CS, Anchorage (AK), pp.1020-1027, April 2001.
19. D. Verma, M. Beigi and R. Jennings **Policy-based SLA Management in Enterprise Networks**. In *Res. Report*, IBM T. J. Watson Res. Center, 2001.
20. D. J. Wetherall, J. V. Guttag and D. L. Tennenhouse. **ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols**. In Proc. *IEEE OPENARCH'98*, San Fransisco (CA), April 1998.

A Methodology on MPLS VPN Service Management with Resilience Constraints

Jong-Tae Park and Min-Hee Kwon

School of Electronic and Electrical Engineering, Kyungpook National University
1370, Sankyuk-Dong, Buk-Gu, Taegu, Korea 702-701
{jtpark, minhi}@ee.knu.ac.kr

Abstract. Recent advent of broad bandwidth wireless and optical networks makes the resilience of information and communication service to system failures to become a critical issue. In this article, we present a methodology for MPLS VPN service management employing a resilience model. The methods can dynamically configure the service paths of MPLS VPN satisfying the TE resilience requirement from the customers. Specifically, we describe backup path design rules and derive the conditions for testing the availability of feasible backup paths satisfying the resilience constraints in a full mesh MPLS VPN. We present fast backup path construction algorithms which could make the MPLS VPN service to be available with minimal disruption, satisfying the resilience requirement from the customers. The simulation has been done to evaluate the performance and service availability in a BGP/MPLS backbone network with full mesh structure.

Keywords: MPLS VPN management, MPLS Recovery Mechanism, Resilience, Path Selection, Fault Management.

1 Introduction

Recent advent of broad bandwidth wireless and optical networks makes the resilience of information and communication service to system failures to become a critical issue. The failure of network components can be generally caused by many reasons including both hardware and software malfunction. The failure caused by hardware malfunction includes the malfunctioning of network interface card, link arithmetic errors in CPU, transmission equipments, and other I/O devices. The failure caused by software may include bugs in the operating system, networking software, and even by malicious virus or worms. In today's Giga-bit high speed network, any failure may incur lots of packet loss, and may incur non-negligible negative impact to the business.

The resilience implies the capability of recovery from these failures. In multi-protocol label switching (MPLS) networks [1], the failures at lower layers may

This work was supported by University Research Program of Ministry of Information & Communication, Korea.

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 118–133, 2005.
© Springer-Verlag Berlin Heidelberg 2005

generate hundreds of link or node failures at higher layers. It is necessary to provide a contracted reliable MPLS service to the customers with minimal or no disruption of service in case of unexpected multiple failure occurrences, resulting in high service availability. Currently, active research work including those of international standard bodies [1, 2] is going on for modeling and realizing the resilience in future high-speed network such as MPLS/GMPLS networks.

Recently, the provisioning of virtual private networks (VPN) over the global Internet is gaining much popularity. VPN provides interconnections of customer sites over a shared network infrastructure. Traditionally, VPNs have been mostly provided by the leased lines, but the development of new technology such as (MPLS) enables the service providers to look for the better cost-effective solutions in terms of scalability, security and quality of service. The provisioning of VPN over MPLS among different Autonomous Systems has been being standardized by IETF [3], and several vendors are already providing proprietary solutions such as Cisco's BGP/MPLS VPN, Nortel's MPLS-based Virtual Router, and Lucent's Virtual Router. MPLS could provide Internet service with QoS guarantee to the customers over MPLS backbone.

In BGP/MPLS VPN, the VPN routing information is distributed by MP-BGP [4] over the service provider's backbone network, and MPLS is used as an underlying network infrastructure device to forward the VPN traffic among the participating VPN sites. In BGP/MPLS VPN, the connection-less IP traffic of a VPN customer site is transparently transmitted through a provider's connection-oriented data paths, more specifically label switched paths (LSPs), of a MPLS backbone. The efficient design of these paths for BGP/MPLS VPN is an active research area [5]. RSVP-TE [6] or CR-LDP [7] is recommended for setting up these paths for BGP/MPLS VPN traffic. MPLS VPN is often configured in a full mesh, and instrumented by establishing and maintaining label switched paths (LSPs) of MPLS.

In this article, we present a methodology for MPLS VPN service management employing a resilience mode presented in the accompanying paper submitted to ISAS 2004 [8]. The methods can dynamically configure the paths of MPLS VPN (LSPs in MPLS network) satisfying the TE resilience requirement from the customers. More specifically, we propose backup path design rules and derive the conditions for testing the availability of feasible backup paths satisfying the resilience constraints in a full mesh MPLS VPN. Fast backup path construction algorithms are developed which could make the MPLS VPN service to be available with minimal disruption, satisfying the resilience requirement from the customers. The simulation has been done to evaluate the performance and service availability in a BGP/MPLS backbone network with full mesh structure.

The rapid recovery from these failures, by dynamically provisioning of fast (real-time) alternative backup paths in the underlying BGP/MPLS backbone is very important to both customers and service provider in order to avoid the disastrous service disconnection and to protect the critical business operation such as bank transaction, critical remote data access, high-level e-government service and so on. Traditionally, there have been lots of research works on fault management of networks, systems and services and various recovery mechanisms [9], including recent work on MPLS [10, 11], GMPLS [12, 13] and optical network [14, 15]. However, there have been few research works for formally defining the resilience [8], where the author presents a simple model for resilience which enables various network and service recovery mechanisms to be represented effectively. In this paper,

we apply the resilience model proposed in [8] to MPLS VPN service management to enhance the availability of the service. Basically, for a given data path associated with VPN service, we try to reserve, in advance, a set of additional backup paths such that these backup paths are intelligently utilized when some components in the primary path are not well operational due to failures.

In Section 2, we briefly introduce characteristics of MPLS VPN and resilience model for service management. In Section 3, we describe backup path design rules for MPLS VPN service and derive the conditions for testing the availability of feasible backup paths satisfying the resilience constraints. In Section 4, dynamic path management strategy for BGP/MPLS VPN service with resilience is presented. Section 5 describes the simulation results to evaluate the performance and service availability, and finally concluding remarks follows in Section 6.

2 MPLS VPN Service and Resilience Model

2.1 Basics of BGP/MPLS VPN Architecture

Figure 1 shows the basic building blocks of BGP/MPLS VPN configuration. A customer edge (CE) device, usually IP-router or a host provides customer access to the service provider network. A PE router is usually ingress or egress label switch router (LSR) of MPLS network, where VPN traffic enters or exits. Each PE router maintains VPN routing information. A provider (P) router is any router within the provider's network which is not directly connected to the CE devices. P routers are MPLS transit LSR which forward VPN data traffic between PE routers. Since the routing information for VPN traffic is maintained in PE routers, P routers are not required to maintain specific VPN routing information. The VPN connections between PE routers are maintained by populating VPN routing and forwarding (VRF) table in a BGP/MPLS VPN backbone network such that each customer connection is mapped to a specific VRF.

After receiving routing information from CE routers, a PE router exchange the routing information with other PE routers using Internal BGP (IBGP). PE routers usually establish IBGP sessions in a full mesh or hub-spoke structure. The forwarding of VPN data traffic across BGP/MPLS backbone requires an establishment and maintenance of connection-oriented data paths between PEs, called label switched paths (LSPs). LSP is the path where the data traverses through LSR across an MPLS network. IBGP sessions for a VPN service are instrumented by establishing and maintaining these LSPs of an MPLS network.

The management of these LSPs are provided by using label distribution protocol (LDP) or RSVP. It is noted that there can be several LSPs with different capabilities between PE routers for a specific VPN service. It is also noted that different VPNs may have different routes between the same PEs by different VRF tables [3]. According to [3], BGP/MPLS VPNs are usually defined by administrative policies, which are used for connectivity and QoS guarantees. BGP/MPLS VPNs are defined by customers and implemented by service providers.

In BGP/MPLS VPN, VPN data packets are forwarded to BGP destinations by simple label-switching of traffic to a BGP next-hop address. The routes from BGP node to a BGP next-hop addresses can be determined via interior gateway protocol (IGP) such as OSPF-TE and IS-IS, and label distribution protocol (LDP) allows these

routes to be associated with MPLS labels. PE routers are the only ones that need to run BGP. The different routes between a pair of PE nodes can be supported by populating VRF table in a BGP/MPLS VPN backbone network. Therefore, the backup path for a specific PE sites for a VPN can be easily realized by using MP-BGP and VRF table.

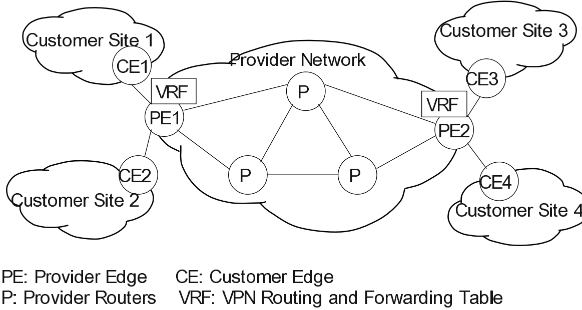


Fig. 1. BGP/MPLS VPN Configuration

2.2 The Resilience Model for MPLS Service Management

The VPN service is realized by maintaining LSPs of MPLS, i.e., MPLS path. The management of this MPLS path is thus closely related to the service availability. The resilience of an MPLS path is informally defined as a capability of providing the communication service with minimal discontinuity and rapidly restoring to the original sound situation in the case that some components of the system experience failures due to unexpected events. There are diverse protection and restoration techniques in MPLS/GMPLS network, which are currently being standardized by IETF [10, 11]. Most of them make use of the backup paths in case that the primary paths may have some problems. A primary path is defined as the working path along which the MPLS data traffic follows [8]. A backup path is defined as the path along which the data traffic follows when the primary path is unavailable due either to the failure of links or nodes [8].

The path resilience in MPLS/GMPLS network is formally defined as a real-valued function [8] such that

$$\text{Path Resilience} = \sum_{ProtectionSet} \frac{1}{m} \bullet \frac{\textit{Number of Protected Components}}{\textit{Total Number of Components}} \qquad (1)$$

Here, m is the multiplicity factor of a primary path and ProtectionSet denotes the set of all the backup paths and/or segments to protect the primary path. The multiplicity factor m of a path defines the number of total primary paths which are sharing a backup path, or a segment. Number of Components implies the total number of components in a path, without including the end-nodes of MPLS network since they are always shared among primary and backup paths. The Total Number of Protected Components implies the total number of components in the path which are protected by backup paths.

In [8], we present a simple model for resilience which enables various network and service recovery mechanisms to be represented effectively. A mapping rule to convert the customer requirement on service availability to the resilience value, and illustrating examples for various protect mode are also described in detail.

3 Design Rules and Availability Testing for Backup Paths of MPLS VPN Service

In this section, we describe backup path design rules for MPLS VPN service and derive the conditions for testing the availability of feasible backup paths satisfying the

3.1 Path Design Rules for MPLS VPN Service

We present a few backup path design rules below for fast backup path construction.

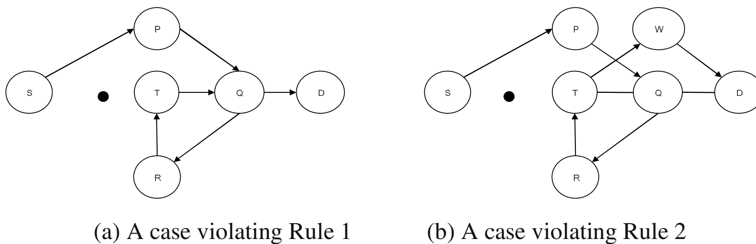
- Rule 1: No node in the path should be trespassed more than once.
- Rule 2: The backup paths for a given primary path should follow the same sequence of nodes and links of the primary path in the sub-paths shared in both the primary path and backup path.

Rule 1 implies that there should be no cycles in the path, which is a usual practice engineered in network design. Rule 2 is applied only for the design of backup paths.

In Figure 2, examples are shown to illustrate the violating cases. In Figure 2(a), the path <S, .., T, Q, D> is a primary path. We show two cases for backup path construction. With the backup path <S, P, Q, R, T, Q, D> shown in Figure 2(a), Rule 1 is violated since the node Q is trespassed twice. For the backup path <S, P, Q, R, T, W, D> shown in Figure 2(b), although Rule 1 is satisfied, Rule 2 is violated since the backup path should follow the sub-path <T, Q, D> of the primary path. In Figure 2(a), there is a cycle <Q, R, T, Q>. Before proceeding further, we need a definition, called k-protected, which is defined as follows.

Definition 3.1: A path is k-protected if any segment of the path, consisting of (k-1) adjacent nodes and k links connecting these nodes can be protected by backup paths.

For example, 1-protected path implies that a link in a path is protected, and 2-protected path implies that one node and 2 links incident to the node in the path are protected. It is noted that given a resilience requirement from customers, a k-protected path corresponding to the requirement can be decided since the resilience can determine the number of protected components in the path.



(a) A case violating Rule 1 (b) A case violating Rule 2

Fig. 2. The case violating Rule 1 or Rule 2

For example, 1-protected path implies that a link in a path is protected, and 2-protected path implies that one node and 2 links incident to the node in the path are protected. It is noted that given a resilience requirement from customers, a k-protected path corresponding to the requirement can be decided since the resilience can determine the number of protected components in the path.

Now, we investigate the inherent properties of generating candidates of backup path for a primary path in an MPLS network. We show that some links emanating from nodes in a primary path do not contribute to the generation of candidates for backup paths. Path design Rule 3 and Path Rejection Rule which are described below specify these properties.

- Rule 3: Protected segment in the primary path should be disjoint with protecting segment in the backup path except the beginning and ending nodes of the segment.

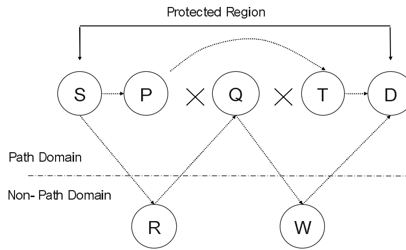


Fig. 3. Example violating Rule 3

Rule 3 implies that no component in the protected segment of a primary path may be used for the construction of backup paths, except the beginning and ending nodes of the protected segment. Figure 3 shows the case which violating Rule 3, where neither the segment <S, R, Q, W, D> nor <S, P, T, D> should not be candidate segment for the backup path construction. In Figure 3, path domain implies the set of nodes in the primary path, and non-path domain implies the set of nodes not included in the primary path.

Theorem 3.1 (Path Rejection Rule): Any link emanating from a node in the protected segment of a primary path which is incident to another node (except the ending node of the protected segment) in the primary path need not to be considered for the backup path design.

Proof: Let us suppose that there is a path <S, .., T, Q, .., D> as shown in Figure 4. Without loss of generality, let us assume that some components (either links, nodes or both) in the segment <S, .., T> fail as indicated by the cross mark in Figure 4. Then, in order to construct a backup path to protect the segment <S, .., T>, we may try to use a direct link <S, Q> from the beginning node S to another node Q in the primary path, as shown in Figure 4. In order to build a backup path in this case, we may go either from Q to R, Q to T, or Q to D. The choice of using either <Q, T> link or the sub-path <Q, .., D> would violate Rule 2. Thus, the link <Q, R> is the only choice for some R in non-path domain. In fact, the selection of any nodes in path domain would violate Rule 2.

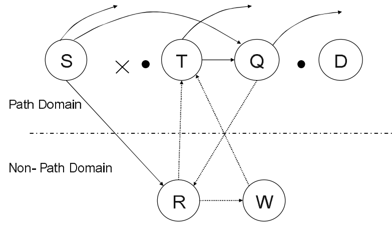


Fig. 4. An example of Path Rejection Rule

As we arrive at the node R, we have to go back to the node T either by selecting the link <R, T> or using some other node W in non-path domain as shown in Figure 4. As we arrive at the node T, there are also two choices: using the link <T, Q> or using the other link. Using the link <T, Q> would violate Rule 1, since a cycle is created, and using the other link to any node in the path would violate Rule 2. Therefore, in order to protect the segment <S, .., T>, any backup path beginning from the direct link from S to another node in path domain would eventually be confronted with violating either Rule 1 or Rule 2. By combining all these facts associated with Rule 1, Rule 2 and Rule 3, we can conclude that any link emanating from a node in a primary which is incident to another node in the primary path need not to be considered for the backup path design. In Figure 4, according to Rule 3, any link emanating from intermediate nodes between nodes S and T should not be used for backup. The link connecting directly the beginning and ending node of a protected segment is the only candidate for a backup segment for the case of using nodes in a path domain. This completes the proof. □

3.2 Availability Testing for Backup Paths of MPLS VPN Service

In this subsection, we derive the conditions for testing the availability of feasible backup paths satisfying the resilience constraints. We first give a definition of path independence in a full mesh MPLS VPN.

Definition 3.2: For a path of a full mesh MPLS VPN, two cycles are said to be path-independent if all the links of two cycles are different except links contained in the path. For example, for the network shown in Figure 5, let us assume that there exist two path cycles $\Phi 1$ and $\Phi 2$, where $\Phi 1 =$ <P1, R, …, Pk+1, P1> and $\Phi 2 =$ < P1, Q, …, Pk+1, P1>. Since there are no links shared in cycles $\Phi 1$ and $\Phi 2$, they are path-independent where the path is <P1, R, …, Pk+1>. □

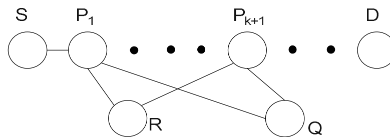


Fig. 5. Two path-independent cycles

Theorem 3.2: For a path, there exist two cycles which are path-independent to each other, and the k links of each cycle are shared with the path. Then, for any failure of links in either one of the cycles which are not contained in the path, we can construct a k-protected path, but not both.

Proof: Without loss of generality, let us assume that there exist two path-independent cycles $\Phi 1$ and $\Phi 2$, where $\Phi 1$ = <P_1, R, …, P_{k+1}, P_1> and $\Phi 2$ = < P_1, Q, …, P_{k+1}, P_1> as shown in Figure 5, where S and D represent the ingress and egress nodes, respectively. It is noted that nodes R and Q are not contained in the path. Assume that some of the links of the sub-path <P_1, P_{k+1}> fail. Suppose that the cycle $\Phi 1$ fails. In other words, either <P_1, R> or <P_{k+1}, R> fail. In this case, we can construct a backup path <S, P_1, Q, P_{k+1}, .., D>. Therefore, the path is k-protected in the case where the cycle $\Phi 1$ fails. We can prove similarly for the case where the cycle $\Phi 2$ fails. If both $\Phi 1$ and $\Phi 2$ fail, the path is disconnected, so that there is no backup path which makes the path to be k-protected. This complete

Theorem 3.3: For an MPLS network with N nodes with full mesh structure, where N > 3 and k > 1, any path with n nodes is k-protected even though ζ number of links including a protected link of the path fails, where

$$\zeta = \begin{cases} (i \times k) + (n-k) & for \quad j \le k \\ (i \times k) + (n+j-2k) & for \quad j > k \end{cases} \tag{2}$$

Here, n = N – 1, n = 2ik + j, j = 0, 1, …, (2k-1), and i is a non-negative integer.

Proof: Without loss of generality, let us assume that the path with n nodes is configured as shown in Figure 6, where the cycles <P_0, P_k, …, P_1, P_0> and <P_1, P_{k+1}, P_k, …, P_1> are independent. It is noted that we do not have to consider the configurations which would violate the path design rules in previous Subsection 3.2. It is easily shown that the number of these types of cycles is equal to (n-k), each of which is made with direct links connecting two nodes which is located k-link away in the path.

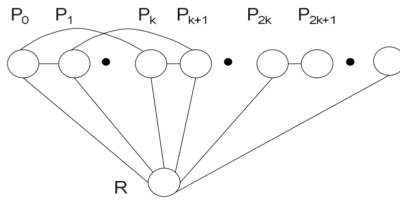


Fig. 6. MPLS VPN configuration for evaluating k-protected path

Next, we show that if there are (i * k) numbers of independent cycles at the lower part of Figure 6, i.e., the cycle starting from the node R. It is also easily shown that the cycle <R, P_0, P_1 …, P_k, R> and <R, P_1, …, $P_{k+1,}$ R> are independent. In this way, we can proceed to make independent cycles starting from R through the link <R, P_0>, and next another cycle through the link <R, P_1>, and so on, until the link <R, P_{k-1}> being covered. However, when we arrive at the link <R, P_k>, it is found that we can not make the cycle <R, P_k, P_{k+1} …, P_{2k}, R> to be independent cycle, since the link <R,

P_k> will be shared between two cycles if being proceeded. We can restart the building of independent cycle from the link <R, P_{2k}> to make the independent cycle <R, P_{2k}, P_{2k+1}, …, P_{3k}, R> if there exist k more nodes in the path starting from P_{2k}. If the number of remaining nodes starting from P_{2k} is less than k, we can not make any other independent cycles covering k links.

Let us assume that there are j numbers of remaining nodes. For a path with n nodes, the path can be represented as a multiple of 2k with remain j nodes, i.e., n = 2k x i + j for some i and j where i is a non-negative integer, and j = 0, 1, …, (2k-1). In this case, for each 2k block of nodes, there can exist k independent cycles, so that the total possible number of independent cycles becomes (i * k) if n = 2ik. If j is less than or equal to 2k, the number of independent cycles would still be (i*k) as mentioned before. If j > k, we can start to make another (j-k) independent cycles, so that the total number of independent cycles would be (i*k) + (j-k). Since there are (n-k) number of independent cycles from the direct link connection, the total number of independent links for the case with $j \leq k$, becomes (i*k) + (n-k). For the case with j > k, it equals to (i*k) + (j-k) + (n-k). This completes the proof.

It is noted that ζ is a maximal number of allowable failed components which could make the primary path to be k-protected under any multiple link failure scenario. Now, let us consider the general case where n < N. We first derive the solution with the assumption that any direct link between nodes which are not involved in the path are not allowed for the construction of backup paths. After deriving the solution, we derive the solution for the general case in which the assumption is removed.

4 Dynamic Path Management of MPLS VPN with Resilience

The establishment and maintenance of VPN paths are dynamically maintained in the sense that both the primary and backup paths are setup concurrently in advance, and when failures occur, all the paths within a domain which are affected by the failures are reconstructed dynamically. This is a hybrid approach in which the merits of both path protection and path restoration methods are taken into account.

The methodology of the dynamic management of BGP/MPLS VPN consists mainly of the following four phases. Here, a component implies either a node or a link as explained previously.

Phase (1) Initialization;
Phase (2) When failure notifications from MPLS backbone arrive, assign a new set of VPN paths to minimize the service disruption.
Phase (3) When overload notifications arrives from MPLS backbone, then distribute the VPN traffic load to other backup paths to maximize the performance.
Phase (4) Update periodically the availability of both primary and backup paths of the MPLS backbone networks with a specified time interval.

In Phase (1) Initialization, a set of primary and backup paths are determined for each VPN, which may take into account the policy of service provider, the quality of service constraints, and the resilience constraints of the MPLS VPN service. The instrumentation may require the extension of VRF tables and the routing information base (RIB). In Phase (2), upon detecting failures of P nodes or links connecting them,

the PE nodes may disable the primary path, and enable the backup path, and switch the data traffic to the backup path. While the backup path is operational, the failure recovery operation can be performed. In Phase (3), the load balancing can be done, and in Phase (4), it periodically checks whether a primary path or backup path for a given MPLS VPN service is available or not. Here, it simply checks whether any path is malfunctioning due to the failures of constituent components of the path. Phase (2) is explained in more detail in Algorithm Dynamic_Path_Mangement which is described below. Every PE executes this procedure whenever a failure notification arrives. Here, the Resilience_Constraint may specify the value k for the k-protected primary path.

Algorithm Dynamic_Path_Mangement (Failure_Notification,Resilience_Constraint);
Begin
  Step (1) If failure notification is not related to paths starting from PE,
    Then return ("Irrelevant Failure Notification");
  Step (2) If a primary path is damaged due to the Failure_Notification,
    Then {
    (2-A) Switch immediately the VPN data traffic to the available backup path;
    (2-B) When the components in the primary path are repaired, restore the primary path;}
  Step (3) Reconstruct the backup paths with Resilience_Constraint, which might have been affected by Failure_Notification ;
End

If failure notifications arrive from P nodes in the MPLS domain such that the primary path is affected, it then immediately switches the VPN data traffic from the primary path to the available backup path. It is noted that a path is affected by a failure notification if any component in the path is associated with the failure. It also reconstructs the backup paths which might have been affected by the failures. The Step (3) of the Algorithm Dynamic_Path_Mangement is described in detail in the backup path reconstruction algorithm, Algorithm Construct_Backup_Path, which is shown below. Here, # (Failure_Notification) implies the number of failed components for the simplicity of explanation. Path and BackupPath indicate the primary path and the backup path, respectively, with array data type. N denotes the total number of nodes. It is noted that once protection mode and a resilience constraint are specified, the value k for a k-protected path can be determined. This implies that the value ζ of Theorem 3.3 can be obtained from Resilience_Constraint as long as a specific protection mode is chosen. Details of deriving the value from Resilience_Consraint is for further study area which we are currently working on. For the simulation, we assume that 1:1 protection mode is applied.

Algorithm Construct_Backup_Path (Failure_Notification, Resilience_Constraint);
  /* This algorithm constructs a backup path using the backup path design rule. */
  /* ζ is determined by the value k of 'k-protected' primary path which can in turn be determined by Resilience_Constraint. */
Begin
    If # (Failure_Notification) $\leq \zeta$
    Then {Pointer := 0; BackupPath[Pointer] := ingress_PE;
      Do {Select a node R where R $\notin$ Path and R $\notin$ BackupPath
        and $\exists$ a link <BackupPath[Pointer], R>;

```
                BackupPath[Pointer + 1] := R ;
                Pointer := Pointer + 1;
        UNTIL ( R = egress_PE or All nodes which are not belonging to
                Path is covered);}}
End
```

Algorithm Construct_Backup_Path first checks whether it is feasible to construct any separate backup paths for the primary path for the case where at least #(Failure_Notification) of MPLS links fail due to some faults. By applying the testing condition derived in Theorem 3.3, if the testing condition is satisfied, we know that there exists an available backup path, and we can rapidly build the backup path which is disjoint with the primary path. For searching the feasible backup path satisfying the resilience constraints, the algorithm uses the backup path design rule in the Subsection 3.1. The backup path construction algorithm is rather simple with the computational complexity of O(n) where n is the number of nodes. For the case where the testing condition is not satisfied, there might not be available backup paths. This requires heuristic path finding algorithms employing provider's policy, and the system robustness, and so on. This is also a further study area which is under investigation.

In summary, each PE node is assumed to have the information of the entire BGP/MPLS network such as topology, available bandwidth, P node capacity and so on. The testing condition of Theorem 3.3 is paraphrased as follows: If the number of failed components is less than or equal to some value which can be determined by the resilience constraints, we can construct at least one separate backup path for a given primary path associated with BGP/MPLS VPN service even though multiple simultaneous links fail in the backbone network. Here, the number of failed links can be identified by analyzing the arrived failure notification messages from all P nodes in the MPLS backbone to a PE node.

5 Examples and Simulation Results

Figure 7 shows the test configuration of MPLS backbone for simulation, consisting of 5 nodes, and the links between them representing logical connections. We first show the examples for explaining the resilience concept. In Figure 7(a), the primary path is <P0, P2> and the backup path is <P0, P1, P2>. Here, P0 and P2 are the source and destination nodes, respectively. For the primary path shown in Figure 7(a), we can make the primary path to have the resilience value of 1, i.e., being 1-protected. In this case, applying Theorem 3.3, we know that ζ becomes 3 since n is equal to 2, and (N-n) is equal to 3. Since n is equal to 2, we know that i is equal to 0 and j becomes 2 in Theorem 3.3. Thus, the maximal number of link failures becomes 3, including failures in the primary path. In other words, it is always possible to construct a backup path as long as the total number of simultaneous link failures, including one in the primary path, is less than 3. Figure 7(a) shows the case in which one link in the backup path fails. When a component in the primary path fails, it is feasible to construct another backup path since the total number of link failures in this case is 2.

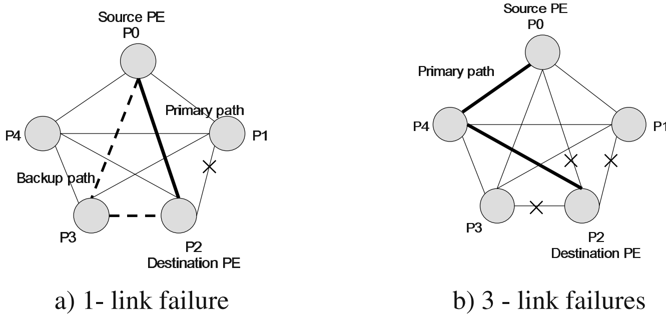


Fig. 7. General case to show non-dependency of ζ on R

Figure 7(b) shows the situation where the primary path consists of 2 links. In order to make the primary path with resilience value of 1, we should build 2-protected primary path. Applying Theorem 3.3, it is calculated that the maximal number of link failures is 3, which includes a link failure in the primary path. For the case of multiple link failures in Figure 7(b), it is not feasible to find a backup path for making the 2-protected primary path, since the total number of link failures would be 4 including one possible link failure in the primary path <P0, P1, P2>.

As described in Section 4, the data for VPN service enters the source node, and is delivered to the destination node if there is no failure in the primary path. If the primary path experiences a failure, the failure notification is sent to the source node, and the data is delivered to the destination via a backup path. If there is a failure in the backup path, it tests the availability of the backup path using the testing condition of Theorem 3.3. If satisfied, a new backup path is established very rapidly, and the data is switched to the new backup path. Before transmitting the data, it first checks whether input buffer at MPLS node is available. If it is available, it delivers the data into the buffer. If not available, it either discards the data or switches the data to the available backup path according to the load sharing policy.

In simulation environment shown in Figure 7, the input data traffic enters into the node P1, and is transmitted to the node P2. Here, P1 and P5 play the role of the source node and destination node, respectively. The source node usually waits for a very short time interval to check whether there are other failure notification messages, and tests the condition for backup availability if a backup path is not reserved or damaged. If the testing condition is satisfied, it can rapidly reconstruct the backup path using the backup path design algorithm. The source node can then switch immediately the input data traffic to the backup path, resulting in the high service availability with minimal service disruption. Before transmitting the input data, the source node also checks whether the input buffer at the MPLS node is available. If available, it delivers the data into the buffer. If not available, it either discards the data or switches the data to the available backup path according to the above load sharing policy.

In Figure 8, we show the simulation results of the throughput for changing input load ranging from 0.5Erlang to 4Erlang where the data rate of each link is assumed to be 100Mbyte, the propagation delay of each link being 10ms and the link error

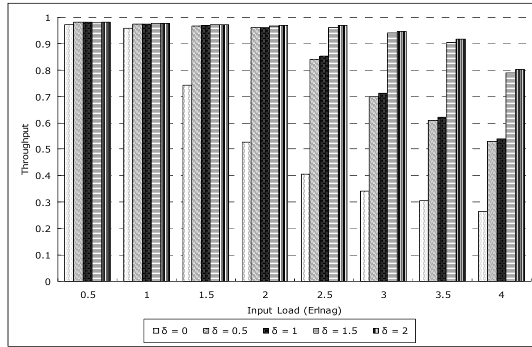


Fig. 8. Throughput vs. Input load

probability to be 0.01 with uniform distribution. The input load is assumed to have a Poisson probability distribution with the average input rate 0.01. For simplicity, the message size is fixed at 1kbyte. Here, the throughput is defined as the ratio of output and input data traffics. The throughput is measured by changing the resilience value, denoted as δ, for the values of 0, 0.5, 1 1.5 and 2. For the case of resilience value 0, the data loss rate rapidly increases as the input data rate increases. However, the data loss rate decreases as the value of resilience increases. This is because with increasing value of resilience, the data can be delivered more reliably.

For example, for the resilience value of 1, approximately 98 % of input data can be transmitted from the source node to the destination node for the input data of up to 2Erlang. However, the throughput slowly drops to about 85 % and even to 70 % when the input traffic load increases to 2.5 and 3Erlang, respectively. Figure 8 shows that the throughput generally decreases as the input load increases. However, the decreasing rate is reduced as the resilience value increases. This is because the larger resilience value is, the greater is the load sharing when the primary path is either failed or overflowed.

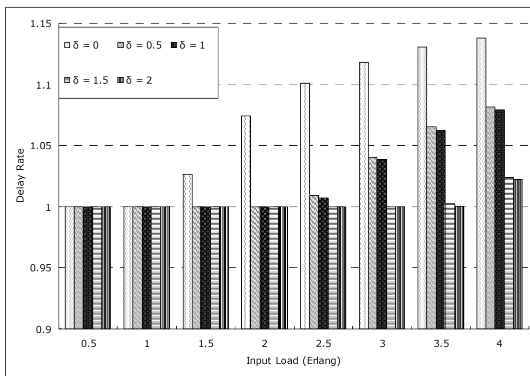


Fig. 9. Delay rate vs. Input load

In Figures 9, we show the simulation results for the delay by input load ranging from 0.5Erlang to 4Erlang with the same simulation environment as in Figure 8. The delay rate is measured by simulation by changing the input load from 0.5 to 4Erlang for the resilience values of 0, 0.5, 1, 1.5 and 2. The delay rate is generally increases as the input loss increases. However, the increasing rate becomes smaller as the resilience values become larger. This is because with the increasing value of resilience, the data can be delivered more reliably. For the cases of resilience value larger than or equal to 1, the input data load can be equally shared among available backup paths.

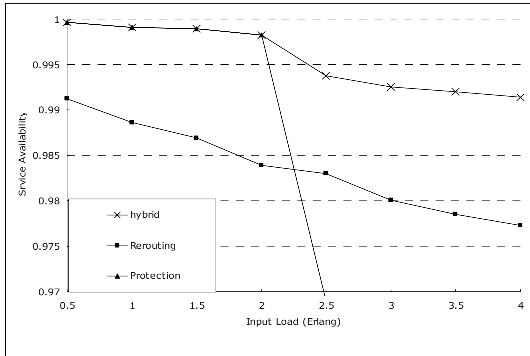


Fig. 10. Service availability vs. Input load

In Figures 10, we show the variations of the statistical average of service. There are two general recovery mechanisms proposed for MPLS path management in IETF standard: protection and rerouting mechanisms. In protection mechanism, the backup path is always preserved in advance, while in rerouting, no backup path is preserved in advance. Our approach can be called a hybrid approach in the sense that backup paths satisfying the resilience constraint, i.e., ‘k-protected’, is established in advance, similarly as in protection mechanism, but with extension of provisioning of new backup paths satisfying the resilience constraints under multiple failure occurrences. In other words, when backup paths are also affected by the multiple failure occurrences, alternative backup paths are constructed, similarly as in rerouting mechanism.

In order to compare the performance of the proposed approach with previous mechanisms described in IETF standards, we have assumed that a separate backup path is maintained for protection mechanism, which is corresponding to the case with resilience value of 1. For rerouting mechanism, it is assumed that a separate alternative path is being constructed when link failures occur in the primary path. We have measured by simulation the service availability for the three approaches: protection, rerouting, and proposed hybrid approach. The service availability is defined as the ratio of the total uptime for a LSP to the total duration of the LSP. As shown in Figure 10, the service availability of the hybrid approach is better that those of protection and rerouting approaches under varying input load conditions. Although

the service availability of protection mechanism is almost similar to that of the hybrid approach, the service availability sharply drops when the input load increases beyond 2.0. This is because that there would be more chance of link failures in the primary path as the input load increases, in which case most of input data traffic can not be delivered to the destination node.

In comparison with the rerouting mechanism, the hybrid approach can provide better service availability than the rerouting mechanism. This is because the backup paths can be maintained with guaranteed resilience, while the rerouting mechanism should find the alternative backup paths every time a failure occurs. When multiple failures occur almost simultaneously, the rerouting mechanism may try to resolve each occurrence of a failure. In comparison with the protection mechanism, it is not possible for the protection mechanism to transmit the data if some components of the backup path fail, while in the hybrid approach, it can find a new backup path.

Furthermore, the proposed hybrid approach can handle multiple simultaneous failures, while trying to find a feasible backup path. Although it is not shown in Figure 10, for the case where there are multiple failure occurrences which would not allow the construction of any backup path, the hybrid approach may immediately notice this worst-case scenario, and, may propose to find sub-optimal solution using some heuristics. However, the traditional rerouting mechanism may not notice the worst situation, and may try to continuously find the solution which may not exist at all, which may eventually result in the large service disruption. In summary, it is found that the hybrid approach could provide better service availability than those of protection and rerouting mechanisms for MPLS VPN path management.

6 Conclusion

In this article, we present the methodology for resilient path design and management for MPLS VPN service. We present methods which can dynamically configure the paths of MPLS VPN service satisfying the TE resilience requirement from the customers. We develop rules, testing conditions and algorithms for fast backup path construction which could make the MPLS VPN service to be available without disruption satisfying the resilience requirement. The simulation has been done to evaluate the various performance figures and service availability with respect to the resilience value. It shows that both delay and throughput can be increased as the resilience value increases.

In the methodology of the dynamic management of BGP/MPLS VPN service, we measure the availability of paths by only taking into account the facts that it is working correctly or not. However, it is also possible to test the utilization of the components, and the availability of the paths can be measured by these factors, which is for further study area. In finding available backup paths, if the testing condition is not satisfied, there might not be backup paths, satisfying the resilience constraints. This requires heuristic methods, which is also a further study area which is under investigation.

References

1. IETF MPLS Working Group : http://www.ietf.org/html.charters/mpls-charter.html.
2. Mannie, E., al, et.: Generalized MPLS Architecture :Internet Draft, draft-ieft-ccamp-gmpls-architecture-07.txt. May 2003.
3. Rosen, E., Rekhter, C. Y. et al. : BGP/MPLS IP VPNs. draft-ietf-ppvpn-rfc2547bis-04.txt. May 2003.
4. Bates, T., Chandra, R., Katz, D., Rekhter, Y. : Multiprotocol Extensions for BGP-4. RFC2858.
5. Tony Bogovic, Seminar 2 : MPLS Overview and Applications. Telcordia Technology. Feb. 2002.
6. Awduche, D., Berger, L., Gan, D., Li, T., Srinivasan, V. and Swallow, G. : RSVP-TE Extensions to RSVP for LSP Tunnels. RFC 3209. December 2001.
7. Jamoussi, B., Andersson, L., et al. : Constraint-Based LSP Setup using LDP : RFC 3212. January 2002.
8. Park, J. T.: A Model of Resilience for MPLS/GMPLS Service Availability. Submitted to ISAS2004.
9. Li, G., Wang, D., et al : Efficient Distributed Path Selection for Shared Restoration Connections. IEEE INFOCOM 2002. pp.140 – 149.
10. Sharma, V. and Hellstrand, F.: Framework for Multi-Protocol Label Switching (MPLS)-based Recovery. February 2003.
11. Huang, C., Sharma, V., Owens, K. and Makam, S.: Building Reliable MPLS Networks Using a Path Protection Mechanism. IEEE Communications Magazine. March 2002. pp. 156-162.
12. Banerjee, A., Drake, Jet., El. : Generalized Multiprotocol Label Switching_An overview of Signaling Enhancements and Recovery Techniques. IEEE Communications Magazine, July 2001, pp. 144-151.
13. Guangzhi Li, Jennifer Yates, Robert Doverspike and Dongmei Wang : Experiments in Fast Restoration using GMPLS in Optical/Electronic Mesh Networks. Postdeadline Papers Digest, OFC-2001. Anaheim. CA. March 2001.
14. Doverspike, R. and Yates, J.: Challenges for MPLS in optical network restoration. IEEE Communications Magazine. Feb. 2001. pp. 89-96.
15. Papadimitriou and Mannie, E.: Analysis of Generalized MPLS-based Recovery Mechanisms (including Protection and restoration). draft-ietf-ccamp-gmpls-recovery-analysis-01.txt. May 2003.

Highly Available Location-Based Services in Mobile Environments

Peter Ibach and Matthias Horbank

Humboldt University, Computer Science Department, Computer Architecture and Communication Group, Rudower Chaussee 25, 12489 Berlin, Germany
{ibach, horbank}@informatik.hu-berlin.de

"*We're not lost. We're locationally challenged.*"
John M. Ford

Abstract. We show how to use Web Services standards for propagation, discovery, and composition of location-based services in mobile environments. To achieve semantic interoperability we express location information in XML using context-specific ontologies. The achieved interoperability allows for context-aware on-demand service composition making the composite service highly available and resilient to environmental dynamics and uncertainties.

1 Introduction

Location-based services (LBS) are services that utilize their ability of location-awareness to simplify user interactions and adapt to the specific context. With advances in automatic position sensing and wireless connectivity, the application range of mobile LBS is rapidly developing, particularly in the field of geographic, telematic, touristic, and logistic information systems.

However, present LBS are to a large extent incompatible with each other and unable to interoperate on location semantics. They are mostly bound to a specific technology reflecting the preferences of the service provider. Typically, proprietary protocols and interfaces are employed to aggregate the different system components for positioning, networking, content, or payment services. In many cases, these components are glued together to form a monolithic and inflexible system. If such a system has to be adapted to another technology, e.g., the change from GPS positioning to in-house WLAN or Bluetooth positioning, it has to be entirely reengineered. Due to the dynamic nature of mobile environments, available resources as well as achievable quality of service levels are incessantly changing. Thus, adaptivity – the ability of steady interoperation of variable resources under changeable connection conditions – becomes crucial for service end-to-end availability in mobile environments.

Let us consider a position sensing service, for example, a satellite-based GPS. If a mobile device moves from outdoor to indoor environments, the signal will likely become unavailable and position sensing will fail. Without the location information expected from this subservice, composite services depending on it will become

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 134–147, 2005.
© Springer-Verlag Berlin Heidelberg 2005

unavailable as well. To arrive at seamless operation, on-the-fly switchover to an alternative position sensing service using a different technology is required. To choose from multiple possible position sensing services, the decision has to consider service availability, quality of service properties, and costs. In the near future, most mobile and wearable devices are expected to have multiple position sensing technologies at disposal, e.g., GPS, GSM, WLAN, and Bluetooth. Nevertheless, new technologies, like at present WiMax or RFID, are continuously emerging. Thus hardware devices and software components, their interfaces and architecture have to be able to deal with changing conditions to make mobile location-based services highly available.

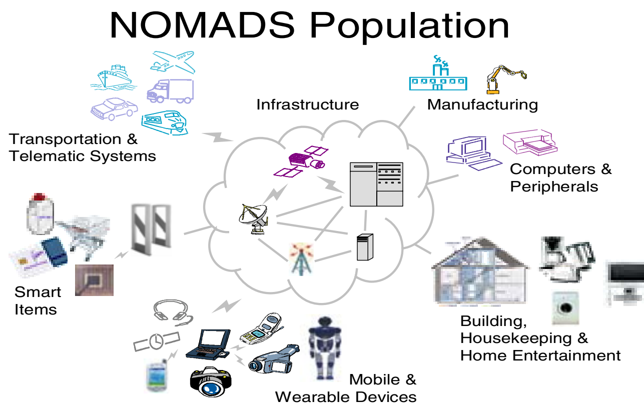


Fig. 1. NOMADS – *N*etworks *o*f *M*obile *A*daptive *D*ependable *S*ystems (for further investigations see [8]). High mobility and dynamics require systems' self-reliance, context awareness, and adaptivity to accomplish dependable operation

Generally, flexible and open standards are the key to enable interoperability and open the door for a prospering variety of new business ideas and services. Web Services Grid and Semantic Web standards are expected to overcome past hurdles by a universal standard for managing services and resource on the web. These standards might accelerate the next chapter of the electronic revolution. Frameworks such as *NOMADS* (see Fig.1) promise to deliver mobile, adaptive, and dependable interoperability, challenging numerous new applications and business opportunities.

Here, we investigate applicability of Web Services Grid standards in the envisaged domain of location-based services in mobile environments and propose a generic model for improved flexibility and availability. Core strategy is the on-the-fly adaptation to a specific situation. This is accomplished by tracking contextual changes and context-aware on-demand composition of appropriate services.

The paper is structured as follows: In Section 2 we summarize background and related work in the area of location-aware computing. In Section 3 we discuss the economic impact of location-based services, service-oriented architectures, and the challenges to be tackled. Section 4 describes the technical approach we are pursuing, demonstrates how it helps delivering improved *m*obility, *a*daptivity, and *d*ependability

(we call these the *MAD* properties), and how location information can be processed semantically. Section 5 describes our implementation experiences and some use case scenarios and Section 6 concludes the results and gives an outlook to future work.

2 Background and Related Work

While in the 60s and 70s mainframe computing flourished and in the 80s and 90s personal computing got predominant we are now entering the age of ubiquitous computing [20] where – in the near future – almost any device will have certain computation and communication power. Pervasive computing further emphasizes the expectation that computational devices and connecting infrastructure are getting more and more omnipresent. With the widespread of mobile and wearable devices capable of location sensing, lots of research has focused on location-based services [14] and location-aware computing [5]. A central problem in context-aware computing is the appropriate description of environmental characteristics [10].

Another technology leap is induced by Radio Frequency Identification (RFID). Through RFID labels, almost any device or item can be cost-effectively transformed into a “smart” information source. In conjunction with the electronic product code infrastructure (EPC Network), smart items will be capable of wirelessly telling who and where they are, what status they have, and which services they offer. An “Internet of things” [1] with billions and soon trillions of seamlessly interconnected devices is about to take over the era of traditional computers and computer networks.

By these developments, information technology is taking a big step towards providing a comprehensive real-time picture of the physical world. Through the spatial organization of information together with location-based services and semantic interoperability, virtual and real spaces will tightly interconnect.

Since the vast possibilities also involve misuse, security and privacy issues are accompanying the developments and receive growing attention. To ensure that humans are not overwhelmed by omnipresent and omnipotent technology, “Ambient Intelligence” [4] represents a vision of sensitive and responsive electronic environments putting humans in the centre of technological developments.

Considering the question of how to organize the evolving number of interacting devices led to various approaches trying to exploit analogies from examples in nature. A variety of concepts emerged which are inspired by physical, biological, sociological, or economical analogies. Basic idea is to design and structure composite systems such that they are able to meet upcoming challenges by actions of its largely autonomous entities. Under the term “autonomic computing” IBM summarizes eight[1] core “elements” [9] – comprising self-configuring, self-healing, self-optimizing, and self-protecting – that are intended to guide the development.

Fundamental concept is the composition of systems by extensive reuse of commodity software/hardware components. Component-based software engineering [9] tries to extend paradigms of object-oriented software engineering such that components – in contrast to objects – can be adapted to the specific conditions of a run time environment without additional interventions (hot deployment). The access

[1] In current implementation, they have been compressed to four.

to a component is exclusively accomplished through its interface. Syntax and semantics of a component should be comprehensively described in its specification.

Agents (also referred to as actors) are especially “intelligent” components that show improved adaptivity in dynamic environments through autonomous goal tracking, context sensitivity, mobility, reactivity and proactivity. This direction is pursued by agent-based software engineering [7, 13] and further extensions for business environments referred to as business agents or agentified enterprise components [17]. Those actors have negotiation capabilities, possess context models to adapt to different deployment contexts, and are able to deal with uncertainties that may arise from unforeseen changes and errors.

To accomplish interoperability on higher levels of semantics, one has to agree on a suitable ontology which defines terminology and relations for each specific application area. A widely accepted ontology that models physical objects and their location is used in the Geographic Information System (GIS), standardized by the OpenGIS Consortium. GPS position sensing together with geocoding services for visualization of geographic information [15] enjoys growing popularity on mobile devices. Yet, seamless outdoor to indoor transitions, global scalability, and changeover to different services, e.g., providing different cartographic material, are usually not addressed. The Physical Markup Language of the EPC Network, standardized by the Auto-ID Center, is intended for product classification, but also allows for spatio-temporal annotations for object tracking and supply chain management. Moreover, the World Wide Web Consortium is extending the Resource Definition Framework to relate Web Content to its associated physical location. In all these attempts, however, expressiveness of location semantics is still in its infancy.

3 Challenges and Expected Benefit

Enterprise applications were initially developed on closed, homogeneous mainframe architectures. In the explosively growing heterogeneous landscape of IT systems in the 80’s and 90’s integration of intra- and inter-company business processes became one of the most important and most cost-intensive tasks of the IT economy. Due to missing or non-transparent standards many enterprises pursued integration by extremely expensive ad-hoc-solutions. With the spreading of the Internet and the increasing importance of electronic business, open Internet-oriented solutions have emerged. Enterprise-internal monolithic software was broken into smaller, autonomous, and flexible components. This enabled the access to services not only enterprise-internally, but along the whole value chain to suppliers, distributors, and customers. We characterize this observation as a shift from rigid systems to flexible service-based architectures, where open and flexible services are the basic building blocks (see Fig. 2).

Grid computing is a service-based form of shared resource usage that is intended to make the access to computation power as simple and omnipresent as electric power supply. It extends peer-to-peer computing as well as cluster computing in a global scale in order to “enable the sharing, selection, and aggregation of geographically distributed heterogeneous resources dynamically at runtime depending on

their availability, capability, performance, cost, and users' quality-of-service requirements" [3].

The Web Services Grid is aimed to overcome the two predominant challenges at the same time: uniform access to services *and* processing resources. Web Services are intended to facilitate the application to application interaction extending established Internet technologies. While some skeptics ask why the services paradigm might prevail against preceding concepts like CORBA, significant advantages are:

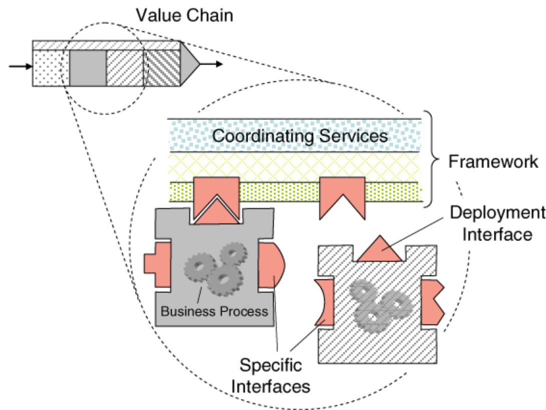


Fig. 2. Loosely coupled, open architectures play an increasingly important role in value chains due to improved possibilities of interoperability, integration, composability, flexibility, reusability and thus increased efficiency at reduced total cost of operation

- market power of stakeholders
- productivity of tools
- integrated support for asynchronous dependable messaging
- service choreography with workflow engines ("two stage programming")
- interoperability and flexibility – paid by some performance loss – through loose coupling based on the eXtensible Markup Language (XML)
- enhanced protocols for propagation, discovery, and invocation of "lightweight" services in ad-hoc networks

Web Services and Grid toolkits like the Globus Toolkit or the Emerging Technology Toolkit (ETTK) based on the Open Grid Services Architecture (OGSA) have helped establishing standards. Component based software for embedded systems [12] and lightweight services [16, 11] expanded the domain to span from distributed client-server applications and globally networked e-business processes down to next generation heterogeneous embedded systems. These developments paved the way towards the general paradigm of service-oriented computing where all kinds of entities are providing, using, searching, or mediating services while efficiently exploiting the available resources. Driving the widespread acceptance of the service-oriented paradigm, LBS might reveal the enormous economic potential of dynamic value webs [6] in mobile business.

4 Adaptive Location-Based Services

From the perspective of a mobile user, the environment is ever-changing as he moves from one location to another. As earlier explained, adaptivity to location characteristics is essential for mobile service availability. In our approach, adaptivity of a composite location-based service – we call these services Adaptive Location-Based Services (ALBS) – is accomplished by choosing the appropriate chain of subservices for composition (see Fig. 3).

Prerequisites are general discoverability, interoperability and composability of subservices through standardized communication protocols and directory services. In the Web Services standard, interoperable communication is accomplished by exchanging XML data over HTTP. But Web Services are not restricted to the WWW or a specific protocol. Rather, it is a promising solution for adaptive application synthesis in distributed, dynamically changing environments. The notion of Web Services goes beyond prescribed client-server communication. Emerging standards for directory services such as UDDI [19] or Web Services Choreography (e.g., BPEL4WS [2]) allow for dynamic discovery of services and composition of multiple Web Services to fully-fledged distributed applications.

Consider a location-based service that requires some input, e.g., accurate position information or the user's choice of payment. The user might present these data to the LBS manually. Likewise, this information might be the result of a preceding Web Service which, for example, reads the geographic position from an attached GPS device. In case of payment, information about the user's choice could be sent to an accounting service which, for example, uses a direct debit authorization. For service composition it is not necessary to know how the accounting is actually performed or how access to the GPS device is implemented, as long as one can trust the responsible Web Services. Authorization and trust will be fundamental for the success of location-based services and Web Services composition. Moreover, protecting privacy regarding the user's trace of information is a severe issue. Further dangers of intrusion to take care of are service spamming, where undesired services are propagated, and service spoofing, where insecure services are offered under disguised identity. Ongoing developments in the Web Services Trust Language (WS-Trust) therefore accommodate a wide variety of security models.

4.1 Using Web Services for ALBS Implementation

We use Web Services standards to implement the appropriate selection of subservices and to process their composition. These comprise the service interface description in the Web Services Description Language (WSDL). In an interface description the *port type* specifies the service's request/response-behavior. A service instance is accessed through a *port*. Each port has to bind to a port type and has to support additional binding information, e.g., the used protocol. In Web environments the Simple Object Access Protocol (SOAP) might be a primary candidate, but other protocols such as CICS (Customer Information Control System) or dependable message queuing can be utilized as well.

For each application to be composed of subservices, a flow through *required* port types and *optional* port types guides the composition process. This flow can be specified using choreography languages (e.g., WSCL or BPEL4WS). Ongoing

developments in Web Services Choreography incorporate far-reaching flow control where, for example, a group of services can be processed transactional or the flow may branch with respect to optional services availability or in case exception occurs. The composition process can be graphically expressed by a path through a network of accessible ports:

The composition process is triggered at service invocation. Whenever an ALBS is invoked, it is dynamically composed of suitable ports. Among the ports of each port type, the best match will be taken with respect to the specific context that determines availability and suitability of each port. Therefore, the Web Services Policy Framework

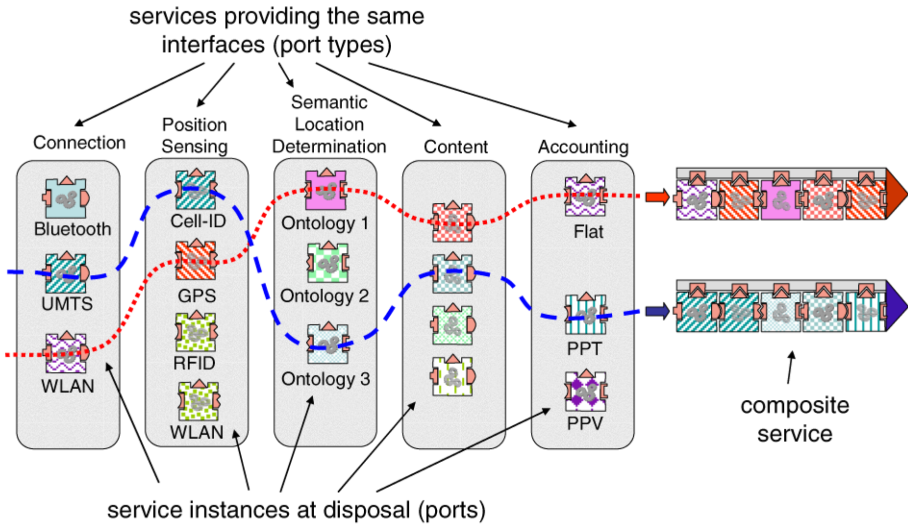


Fig. 3. Composition Process

defines general purpose mechanism for associating policy expressions with subjects. Thus, specific property assertions can easily be attached to ports and registered in the repository.

For successful composition, at least one port of each required port type has to be accessible. Of course, subsequent composition processes will not take place if no relevant context change occurred. Context tracking, event filtering, and event notification combined with prediction techniques might further enhance the composition process and reduce latency time.

In the example presented in Fig. 3 there are five port types:

- **Connection:** Services of this type allow for access to available networks. This could be WLAN, LAN, Bluetooth, GSM, GPRS, or UMTS connections. Properties attached to these ports comprise information about bandwidth, costs, power consumption, encryption, or latency time.
- **Position Sensing:** This port type provides location information. Ports can be GPS-receivers, services based on stationary RFID-tags, or Bluetooth transmitters. Other services, e.g., based on WLAN-positioning or Cell-ID in cellular networks, are possible candidates as well. The properties should contain information about the

accuracy of the location information. Extensions of position sensing services might be able to recognize direction, speed, and variance of movement. (WLAN and Bluetooth positioning base on signal strengths of different access points. For each position, these signal strengths exhibit a certain characteristics that can be translated into location information by a signal characteristics map. Since the signal strengths vary, the map needs periodic update. Nevertheless, coverage and accuracy of the positioning may be insufficient for some LBS. However, this way PDAs, laptops, or other mobile devices can locate themselves independently of GPS availability.)

- **Semantic location determination:** Information about location semantics is offered by this port type. Input is the context-specific sensor data (containing geographic location, RFID numbers, available services, or other specific characteristics that
- could be utilized to reason about the position). The response includes the semantic position according to a given ontology.
- **Content:** This port type offers content for a given geographic or semantic location. It receives a message with the location information which then is processed. The returned message contains information (text, pictures, audio, or video if requested) about the given location. To process the location information semantically, some common ontology is required (see Section 4.4. for further investigations on location semantics).
- **Accounting:** Accounting port type allows on-demand billing of services used.

4.2 Run-Time Adaptation

The sequence chart (see Fig. 4) shows the message sequence of service interaction. The setup in this example consists of a service instance supervising the application control flow, the registry, e.g., an UDDI-implementation, two ports connecting to position sensing services (a GPS service and a WLAN positioning service), and two content ports. The example indicates how the composite service remains viable if some of its ports (here, the GPS positioning service) become temporarily unavailable, and how on-the-fly switchover to a replacement service (here, a WLAN positioning service[2]) takes place.

The first sequence (messages 1-8) shows a service request using the available GPS service:

1. Search the registry for position sensing ports
2. Registry returns a GPS-receiver service port
3. Request position from returned port
4. GPS-receiver returns position
5. Search the registry for content port
6. Registry returns port of Content Provider 1
7. Request content from returned port
8. Content Provider 1 returns content data, e.g., a city map in which the building is located

---

[2] Here, a connection switchover, e.g., from UMTS to WLAN connection, will probably occur. However, this is processed analogously and is not addressed in the Figure.

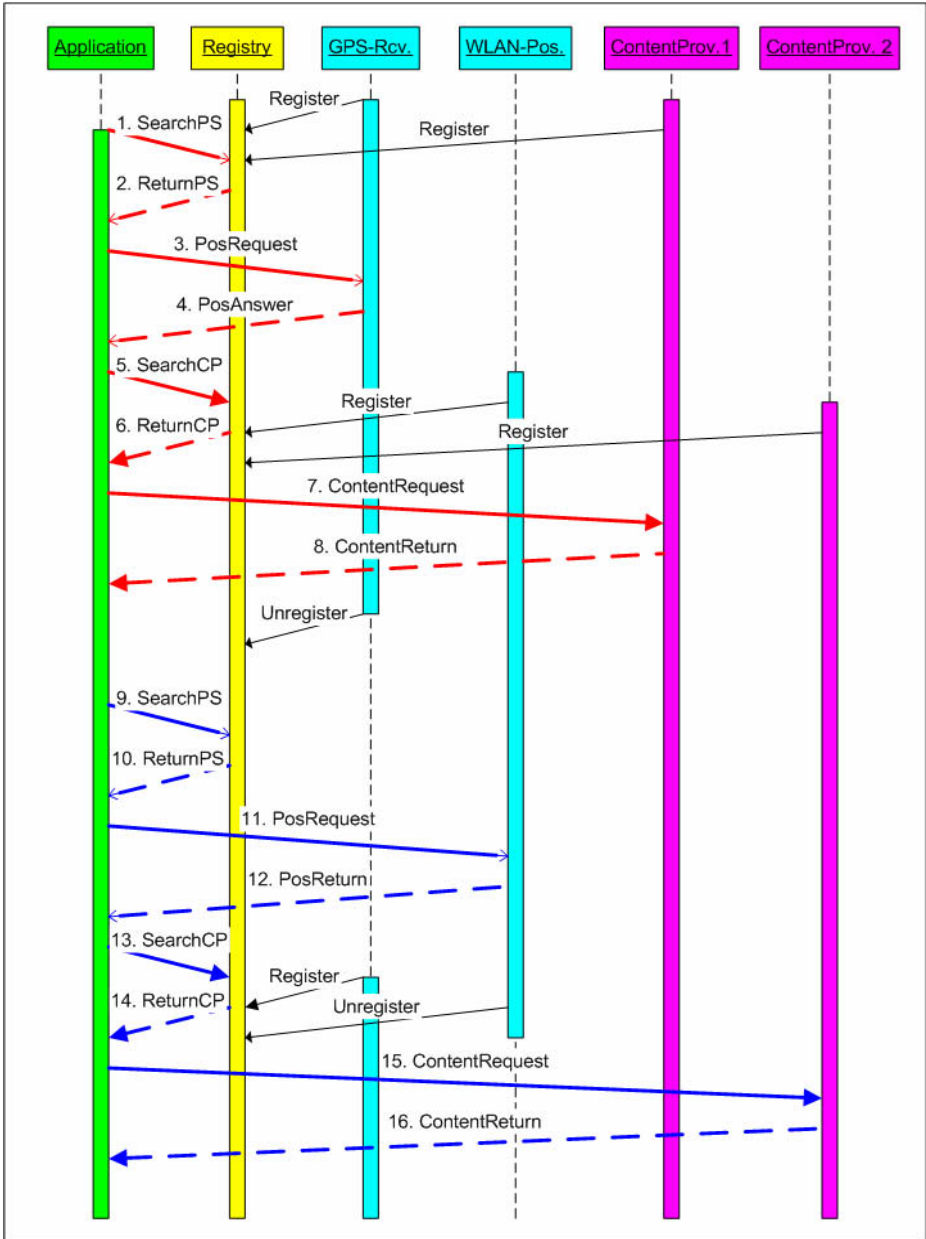


Fig. 4. Sequence chart of service interaction

Before message 9 is being sent, possibly the mobile user is entering a building, where the GPS device cannot receive the satellite signal and therefore unregisters its

service from the registry. Supposing an in-house WLAN positioning service becomes available, the second sequence (9-16) shows the service request after this context change:

9. Search the registry for position sensing port
10. Registry returns port of WLAN-positioning service
11. Request position from returned port
12. WLAN-positioning service returns position
13. Search the registry for content provider port
14. Registry returns port of Content Provider 1 and Content Provider 2
15. Supposing semantic information is available that indicates the user is inside the building, Content Provider 2 providing corresponding content will be prioritized and requested
16. Content Provider 2 returns content data, e.g., a map that provides location-based guidance inside the building

As the sequence chart indicates, adaptivity results from context-sensitive service composition. Thereby, the messaging behavior of each subservice remains independent of context changes. This is possible because ports of the same port type can be interchangeably replaced without interfering with the ports' WSDL-prescribed request/response-behavior.

Traditional monolithic LBS typically do not provide this degree of context adaptivity (here, to switch to WLAN positioning in case the GPS becomes unavailable) without being explicitly designed for every possible change of interoperation. Furthermore, they hardly adapt to emerging technologies that were not foreseeable at design time. In contrast – provided that messaging behavior of new services remains compatible with the given type definition – ALBS can adapt to changing or newly emerging conditions without extra programming effort.

4.3 Using ALBS in Mobile Environments

The fundamental concept of the service-oriented paradigm is to enable uniform access to all resources via services – including mobile or embedded devices, and hardware resources, for example, GPS receivers or WLAN adapters. In dynamic environments, where network topology, connections, and bandwidth are unstable and connected devices may have limited resource power, this requires specific methods for service propagation, discovery, invocation, and processing:

- **WS-Discovery:** The Web Services Dynamic Discovery (WS-Discovery) standard defines a multicast protocol to propagate and locate services on ad-hoc networks in peer-to-peer manner. It supports announcement of both service offers *and* service requests. Efficient algorithms (caching, multicast listening, discovery proxies, message forwarding, filtering, scope adjustment, and multicast suppression) keep network traffic for announcing and probing manageable. Thus, the protocol scales to a large number of endpoints.
- **Lightweight Services:** For efficient invocation and processing of Web Services on embedded devices with limited processing and communication power,

"lightweight" services utilize specific real-time protocols, programming languages, scheduling algorithms, message queuing policies, or XML coding and parsing schemes.

In our example, the mobile device multicasts its request to the devices within local reach and collects the service announcements. The GPS receiver announces a service for position sensing and the WLAN adapter announces two services, one for position sensing and one for connection (see Fig.3). These ports are stored in the local registry cache. Retrieved entries from the global registry are cached as well. To locate a service, the discovery service is instructed to retrieve the corresponding port type. Additionally, the discovery service can look for certain assertions to be satisfied. Thus, the ALBS application communicates with local services the same way it does with remote services. All services are propagated, discovered, and invoked by standard Web Services protocols.

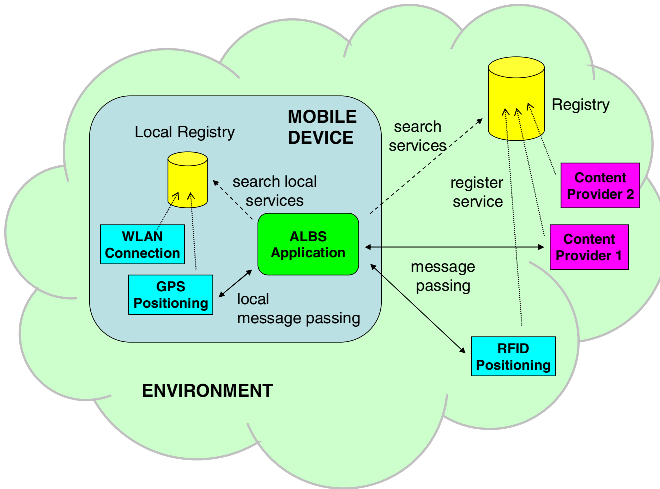


Fig. 5. The ALBS architecture allows for location-aware computing based on universal service-oriented communication

4.4 Semantic Location Information

For semantic interpretation we distinguish the following LBS classes: Location-based services can be provided by some immobile unit, e.g., a museum or a botanical garden. Typically such immobile units provide *stationary LBS* which are fixed to a given location. A common problem is to semantically detect the location and find or filter stationary services related to that location. For example, a user's movement in a museum can tell that he might be interested in information about a specific exhibition object (e.g., he moves to that object and then, while looking at it, stops moving for some seconds). A location-aware device then could request the assigned service.

Likewise, some immobile units may provide *general LBS* that are location-independently accessible but require a location parameter. Examples are a regional weather forecasting service or a service that processes queries like “where is the next subway station?” Regarding *mobile LBS*, the location is a parameter of the behavior of a mobile device. Imagine a user who travels with his laptop. If the laptop recognizes the availability of a specific LAN connection, it could conclude where it is located (e.g., in the user’s office) and adapt its behavior (e.g., synchronize certain files). Finally, *interdependent LBS* require multiple related location parameters, e.g., a people finding service that guides mobile users to meet at some intermediate place. All these cases demand for appropriate semantic interpretation of location.

Let us consider the following example of a mobile LBS in more detail: A user wants his mobile phone to automatically activate the hands-free speaking system inside a car or mute when inside a theatre. However, a cellular phone cannot tell from GPS coordinates or from its cell ID that it is inside a theatre. But if there is a service that translates the GPS coordinates or the cell ID to semantic location information like “this is a theatre” or “this is a place to be silent”, the “mute feature” can be automated. The Resource Description Framework (RDF) addresses these issues and may be used to accomplish such communication.

For example, an extended position sensing service may return the semantic location “prater.theatres.berlin.de”. To know how to act on this location information, a service assigned to it might return the following RDF message indicating that mobile phones and other possibly “obtrusive” devices should be switched off. Since the theatre is an “ambient” place in the following ontology, the device can understand that it should mute:

```
<rdf:Description about="urn://prater.theatres.berlin.de">
<rdf:type resource="urn://myontology.myID.de/Schema/theatre"/>
<rdf:type resource="urn://myontology.myID.de/Schema/places/ambient"/>
<t:Name>Prater</t:Name>
<t:DesiredCellPhoneActivity>Silent</t:DesiredCellPhoneActivity>
</rdf:Description>
```

Unfortunately, there are various ways to express such additional semantic location information. The device, therefore, not only must be able to access the ontology that is applicable, it moreover needs to know how to map this ontology to its decision alternatives. However, to the best of our knowledge, comprehensive ontologies widely accepted and suitable for broad semantic location processing are not available as yet.

5 Case Study

Currently, we are working on an adaptive location-based service prototype. It provides the basic functionality of a mobile information system for the “WISTA Adlershof Science and Technology Park” [21] in Berlin, where natural science departments of the Humboldt University are located as well. Based on the Emerging Technologies Toolkit (ETTK) available from IBM developerWorks, we provide part of the ALBS infrastructure. We are defining some basic port types (Content and Position Sensing) allowing all companies from the WISTA area to provide further

location-based information and services. Additionally, a web crawler will scan the regional websites for location information and, if location can be determined, assigns the website to that location. Geographic information and a geo-referenced map are supplied by the Geographic Institute of Humboldt University, which takes part in this case study. A widely available WLAN-infrastructure gives mobile devices access to remote services and allows the setup of WLAN-based position sensing services.

Location-based service will also be available for "virtual travelers". They explore the WISTA map on the Internet that visualizes location-specific information and stationary services. By point-and-click, it is possible to directly access these stationary LBS or to forward the specific position to some general LBS. That way LBS link virtual to physical spaces. For example, if a user is visiting the Internet site of a company, the company's physical location will be determined and can serve as input for subsequent LBS. Vice versa there are LBS that link from physical to virtual spaces, e.g., one that processes instructions such as "show me the website of the restaurants located within 5 minutes walk". In future, a position sensing service can as well determine the semantic position within the virtual space. For example, if position sensing detects that the user is visiting some product information site, it can take him to product-related offers, e.g., test reports or best price comparisons.

6 Outlook and Conclusions

We have shown how to flexibly compose Adaptive Location-Based Services (ALBS) with Web Services technology achieving high service availability. Further, we outlined how location information can be processed semantically. We anticipate that the methodology will be applicable to future context-aware computing in distributed, heterogeneous, and dynamic environments at great degree of interoperability – across various protocols, interfaces, programming languages, devices, connection lines, operation systems, platforms, enterprise boundaries, vendors, and service providers.

We also expect that broad interconnection and seamless interoperability of processes and devices together with tight interconnection of physical and digital spaces will bring convergence of virtuality and reality, changing the way we think, work, and live. As public dependence on information systems will continue to rise and there will be insufficient human resources to continually support and maintain the computing/communication infrastructure, the vision of adaptive, maintenance-free, self-relying systems must become a reality. The goal will be to deliver MAD properties (Mobility, Adaptivity, and Dependability) at low cost and in intelligent and highly semantic manner, making it possible to enter the age of "subdued computing", where the information infrastructure at best supports human computing and communication needs, but discreetly fades into the background.

References

1. Auto-ID Center: Creating an Internet of Things. www.epcglobalinc.org (accessed April 2004).
2. BPEL4WS, Business Process Execution Language for Web Services, Version 1.1, www-106.ibm.com/developerworks/library/ws-bpel

3. Buyya, R.: Economic-based Distributed Resource Management and Scheduling for Grid Computing. Ph.D Thesis, Monash University, Melbourne, Australia, April 2002
4. European Symposium on Ambient Intelligence (EUSAI), www.eusai.net (accessed April 2004)
5. Hazas, M., Scott, J., Krumm, J.: Location-Aware Computing Comes of Age. IEEE Computer, 37(2):95–97, Feb 2004
6. Hawaii International Conference on System Sciences, Value Webs in the Digital Economy Mini-Track, Hawaii, Jan 2005, www.value-webs.org
7. Jennings N., Sycara, K. et al.: A Roadmap of Research and Development. Autonomous Agents and Multi-Agent Systems, 1(1):7–38, Jan 1998
8. Malek, M.: The NOMADS Republic. International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine and Mobile Technologies on the Internet, Scuola Superiore G. Reiss Romoli (SSGRR), Telecom Italia, L'Aquila, Italy, 2003
9. Kephart, J., Chess, D.: The Vision of Autonomic Computing. IEEE Computer, 36(1): 41-50, Jan 2003
10. Meissen, U., Pfennigschmidt, St., Voisard, A., Wahnfried, T.: Context- and Situation-Awareness in Information Logistics. International Conference on Extending Database Technology (EDBT), Workshop on Pervasive Information Management, Heraklion, Greece, March 2004
11. Milanovic, N., Richling, J., Malek, M.: Lightweight Services for Embedded Systems. IEEE Workshop on Software Technologies for Embedded and Ubiquitous Computing Systems (WSTFEUS), Vienna, Austria, 2004
12. Müller, P., Stich, Ch., Zeidler, Ch.: Components @ work: Component Technology for Embedded Systems. 27th Euromicro Conference, Warsaw, Poland, Sep 2001.
13. Papazoglou, M.: Agent-oriented Technology in Support of E-Business: Enabling the Development of "Intelligent" Business Agents for Adaptive, Reusable Software. Communications of the ACM, 44(4):71–77, April 2001
14. Rao, B., Minakakis, L.: Evolution of Mobile Location-based Services. Communications of the ACM, 46(12):61-65, Dec 2003
15. Reichenbacher, T. : Mobile Cartography – Adaptive Visualisation of Geographic Information on Mobile Devices. Dissertation submitted at the Institute of Photogrammetry und Cartography, Technical University, Munich, 2004
16. Schwan, K., Poellabauer, Ch., Eisenhauer, G., Pande, S., Pu, C.: Infofabric: Adaptive Services in Distributed Embedded Systems. IEEE Workshop on Large Scale Real-Time and Embedded Systems (in conjunction with RTSS 2002), Austin, TX, Dec 2002
17. Sutherland, J., van den Heuvel, W.-J.: Enterprise Application Integration and Complex Adaptive Systems. Communications of the ACM, 45(10):59–64, Oct 2002
18. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 2003
19. UDDI, Universal Description, Discovery and Integration of Web Services,www.uddi.org
20. Weiser, M.: Some Computer Science Problems in Ubiquitous Computing. Communications of the ACM, 36(7):74-83, July 1993
21. WISTA Adlershof Science and Technology Park, www.wista.de

On Enhancing the Robustness of Commercial Operating Systems*

Andréas Johansson, Adina Sârbu, Arshad Jhumka, and Neeraj Suri

Department of Computer Science,
Technische Universität Darmstadt, Germany
{aja, adina, arshad, suri}@informatik.tu-darmstadt.de

Abstract. A ubiquitous computing system derives its operations from the collective interactions of its constituent components. Consequently, a *robust* ubiquitous system entails that the discrete components must be robust to handle errors arising in themselves and over interactions with other system components. This paper conceptually outlines a profiling framework that assists in finding weaknesses in one of the fundamental building blocks of computer based systems, namely the Operating System (OS). The framework allows a system designer to ascertain possible error propagation paths, from drivers through the OS to applications. This significantly helps enhance the OS (or driver/application) with selective robustness hardening capabilities, i.e., robustness wrappers.

1 Introductions: The Ubiquitous Computing Perspective

A ubiquitous computing (UC) environment harnesses the collective capabilities of diverse computational components via dynamic resource management as warranted in mobile, networked and heterogeneous system environments. The utility of such UC systems arises only if adequate robustness in the UC infrastructure exists for it to provide for dependable service delivery. It is evident that achieving an acceptable level of trust in such consolidated systems also necessitates corresponding design methods for evaluating (and forecasting) how perturbations in the system affect the services provided. Such perturbations could arise as device defects, in UC component interactions, bugs in software etc. For UC components themselves, the aspects of heterogeneity and mobility translate to the problem of not knowing at design time the precise characterization of their operational environment(s). The problem is further complicated in that many of the devices are too deeply embedded to be modifiable. However, during deployment, (robustness) profiling of the system is viable. This will give rise to suggested locations of robustness gaps (hence, of enhancements) within the system. However, owing to the dynamic nature of UC, profiling the overall system is not viable. Thus,

* We appreciate the inspration & insights from Dr Martin Hiller and the funding support of Microsoft Research through the Innovation Excellence Program.

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 148–159, 2005.
© Springer-Verlag Berlin Heidelberg 2005

we propose to perform a two-level profiling: (i) profiling the platform, including components that will be long-term present in the system (middleware, OS, drivers, HW etc) (ii) profiling the applications, which continually change over the operation of the system. The problems with equipping the system with added capabilities during run-time or deployment include (a) not knowing which errors to protect against, (b) not knowing where to add enhancements in the system, and (c) how to design effective and efficient wrappers.

1.1 The Role of OS: Paper Objectives

Targeting the OS as a key building block in any UC system, OS robustness hardening is a fundamental driver for providing robustness of the systems built upon them. The OS manages the hardware resources available and acts as a supplier of services for applications through programming interfaces. A potential problem for any OS is that it must be able to handle diverse hardware resources and applications. The risk of robustness gaps in the OS is apparent and the need for robustness enhancements exists. Naturally, adding robustness enhancements comes as a cost trade-off with other system properties, e.g., performance, determinism etc. A trade-off analysis needs span the need for enhancements (wrappers) and their consequent properties (coverage, timing etc).

As the overall goal is to prevent application failure, the interactions across application and OS are key. Owing to the dynamic nature of UC, the information needs to be categorized into two parts: (i) an OS profile, and (ii) an application profile. These profiles will aid in identifying vulnerabilities in the system, thereby guiding the effective placement of wrappers in the system. Thus, our research objectives, to facilitate development of a systematic robustness process, i.e., placement and compositions of wrappers in an OS span the following themes:

- A profile of possible OS vulnerabilities based on system error propagation.
- Application profiles, including (a) criticality of the applications, (b) the applications use of the OS, and (c) sensitiveness to robustness gaps in the OS.
- Error detection mechanisms ranging in type, size and complexity depending on their placement. Estimations of the properties of the detectors (e.g., completeness, accuracy, overhead in performance, size, cost).
- Error correction mechanisms following error detection; ranging from halting the system to performing advanced recovery.
- Assessment and tradeoffs - effectiveness & cost - analysis framework.

This paper focuses on the first two problems, i.e., to develop systematic methods for OS profiling. Profiles provide information on where errors in the OS are more likely to appear and cause damage. The purpose is to determine robustness gaps in the OS that can be used to guide locating and constructing effective robustness wrappers. We discuss the relevant profiling process and their experimental assessment. The result of this profiling will invariably depend on the errors considered. We specifically intend to address errors occurring in OS drivers and their impact on services the OS provides for applications.

1.2 Robustness Hardening: Related Work

Software profiling represents the process of assessing the data flow properties within the software structures. A common type of profiling is determining where the execution time of a program is spent. Such profiling can locate performance bottlenecks and even design errors in a program. Other profiles might tell the programmer which functions in the OS are used and the relative time spent performing them, e.g., the `strace` utility for UNIX-like systems, which profiles a program's use of the OS by showing the system calls made and the signals used.

In our EPIC framework (Exposure, Permeability, Impact and Criticality)[9] static modular SW (fixed set of modules that interact in a predefined manner), was profiled for error flow to ascertain the most effective placement of wrappers. The framework (and the supporting experimental tool PROPANE [8, 7]) focused on profiling the signals used in the interaction between modules. The profiling consisted of both error propagation and effect profiles. Using the permeability, and exposure metrics, propagation profiles reveal information on where errors propagate through the system and which modules/signals are more exposed to propagating errors. The error effect profiles are used to cover the cases where an error is unlikely to occur, but potentially has a high impact on system output. The software model for EPIC is static software; this makes it possible to find all communication paths in the system and then profile accordingly. In our OS themed work, the emphasis is on dynamic SW interactions. Also the set of applications is not generally known, all possible interaction paths (and consequently error propagation paths) are not known *a priori*.

In [5] errors in C-libraries were studied. A tool called HEALERS was developed which automatically tests (using fault injection) the library functions against their specification and generates wrappers to stop non-robust calls to be made. This approach differs from our profiling strategy since it only focuses on robustness in library functions. We are more interested in how errors can propagate in the system, thus allowing us to choose more than one possible location for wrappers (OS-application or OS-driver interface). However, the type of errors proposed in HEALERS are similar to the ones considered in our work.

A related approach is used in Ballista [4, 3]. Here, OS interfaces are tested with a combination of correct and incorrect parameter values and the behavior of the OS categorized according to a failure scale, ranging from "OS crash" to "no observation". Using this relatively simple approach, this method managed to find a number of robustness flaws (crashed or hung OS's) in both commercial and Open Source OS's.

Both HEALERS and Ballista are different from our proposed approach as they study the effect of malfunctioning applications on the OS whereas our method considers the effect of malfunctioning drivers on the OS. Both of these areas are important and they should be seen as complementary perspectives.

A fault injection based approach was used in [1], where the behavior of microkernel based OS's in presence of faults was studied. Faults were injected in both the memory of the kernels and in parameter values in calls to kernel primitives by applications. The effects on applications were studied along with error prop-

agation between the kernel modules. This study differs from ours as we consider a constrained model of the OS, where internal communication details are not accessible. Also the errors considered are different. We focus on errors occurring in drivers and not in applications or internal kernel errors.

Nooks [15, 14], focuses on errors in the driver subsystem considering that drivers are a primary source of OS failures. A new subsystem layer is defined in which kernel extensions (like device drivers) can be isolated in protection domains. A driver executing in a protection domain has limited possibilities of corrupting kernel address space and objects. Nooks provides good coverage for many software and hardware faults occurring in extensions at the price of high execution overhead (up to 60% slowdown for a web server benchmark). Nooks focuses on isolating drivers from all errors impacting the OS. However, we focus on data level errors and find out which specific errors actually propagate and then protect against these. Nooks also requires a white box approach unlike our black box model of SW. Both methods are useful for malfunctioning drivers but they have different properties such as coverage and performance.

2 System and Error Model

Our intent is to develop a profiling framework applicable to diverse OS's. Thus, we utilize a generic model of a computer system, comprising of four major layers; hardware, drivers, OS and applications, see Figure 1. Each layer supplies a set of services to the next layer. For us, a service is typically a function call, i.e., for drivers it is an entry point in the driver and for the OS it is the system call or library functions. We say that a set of services makes up an interface, for instance a driver interface.

We do not assume any specific hardware architecture (CPU, disks, peripheral devices etc.), as long as the hardware layer contains all devices needed to support the remaining layers of the system.

The driver layer is responsible for handling the interaction between the hardware and the OS. Drivers are SW programs that are generally specific to a certain piece of hardware and OS. The system has a set of drivers $\mathbb{D}$, denoted $d_1, d_2, \cdots, d_D$. Each driver provides a set of services to the OS.

In the OS layer, we also include shared libraries that exist in the system. We include them in the OS layer as it reflects the point of view of a programmer, i.e., the libraries support functionalities that the programs need, very much the same as the OS. The OS provides a set of services (OS services), $\mathbb{S} = \{s_i : i \in [1, S]\}$, to be used by the applications.

The top layer is the application layer, where the programs execute performing some specific task for a user. Applications make use of OS resources either directly or through the use of libraries. The services provided in $\mathbb{S}$ may selectively be used by a certain application, i.e., each application uses a subset of $\mathbb{S}$. Each application, APP_k, uses M_k of the interfaces provided by the OS. Thus we define the set of interfaces that APP_k uses, as: $\mathbb{P}_k = \{AS_i \in \mathbb{S} \text{ used by } APP_k\}$.

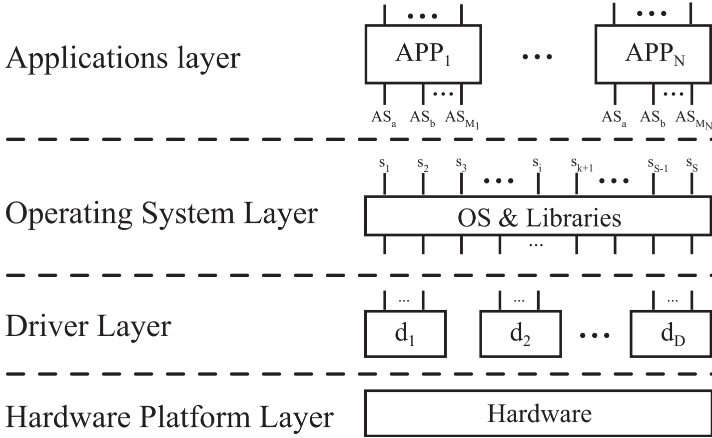


Fig. 1. General System Model

In this work we consider a black box SW model, i.e., we assume no knowledge about the components in the respective layers but only their interface specifications. They are supplied in binary form (for SW) and can be executed on a given hardware platform. We do not make any assumption constraining the behavior of any component when it is subjected to errors/stresses. Such stress conditions could be heavy load on a component, misuse of its interface or resource depletion.

2.1 Error Model

Focusing on a black-box model of the system, the only place where monitoring/wrapping can take place is at the interfaces between the layers in the system. We define two major interfaces, between the OS and the drivers and the OS and applications. We consider data level errors occurring in the OS-driver interface as our primary error model. Recent studies have demonstrated that drivers are a major source of OS failures [2, 15, 12]. This arises as they are often not tested as rigourously as the OS kernel; they may have been designed external to the OS development team, lacking complete details of the system; and they may also be affected by malfunctioning hardware. Drivers represent a large part of an OS package in sheer size which further warrants their focused consideration (as the number of expected faults generally increases as the size increases). We focus on data level errors as they are possible to use with a black box system. They also represent a set of detectable errors, i.e., one can define detectors in the form of assertions to detect them.

In other robustness studies different fault/error models where used, mostly bit-flips in parameter values and/or data and code memory areas of programs [7, 1, 6]. Instead, we focus on data level faults at interface levels as they best represent actual SW errors.

A data level error implies that the value of the data communicated through the interface is “erroneous”, i.e., not the expected value according to some om-

niscient viewer. Thus, the "state" of the program changes, and this error propagates to the OS causing subsequent errors. To simulate data errors, we specifically change the values of parameters used in the targeted interface. The type of error injected is decided by the type of the parameter used. It is important to note that the chosen error model and its representativeness of actual OS errors, fundamentally affects the relevance of any obtained results.

3 Measures of Error Propagation

The purpose of a profiling framework is to outline SW characteristics for a designer. The desired characteristic also drives the method used to gather data. Our focus is on effective location of OS wrappers to handle driver errors. Thus, we target determining the specific drivers that have a higher likelihood of error propagation. Similarly, on error occurrence, to establish which OS services are more exposed to these errors.

The final goal of our profiling is to estimate the impacts errors have on the services provided by applications. One problem is that profiling with respect to a certain set of applications running on the system will not give the same result as a different set of applications. The way applications use the OS and their importance will influence the results of the profiling. To capture this distinction we propose to use a separate profile for an application's use of the OS. This profile must include information revealing which OS services the application depends upon and notions of each service importance to the application (some are most likely more important for the continued functioning of the application than others). To analyze a system with more than one application, with varying importance, priorities must be established across them. This can then be used when the assignment of wrappers is made.

3.1 Operating System Profiles

The OS profiles consider how errors in drivers spread through the OS to its services for applications. These profiles are naturally specific to an OS and its drivers. The first measure (*Service Error Permeability*) defines the probability of errors propagating through the OS. This measure is used to ascertain which OS services are more susceptible to propagating errors (*OS Service Error Exposure*) and which drivers are more likely to spread them (*Driver Error Diffusion*).

Service Error Permeability. The goal of OS profiling is to characterize how errors in drivers influence the OS services provided. We start by defining the service error permeability, $P^{j,k}$ for an OS service $s_j \in \mathbb{S}$ and a driver $d_k \in \mathbb{D}$.

$$P^{j,k} = Pr\left(\text{error in } s_j | \text{error in } d_k\right) \tag{1}$$

Eq. 1 describes the relation of one driver to one OS service. This measure gives an indication of the permeability of the particular OS service, i.e., how 'easily' does the service let errors in the driver propagate to applications using

it. A higher probability naturally means that either (a) the driver needs to be enhanced to make sure that it does not produce errors, or (b) some detection (and recovery) is needed in the application or elsewhere to handle errors propagating.

OS Service Error Exposure. The OS error permeability considers discrete drivers on a stand-alone basis. To ascertain which OS service is the most sensitive to errors propagating through the OS, then more than one driver needs to be considered. We use the measure OS error permeability, to compose the *OS Service Error Exposure* for an OS service s_j, namely E^j:

$$E^j = \sum^{\forall d_k \in \mathbb{D}} P^{j,k} \tag{2}$$

For E^j we consider the set of all drivers, $\mathbb{D}$. If one driver does not affect a service, for instance it is not at all used by that service, then the Service Error Permeability will be zero and thus should not affect the OS Service Error Exposure. The OS Service Error Exposure gives an ordering across OS services (given that more than one service has been tested and have service permeability values) which orders services based on their susceptibility to errors passing through the OS. With this measure the tester can focus attention to particular services that may require more work or as a means to place wrappers.

Driver Error Diffusion. One might not only be interested in finding out which services are more exposed to errors but also which driver is spreading them the most, i.e., which driver, if acting faulty, has the potential of spreading errors the most in the system. To find these drivers we define a measure that considers one particular driver's relation to many services, *Driver Error Diffusion*, D^k, for a driver $d_k \in \mathbb{D}$ set of services:

$$D^k = \sum^{\forall s_j \in \mathbb{S}} P^{j,k} \tag{3}$$

The Driver error diffusion also creates an ordering across the drivers. It ranks the drivers according to their potential for spreading errors in the system. Note that we do not try to test the drivers per se, so this measure only tells us which drivers **may** corrupt the system by spreading errors. It is actually a property of the OS and not the drivers.

3.2 Application Profile

To estimate the effect of errors in OS services, we need to establish the applications usage of these services, specifically (a) which services are invoked and (b) their respective "importance". The *importance* of the service is defined with respect to how critical it is to the continued functioning of the application. If an error in an OS service always causes the application to crash, we say that this OS service is important to the application. On the other hand, if it does not have any effect on the application, because the application has built-in error correction,

it is of no importance. The importance $i_j, 0 \leq i_j \leq 1$ of an OS service $s_j \in \mathbb{P}_k$ is defined as the probability that given an error in an OS service, it causes the application APP_k to fail. $\mathbb{A}_k$ is a set of tuples, (s_i, i_i). For each OS interface $s_i \in \mathbb{P}_k$ used by the application there is one and only one such tuple. Each tuple includes the service itself and the value i_i which describes the importance of the service to the application. The application profile can and should be constructed before the system is deployed, i.e., not during run-time.

$$\mathbb{A}_k = \{(s_i, i_i) : s_i \in \mathbb{P}_k\} \tag{4}$$

A third parameter of interest is the *criticality* of the application. If there is more than one application running on the system, some of them might be more important than others. We then say that it has a higher degree of *criticality*. For comparing profiles across applications we need to know if adding wrappers to "help" a certain application is more important than another application. In EPIC [9] a scale from 0 to 1 was used for weighing the output signals according to their criticality. 1 indicated the highest possible criticality and a 0 indicated that the output signal is non-critical. A similar scale is used here too, indicating a range from non-critical to highly critical. A criticality value, C is thus assigned to every application APP_k, $0 \leq C_k \leq 1$.

3.3 Application Service Exposure

The Service Error Exposure and the Application profile together determine how one application is affected by errors occurring in drivers. This can be used to establish wrapper locations, given a set of drivers and applications. Potentially this trade-off can be made online (if applications are shipped profiles) as applications comes (and leaves) the system.

$$SE_k = C_k \cdot \sum^{\forall s_i \in \mathbb{A}_k} \left(E^i \cdot i_i\right) \tag{5}$$

Eq. 2 & 4 aid in predicting how an application will react to errors propagating in the system. The *service exposure* composes the error permeability profiles with the application profiles, to predict the behavior of the application. For an application APP_k, given the profile $\mathbb{A}_k$ and the corresponding service propagation values. The values of the application error exposures are used to create a relative ranking of applications and also the wrapping priority.

4 Experimental Evaluations

To make use of the analytical framework presented in the previous section, values for the profiles must be obtained. Code inspection analyses, expert analyses, bug reports/error logs and fault injection (FI) experiments are all possible approaches in this respect. We have chosen FI as its utility has been established (and also used in the EPIC framework using the PROPANE tool [8, 7]); it is also usable

at design time (in contrast to bug reports and error logs) and it can be used without the full source code availability.

As indicated in Sec. 3 the only measure that needs to be experimentally estimated is OS Error Permeability, $P^{j,k}$, as both OS Service Error Exposure and Driver Error Diffusion can be derived from this measure. To get an experimental estimation of $P^{j,k}$ we will inject faults in the interface between driver d_k and the OS and monitor the result of executions of service s_j by writing an application that uses this service. We estimate the error permeability as the ratio of detected errors, $n_{detected}$ at the service level to the number of injected errors $n_{injected}$.

$$P_{est}^{i,j} = \frac{n_{detected}}{n_{injected}} \tag{6}$$

The values of E_{est}^{j} and D_{est}^{j} can be calculated using $P_{est}^{i,j}$ using Eq. 2 & 3. Naturally not all application cause usuage of all drivers, or all of their services, so the FI experiments will be limited to the driver services actually used. Eq. 6 requires detecting that an error has actually propagated. There are several types of outcomes from a FI experiments and they can be classified in different classes. We use a failure mode scale similar (but not the same) to the one used in [3] as:

- Error propagated, but still satisfied the specification of the service
- Error propagated but violated the specification of the service
- The OS crash/hung due to the error

For the first class it must be clear what is meant by a specification. In this work we consider the specification given to a programmer for the OS. This can be for instance `man` files for Linux/UNIX or the help sections for a Windows computer. An example of an outcome that will end up in this class is (a) when an error code or exception is returned that is a member of the set of allowed codes for this call, or (b) if a data value was corrupted and propagated to the service, but did not violate the specification.

The second class contains those outcomes where the result is violating the specification of the service. For instance returning a string of certain length when another parameter specifying the length holds a different number. Raising an unspecified exception or returning an unspecified error code also ends up in this class. If the application hangs or crashes but other applications in the system remain unharmed, they end up in this class as well.

If the OS hangs or crashes, no service progresses. This state can be detected by an outside monitor. The difference between the classes is that the crash usually produces some form of dump, indicating that the OS has crashed and some additional information. Currently we do not separate these classes, but a designer may choose to do so.

Apart from the failure modes defined above, one more outcome is possible, namely *No Failure*, i.e., no failure is observed at the OS service level. The error might be dormant within the system, but in order to limit the time to perform the experiments a timeout will be defined after which the error is considered not to have propagated.

4.1 Inserting Probes

Sec. 3 outlines the need to facilitate monitoring between the system layers, see Fig 1. Thus, we need to insert probes between the application and OS layers and between the OS and drivers layers. These probes are needed to (a) monitor the communication between the layers in the system and (b) for actually inserting perturbations in this communication, i.e., simulating the occurrence of errors. The first case, can be achieved simply by writing special purpose applications which sole purpose is to actively use the services provided by the OS. To achieve the latter monitoring we design a new driver that is loaded instead of the driver which we want to monitor/wrap. The needed "wrapping driver" is to first load the existing driver and then set up all data structures. Then it passes on any calls from the OS to the real driver and the result back to the OS. None of the methods interfere with our intent to use black-box methods since it does not involve modifications of the OS and/or the existing applications or drivers. Some reconfiguration may be needed but access to source code is not required.

4.2 Estimating Application Profile

To obtain the application service exposure presented in Sec. 3.3 (using Eq. 2 and 5) we need to experimentally derive the application profile $\mathbb{A}_k$ and the error permeability values for each driver $P^{j,k}$. The application profile is derived using a profiling tool that executes the application and produces a list of calls made to the OS. The calls are matched against $\mathbb{S}$ to produce $\mathbb{A}_k$. For each interface that one application uses, an importance value must be assigned. The importance value i_i can be derived using a tool that injects faults in the output of system calls made by the application. Calls are intercepted by additional wrapper layer, the actual call is made to the OS and the return value can then be altered, i.e., an error can be inserted and the result is passed on to the OS. A similar approach was used for the Fuzz tool where UNIX utilities were fed with random input strings and their robustness depended on their response to these streams [11]. Another similar approach was used for Windows NT applications in [13]. The results from a similar tool can be used to assign the importance values.

Finally the criticality value must be assigned. This value represents the criticality of an application relative to other applications. It is used to bias the application service profile towards the applications that are of higher importance. This value is assigned by the profiler and it depends on the set of applications present. C_k is a value between 0 and 1, where 1 indicates high criticality.

5 Discussion and Future Work

As noted in Sec. 3, the profiles presented will only provide a relative ordering across OS services, drivers and applications. To get real values for the probabilities of real errors occurring we need to know precise nature and the arrival rate of real errors. However, our method is targeted at placement of wrappers.

It illustrates to the profiler which services are more exposed to errors and thus require wrapping. Note that this is always an estimation based on the probabilistic nature of testing. Additional knowledge about the system can be used to complement the profiling. For instance, if empirical knowledge exists about the "quality" of different drivers, this can be used in the profiling to exclude certain drivers from the profile or weigh their scores so that their impact is reduced.

To be able to decide, with the help of the profiles, if more resources should be spent within the system on adding wrappers, some notion of robustness level is needed. We need methods for determining when enough wrappers have been added to the system. Possible heuristics can entail removing all potential crashes of the system and then one by one removing the less severe errors until a requisite number have been removed incrementally.

Another issue that may impact the location of wrappers is the wrappers themselves. Each wrapper comes with an associated cost, in size and overhead. The cost-efficiency trade-off is part of our intended future work, as well as the actual design and evaluation of wrappers. For evaluating wrappers, important parameters include the completeness and the accuracy, i.e., how many of the real errors are detected and how many mistakes are made. Also, the more complex a wrapper is, the higher its execution cost is. A coverage versus performance trade-off is an essential consideration.

A desired property of wrappers is to be able to generate them during run-time. When an application is loaded for execution, its associated profile is composed with the OS profile to find out if the current placement of wrappers in the system is optimal. We therefore need both the means to do the trade-off during run-time as well as the means to instantiate and deactivate wrappers without interfering with the functioning of the rest of the system.

A limitation of our approach is that dependencies between the application and the OS might exist that we currently do no cover. For instance we do not consider the order in which calls are made by an application to the OS. The ordering of calls, with respect to errors, might have an impact on the resulting behavior of the application. Investigating such dependencies and their implications for wrapper placement is part of future extensions for our work.

When determining if an error propagated, one needs to know what constitutes an error for a service. This information can (ideally) be derived from the specification of the service. This is not always possible due to the lack of or incompleteness of the specifications given. In the past, many fault injection experiments have utilized a so called golden run, i.e., a test program is executed without faults and the outcome is then compared with one run with faults presents to find deviations, i.e., errors. For OS's creating a golden run is non-trivial given the non-determinism in scheduling etc. One option would be to restart the system before every experiment and then conduct the tests (golden runs as well as FI experiments). Another option would be to run several tests without faults and use a mean behavior as the golden run.

6 Summary

Overall, the proposed profiling framework described in this paper aids a designer to effectively enhance the OS with error protection wrappers. By studying how errors propagate from drivers through the OS, potentially exposed OS services can be identified. Combining this with knowledge on the dependencies between the OS and an application, a suggested placement of wrappers can be obtained.

References

1. J. Arlat et al. Dependability of COTS Microkernel-based Systems. *IEEE Trans. on Computers*, 51(2):138–163, 2002.
2. A. Chou et al. An Empirical Study of Operating System Errors. In *Symposium on Operating Systems Principles*, pp. 73–88, 2001.
3. J. DeVale and P. Koopman. Performance Evaluation of Exception Handling in I/O Libraries. In *Proc. DSN*, pp. 519–524, 2001
4. J. DeVale and P. Koopman. Robust Software - No More Excuses. In *Proc. DSN*, pp. 145–154, 2002
5. C. Fetzer and Z. Xiao. An Automated Approach to Increasing the Robustness of C Libraries. In *Proc. DSN*, pp. 155–164, 2002
6. W. Gu et al. Characterization of Linux Kernel Behavior Under Errors. In *Proc. DSN*, pp. 459–468 , 2003
7. M. Hiller, A. Jhumka, and N. Suri. PROPANE: An Environment for Examining the Propagation of Errors in Software. In *Proc. of ISSTA*, pp. 81–85, 2002
8. M. Hiller, A. Jhumka, and N. Suri. An Approach for Analysing the Propagation of Data Errors in Software. In *Proc. DSN*, pp. 161–170, 2001
9. M. Hiller, A. Jhumka, and N. Suri. EPIC: Profiling the Propagation and Effect of Data Errors in Software. in *IEEE Trans. on Computers*, pp. 512-530, May 2004
10. C. Michael and R. Jones. On the Uniformity of Error Propagation in Software. In *Proc. of COMPASS*, pp. 68–76, 1997
11. B. Miller et al. An Empirical Study of the Reliability of Unix Utilities. *CACM* 33(12):32–44, 1990
12. B. Murphy and B. Levidow. Windows 2000 Dependability. In *Proc. of the Workshop on Dependable Networks and OS, DSN*, pp. D20–28, 2000
13. M. Schmid et al. Techniques for Evaluating the Robustness of Windows NT Software. In *Proc. of DARPA Information Survivability Conference & Exposition*, volume 2, pp. 1347–1360, 2000
14. M. Swift et al. Nooks: An Architecture for Reliable Device Drivers. In *Proc of the Tenth ACM SIGOPS European Workshop*, pp. 101–107, 2002
15. M. Swift et al. Improving the Reliability of Commodity Operating Systems. In *Proc of SOSP*, pp. 207–222, 2003

A Modular Approach for Model-Based Dependability Evaluation of a Class of Systems

Stefano Porcarelli[1], Felicita Di Giandomenico[1], Paolo Lollini[2], and Andrea Bondavalli[2]

[1] Italian National Research Council, ISTI Dept., via Moruzzi 1, I-56124, Italy
{porcarelli, digiandomenico}@isti.cnr.it
[2] University of Firenze, Dip. Sistemi e Informatica, via Lombroso 67/A, I-50134, Italy
{lollini, a.bondavalli}@dsi.unifi.it

Abstract. Modeling for dependability and performance evaluation has proven to be a useful and versatile approach in all the phases of the system life cycle. Indeed, a widely used approach in performability modeling is to describe the system by state space models (like Markov models). However, for large systems, the state space of the system model may result extremely large, making it very hard to solve. Taking advantage of the characteristics of a particular class of systems, this paper develops a methodology to construct an efficient, scalable and easily maintainable architectural model for such class, especially tailored to dependability analysis. Although limited to the class considered, the proposed methodology shows very attractive because of its ability to master complexity, both in the model design phase and, then, in its solution. A representative case study is also included.

1 Introduction

Analytical and simulative modeling for dependability and performance evaluation has proven to be a useful and versatile approach in all the phases of system life cycle. During design phase, models give an early validation of the concepts and architectural choices, allow comparing different solutions to highlight problems within the design and to select the most suitable one. During the operational life of the systems, models allow to detect dependability and performance bottlenecks and to suggest solutions to be adopted for future releases.

The model-based evaluation method has gained wide applicability and usage since a few decades. However, to keep pace with modern applications, modeling methodologies need to evolve towards more and more efficient solutions. In fact, although building models of simple mechanisms may be easy, the overall description of critical complex systems accounting at the same time for all their relevant aspects is not trivial at all: the information explosion problem remains the major difficulty for practical applications. To successfully deal with large

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 160–174, 2005.
© Springer-Verlag Berlin Heidelberg 2005

and complex models, the "divide and conquer" approach in which the solution of the entire model is constructed on the basis of the solutions of its individual sub-models [2] is followed. However, to properly cope with state explosion, it is not sufficient to resort to a modular and compositional approach at model design level, but this same approach has to be pursued at model solution level as well. Since this is clearly a pressing requirement, a number of research studies have started to tackle this problem in the last years, and some interesting approaches have appeared in the literature. [1] describes an approach for solving reliability models for systems with repairable components and introduces several approximations (effective in practice). [3] describes a general-purpose technique useful for an efficient analytic solution of a particular class of nonproduct-form models. Some other authors ([6], [10]) describe techniques based on generating the model of a modular system by composition of the submodels of its components. [7] proposes an efficient modeling approach and solution for Phase Mission Systems. As expected, given the high difficulty of the problem when approached in its most general terms, the existing proposals do not attempt general solution for the entire spectrum of systems; rather, they address specific system categories. Pursuing similar objectives, the goal of this paper is to present a novel methodology for dependability and performability evaluation tailored to a particular class of systems.

Specifically, we consider systems running applications composed of a set of functional tasks organized in such a way that the flow of computation moves along a hierarchy, without any dependency among components at the same hierarchical level, as better described in the next sections. Typical applications resembling such structure are control systems or resource management systems, where activities are carried on cyclically and consist of the following sequential tasks: monitoring of the system/subsystem behavior, elaboration of some computation to determine next actions and actuation of the selected actions. Taking advantage of the peculiar characteristics of such systems category, our modeling approach provides an efficient, scalable and easily maintainable architectural model that allows to better master complexity both in the design of the model and in its solution. To illustrate the methodology, its application to a case study is shown.

The rest of this paper is organized as follows. Section 2 describes the particular class of systems considered. Section 3 is devoted to the full description of the modeling approach. In Section 4 a case study is presented to illustrate a practical application of the methodology. Conclusions are drawn in Section 5.

2 Characteristics of the Considered Class of Systems

The class of systems our methodology focuses on is mainly characterized by a hierarchical structure of the system components and of the computation flow, as graphically depicted in Figure 1.

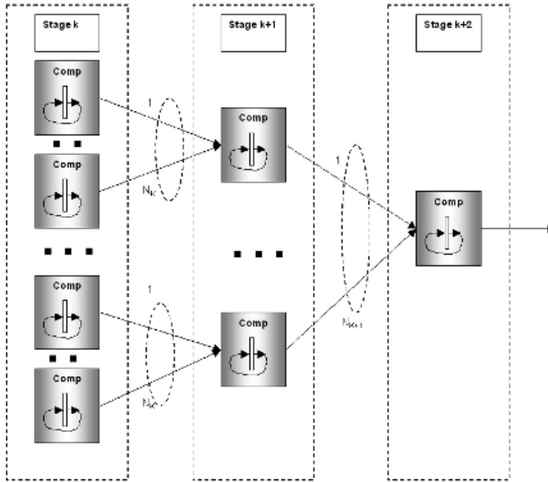


Fig. 1. Targeted class of systems

In more details, the main properties characterizing this class of systems are:

1. The system is composed of sets of hardware or software components (*COMP*) which can be logically grouped in "stages" (Stage 1, ..., k, $k+1$, $k+2$, ...);
2. A component at stage k (COMP^k) may interact only with those at stages $(k-1)$ and $(k+1)$ by means of message exchange and these interactions are unidirectional (e.g. from stage k to stage $(k+1)$). Therefore, there could be a functional dependency between two generic components COMP^k and COMP^{k+1}. A functional dependency between one component at stage k and more than one component at stage $(k+1)$ is not explicitly considered as it is equivalent to consider some (logical) replications of the component at stage k, each one interacting with only one component at stage $(k+1)$.
3. The interaction among components and the failure assumptions on each component are highlighted in Figure 2. This scheme is very general and must be specialized for the particular component under analysis. To explain the generic component's behavior, let's suppose it receives an input following a Poisson distribution with a rate λ^{IN}. These inputs are assumed to be correct or incorrect with a probability α and $1-\alpha$, respectively.

   In correspondence of inputs, which arrive with a rate λ^{IN}, the component produces an output with a rate $p * \lambda^{IN}$, where p is the probability a received input leads the component to produce an output. Moreover, the component is assumed to possibly behave incorrectly by self-generating spurious outputs with a rate λ^{S}. Thus, the "potential"[1] output rate of the component is expressed as $\lambda^{IN \to OUT} = \lambda^{IN} + \lambda^{S}$. From the point of view of

[1] Here, a "potential" output encompasses both emitted and omitted output ($p = 1$), while for "output" we refer only to those emitted.

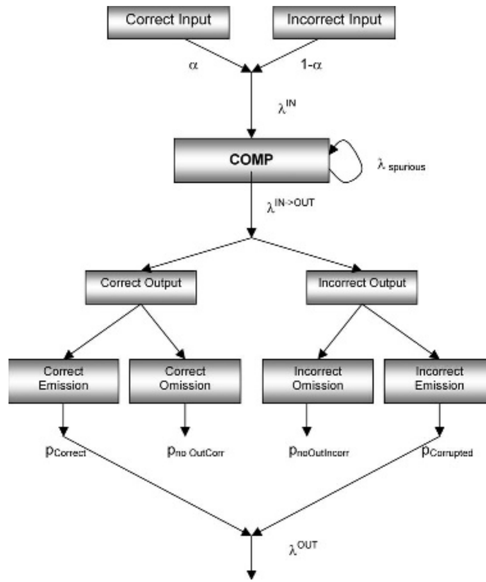


Fig. 2. How a Generic Component Interacts with Others

propagation, an output issued by COMP is propagated to another component with a rate $\lambda^{OUT} = (p_{Correct} + p_{Corrupted}) * \lambda^{IN \to OUT}$, where $p_{Correct}$ and $p_{Corrupted}$ represent the probabilities of generating a correct output (correct emission) and an incorrect output (incorrect emission), respectively. In general, a *correct emission* happens whenever a correct output is produced. A correct emission is possible i) in response to a correct input if the system is free from errors, or ii) in response to a correct input, if system errors are detected and tolerated. An *incorrect emission* happens either in reply to an incorrect input, or as consequence of a spurious output or of a wrong processing of a correct input. A *correct omission* may happen as consequence of an incorrect input or of an erroneous status of the system. An *incorrect omission* may happen as consequence of wrong processing of a correct input. These input-output combinations are summarized in Table 1. The input/output parameters characterizing each component are instead summarized in Table 2, where $p_{noOutCorr}$ and $p_{noOutIncorr}$ are the probabilities that the output is correctly omitted and incorrectly omitted, respectively.

As already discussed in the Introduction, although the presented system characterization does not cover each conceivable system, yet it is well suited for a restricted but well populated class of systems, like control systems and resource management systems. Given the behavior structure and failure semantics depicted in Figure 2, typical measures of interest from the dependability point of view in this context are:

Table 1. Input-output combinations

Input	Corresponding feasible output
Spurious output (internally generated)	Incorrect Emission
Correct input	Correct Emission, Incorrect Emission, Incorrect Omission
Incorrect input	Correct Omission, Incorrect Emission

Table 2. Input-output parameters for a component model

Input parameters	Output parameters
α, λ^{IN}	λ^{OUT}, $p_{Correct}$, $p_{noOutCorr}$, $p_{noOutIncorr}$, $p_{Corrupted}$

1. The probability of correct and incorrect emission;
2. The probability of correct and incorrect omission;
3. The overall probability that the system does not undertake wrong actions.

3 The Proposed Modeling Approach

In this section we describe our modeling approach, which fully exploits the above discussed characteristics of the reference class of systems. First, we deal with the model design process, that is, how to model a complex system starting from its functional specification and applying a stepwise refinement to decompose it in small sub–models. Then, the second part of the methodology is presented, which concerns the modular model solution, carried out in a bottom-up fashion. In fact, a methodological approach becomes attractive when it is not only directed to build models in a compositional way, but it also includes some capabilities to reduce their solution complexity. The philosophy of our modeling approach is shown in Figure 3.

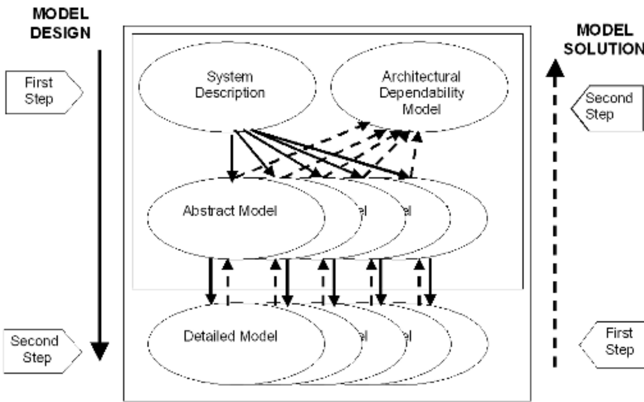


Fig. 3. Modeling approach

In order to construct an efficient, scalable and easily maintainable architectural model, we introduce a stepwise modeling refinement approach, both for the model design process and for the model solution. Another advantage of this approach is to allow models refinement as soon as system implementation details are known or/and need to be added or investigated.

3.1 The Model Design Process

The model design process adopts a top-down approach, moving from the entire system description to the definition of the detailed sub-models, while the model solution process follows a bottom-up approach. As inspired by [5], the system is firstly analyzed from a functional point of view (functional analysis), in order to identify its critical system functions with respect to the validation objectives. Each of these functions corresponds to a critical service provided by a component.

The overall system is then decomposed in subcomponents, each one performing a critical subfunction, and each subfunction is implemented using a model that describes its behavior. Therefore, starting from the high-level abstract model, we perform a decomposition in more elementary (but more detailed) sub–models, until the required level of detail is obtained.

The definition of the functional (*abstract*) model represents the first step of our modeling approach. The rules and the interfaces for merging them in the architectural dependability model are also identified in this phase. The second step consists in detailing each service in terms of its software and hardware components in a detailed (*structural*) model accounting for their behavior (with respect to the occurrence of faults). The fundamental property of a functional model is to take into account all the relationships among services: a service can depend directly from the state of another service or, indirectly, on the output generated from another service. The detailed model defines the structural dependencies (when existing) among the internal sub–components: the state of a sub–component can depend from the state (failed or healthy) of another sub–component.

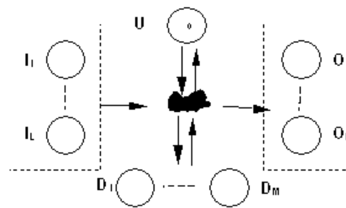


Fig. 4. Functional-level model related to a single service

Figure 4 shows the functional-level model related to a single service. The internal state S is here composed of the place U, representing the nominal state,

and of the places $D_1 \dots D_M$, representing different possible erroneous (degraded) states. The places $I_1 \dots I_L$ and $O_1 \dots O_N$ represent, respectively, the input (correct or exceptional, due to propagation of failures from interacting modules) and the output of the model (correct behavior or failure - distinguishing several failure modes). The state changes (from the nominal, correct state to the erroneous states and viceversa) and the flow between the input and output places are regulated by a structural model of the service implementation, indicated in Figure 4 as a black cloud.

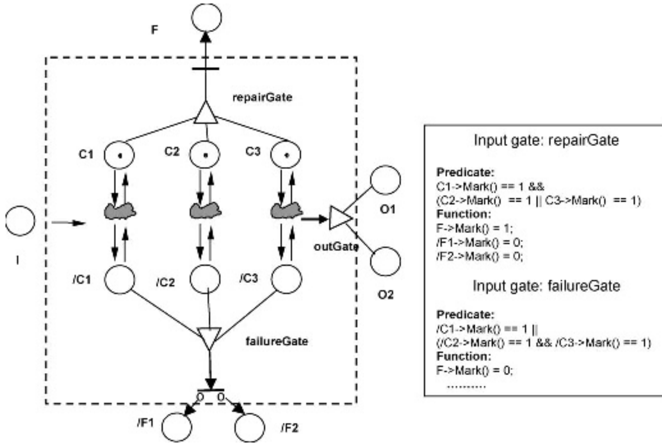


Fig. 5. An example of Detailed Model

An example of a structural model using SANs [11] is shown in Figure 5, where the service F is obtained through components C_1, C_2 and C_3. In turn, each component has the same structure of Figure 4, where the input/output relationships with other components are not considered. Notice that how these components concur to determine the state of the service (F, $/F_1$ or $/F_2$) is described in a simple way by means of the input gates *repairGate* and *failureGate* and the output gate *outGate* (in this example they are not fully specified because they are system dependent). C_2 and C_3 are in parallel and in series with C_1: F is in its nominal state if C_1 is up and at least one of C_2 and C_3 is up. The output gate *outGate* defines the relationship between input and output, given the internal status of the system.

3.2 The Model Solution Process

The model solution follows a bottom-up approach from the detailed model up to the abstract model. The implementation is strictly related to the environment characteristics of the system under analysis. Actually, starting from the general class of systems of Figure 1, we can derive several simplified systems that can be solved very efficiently.

Environment Characteristics. We denote with $\lambda_i^{OUT,\ COMP^k}$ the intensity of the output process of the i-th component of a group belonging to stage k (COMP_i^k). We make the following assumptions:

1. The distribution of the input process of each component is Poisson with rate λ^{IN}. This is accepted in the literature when the number of arrivals in a given time interval of time are independent of past arrivals.
2. The distribution of the output process of each component is Poisson distributed with a rate λ^{OUT}. This assumption corresponds, for example, to the case in which the inputs are processed sequentially without queuing and losses, and the processing time of the input is deterministic. Equivalently, we could obtain the same output distribution considering that the service time is Poisson distributed and that the component operates as a steady-state M/M/1 queuing network [12].

Using the assumption that the output process of COMP_i^k is Poisson distributed with rate $\lambda_i^{OUT,\ COMP^k}$, the superposition of N_k Poisson processes with intensities $\lambda_1^{OUT,\ COMP^k}, \ldots, \lambda_{N_k}^{OUT,\ COMP^k}$ is equivalent to a Poisson process with intensity equal to $\lambda_1^{OUT,\ COMP^k} + \ldots + \lambda_{N_k}^{OUT,\ COMP^k}$.

Solving the detailed model of components COMP_1^k, ..., $\text{COMP}_{N_k}^k$ leads to the evaluation of the probabilities of correct/incorrect output emission/omission and the intensity of the output process of a group of N_k components. Let's defining as $P_{Correct}^{k_i}$, and $P_{Corrupted}^{k_i}$ the probability of correct emission, and the probability of incorrect emission of COMP_i^k, respectively. Notice that these probabilities depend upon the intensity of the input process ($\lambda_i^{IN,\ COMP^k}$) and of spurious alarms ($\lambda_i^{S,\ COMP^k}$) (both supposed being Poisson). The following relations holds:

$$\Lambda^{OUT,\ COMP^k} = \sum_{i=1}^{N_k} \lambda_i^{OUT,\ COMP^k} \quad , \tag{1}$$

$$\alpha_{COMP^{k+1}} = \frac{1}{\Lambda^{OUT,\ COMP^k}} \sum_{i=1}^{N_k} \lambda_i^{OUT,\ COMP^k} \frac{P_{Correct}^{k_i}}{(P_{Correct}^{k_i} + P_{Corrupted}^{k_i})} \quad , \tag{2}$$

where $\Lambda^{OUT,\ COMP^k}$ is the intensity of the process achieved by aggregating the output processes of the components COMP_1^k, ..., $\text{COMP}_{N_k}^k$, while $\alpha_{COMP^{k+1}}$ is the probability that the next component at stage $k+1$ receives a correct input. Analogous considerations hold for COMP^{k+1}, and so on. This general approach can be specified for the following cases:

– If all groups of N_k components at stage k are identical, the total number of detailed models to be solved in order to evaluate the system of Figure 1 is equal to $\sum_{k=0}^{K} N_k$, where K is the number of "stages" in the system. It corresponds to evaluate the system of Figure 1 with its equivalent model of Figure 6 where $\Lambda^{OUT,\ COMP^k}$ and $\alpha_{COMP^{k+1}}$ are evaluated by means of equations (1) and (2), respectively. This way, the "tree" structure of the system of Figure 1 collapses in a unique "branch" from the point of view of system evaluation.

– If all groups of N_k components can not be considered identical at each stage, the number of models to be solved depends on the number of different "branches" in which the overall model can be simplified (see Figure 1).
– If for each stage k of the system, all the components are identical, it is possible to solve only K detailed models, one for each stage. Therefore, if all the components at level k are identical, than $\lambda_i^{OUT,\ COMP^k} = \lambda^{OUT,\ COMP^k}$, $P^{k_i}_{Correct} = P^k_{Correct}$, $P^{k_i}_{Corrupted} = P^k_{Corrupted}$, and the previous equations reduce to

$$\Lambda^{OUT,\ COMP^k} = N_k^{TOT} * \lambda^{OUT,\ COMP^k} \quad , \tag{3}$$

$$\alpha_{COMP^{k+1}} = \frac{P^k_{Correct}}{(P^k_{Correct} + P^k_{Corrupted})} \quad , \tag{4}$$

where N_k^{TOT} is the total number of components at stage k.
Equivalently, the general model of Figure 1 is reduced to the equivalent simplified system model of Figure 6 that can be solved more easily.

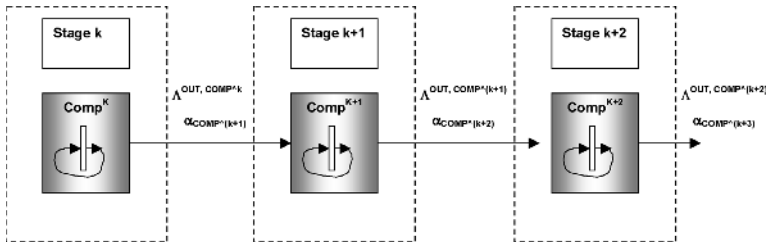


Fig. 6. Simplified system model equivalent to Figure 1

If it can not be assumed that the output process of COMP^{k_i} follows a Poisson distribution, the general approach is still valid provided that the detailed model is slightly modified allowing to estimate the real distribution of such a process. The same distribution will be used as input at the $k + 1$ stage. However, in general, it will be no longer possible to solve the models analytically.

If the measures of interest are probabilities, the moments of the distribution of the events which yield such probabilities are not considered at all. In this case it is not necessary to use, at the abstract level, models having the same distribution estimated at the detailed ones. If, on the contrary, we are interested in evaluating the moments of the distribution of correct/incorrect output emission/omission, the output processes distributions achieved by the detailed models have to be used for the solution of the abstract models.

The Schema. According to Figure 3 (showing the philosophy of our modeling approach) the model solution follows a bottom-up approach: solution of a detailed model is exploited to set up the parameters of the corresponding abstract model and of the detailed model of the next (contiguous) components (the

output of the detailed COMP^k model acts as input for the detailed COMP^{k+1} model). To keep the presentation simple, the model solution scheme is described in the case where all the N_k components at each stage k are identical; therefore only K detailed models (one for each stage) have to be solved. Figure 7 shows the relationships among a detailed model of COMP^k and the model COMP^{k+1}.

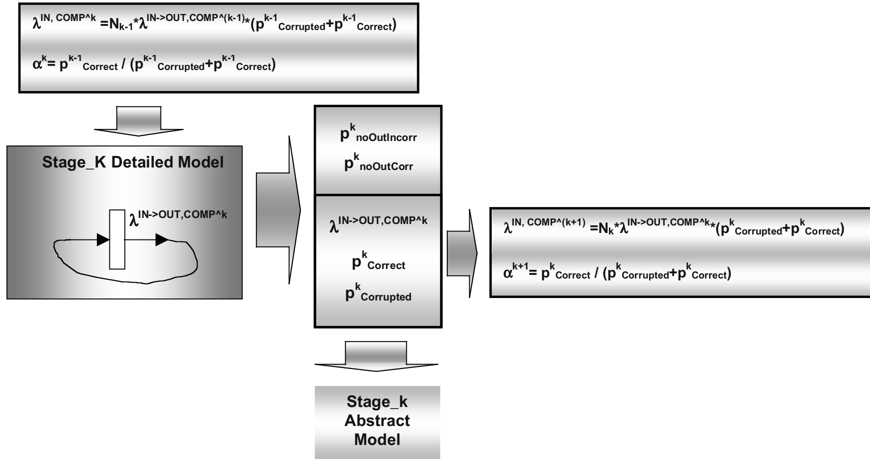


Fig. 7. Relationships between models solutions

With reference to the measures of interest listed in Section 2, the outcomes of the detailed model COMP^k are:

1. $p^k_{noOutCor}$: is the probability that no output is produced by component COMP^k, as a consequence of an incorrect input;
2. $p^k_{noOutIncorr}$: is the probability that an expected output is incorrectly not propagated by component COMP^k, as consequence of an internal fault;
3. $\lambda^{IN\rightarrow OUT,\ COMP^k} * (p^k_{Corrupted} + p^k_{Correct})$: is the rate of messages propagated by component COMP^k to component COMP^{k+1};
4. $p^k_{Correct}$: is the correct emission probability;
5. $p^k_{Corrupted}$: is the emission failure probability. This value encompasses both an expected wrong emission (as consequence of wrong internal processing) and the unexpected emission (as consequence of an internal self-generated false alarm).

All these parameters are used in the abstract model of component COMP^k (see Figure 7) while $\lambda^{IN\rightarrow OUT,\ COMP^k}$, $p^k_{Correct}$ and $p^k_{Corrupted}$ are used to derive the parameter $\lambda^{IN,\ COMP^{k+1}}$ to be used in the detailed model of COMP^{k+1}. In the system framework COMP^k and COMP^{k+1} represent two components directly connected that exchange messages in one direction (from COMP^k to COMP^{k+1}).

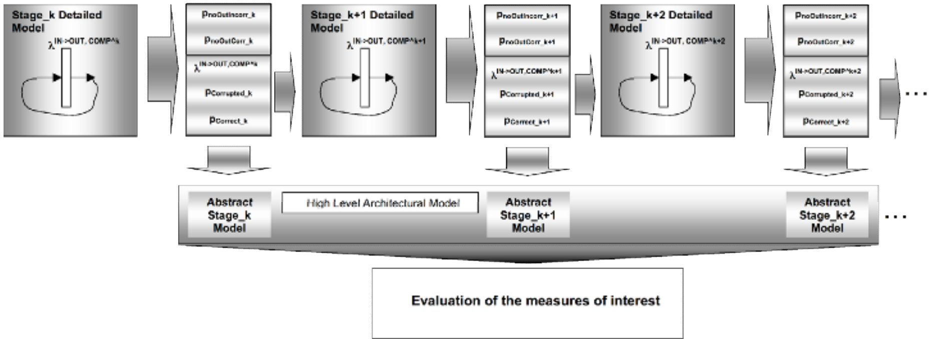


Fig. 8. Overall Solution Scheme

Summarizing, the overall solution scheme is shown in Figure 8. The detailed models are solved separately: firstly, it is solved the model of COMP^k, then the values provided by equations (3) and (4) are passed as input to the detailed model of COMP^{k+1} and so on. Finally, the probabilities of correct/incorrect output emission/omission are passed to the corresponding abstract models, they are joined together and then the overall abstract model is solved.

The advantages of the proposed approach are in two directions: first, to cope with the problem of state space explosion when modeling a complex system and, second, to allow efficient model solution for those systems having most of their components identical and interacting each others only by means of message exchange. Actually, in case the components are not all equal, a larger number of detailed models have to be solved but still separately. Thus, the overall model, encompassing all the useful information with respect to the measures of interest, is achieved by joining the abstract models.

4 An Application: The CAUTION++ System Architecture

The main objective of CAUTION++ [9] is the smooth transition from existing wireless systems to new generation ones. This is pursued by designing and developing a novel, low cost, flexible, highly efficient and scalable system able to be utilized by mobile operators to increase the performance of all network segments (namely, GSM, GPRS, UMTS and WLAN). Different segments can use different technologies and radio access. CAUTION++ exploits the available system resources by enabling real-time monitoring, alarming, immediate adaptive application of RRM (Radio Resource Management) techniques, vertical handover to other systems (possibly of other operators) having as a major goal to optimize the operators' revenue and the users' satisfaction.

The architectural solution is based on the concept of "monitor and manage". All resources at the air-interface are monitored in real-time and proper system

components are developed to handle generated alarms through a set of RRM (Radio Resource Management) techniques, to be applied where needed. The decisions-making process is performed at two levels: a local resource management in charge of managing the capacity of a each single network and a global resource management, which is in charge of inter-network coordination for the sake of the overall optimization of network capacity.

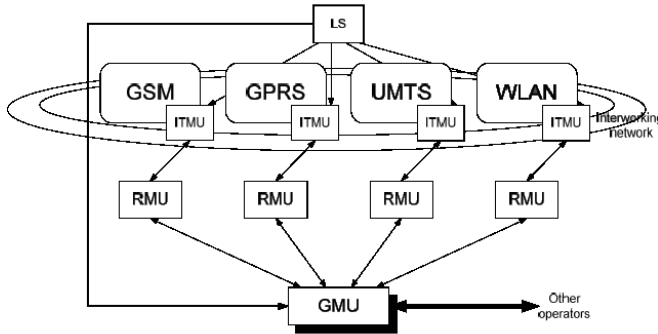


Fig. 9. Network Architecture for provision of capacity management mechanisms

Figure 9 shows the main components of the CAUTION++ architecture. Each network segment has its own ITMU (Interface Traffic Monitoring Unit) and RMU (Resource Management unit) which allow to monitor and manage the attached network, respectively. Within each operator network, a GMU (Global Management unit) can perform a global optimization. Different GMUs cooperate to optimize among different operators. A Location Server (LS) can be used to track users' mobility and location: such information can be exploited by GMU for a global optimization.

At the level of resource management provisions as offered by CAUTION++, the starting point in addressing dependability issues is that the system under development builds upon a number of existing network segments, each one characterized by specific dependability and performance properties, in accordance with the specific configuration adopted by the involved operators. The basic network segments are therefore to be regarded as "non-touchable" system components. Of course, their dependability and performance aspects can be analyzed, but despite they have an impact on the overall system figures, they have to be accepted as they are. Actually, there is relevant interest in analyzing performance and dependability aspects of wireless systems. In particular, the work in [4] contributes to the analysis of GPRS by providing a modeling approach suitable for investigating on the effects of outage periods on the service provision, with special attention on the user perception of the QoS. Two different levels of modeling have been considered. The first one defines a GPRS network availability model, which focuses on the dependability of the various components of the GPRS infrastructure, while the other one defines a GPRS service dependability model,

which builds upon the network availability model. The latter model takes as input the detailed stochastic characterization of the outages that is obtained from the GPRS network availability model, and maps on it typical service requests pattern of GPRS applications. The resulting composed model allows bridging the gap between the classical network perspective commonly taken when studying the availability of telecommunications systems, and a user and application centric analysis of the dependability of services that can be provided through the packet data service of GPRS.

Given the objectives of the CAUTION++ project , the dependability requirements have to be therefore achieved by acting on the components of its architecture and on the infrastructure connecting them.

The most important and challenging dependability requirement on the CAUTION++ architecture is to prevent RMU and/or GMU subsystems from carrying out wrong reconfiguration actions or when is not necessary (as consequence of some fault). Therefore, we are interested in the probability of correct/incorrect emission and omission at RMU and GMU level.

The solution scheme for CAUTION++ is presented in Figure 10. It consists of a three "stage" system in which each "stage" is composed of one component (ITMU, RMU and GMU detailed model). This schema could be also valid in a more complex scenario where more than one ITMU, RMU and GMU are present, provided that all components at the same level are identical. For a more detailed description of this case study, refer to [8].

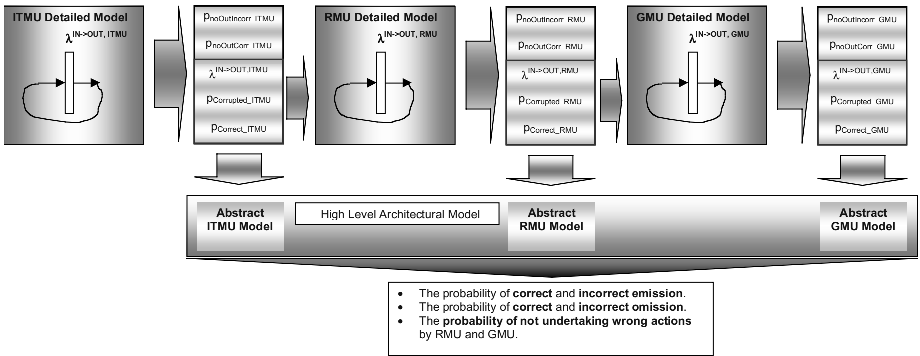


Fig. 10. CAUTION++ Overall Solution Scheme

5 Conclusions

This paper has focused on the development of a general evaluation methodology to master system complexity and favor model reusability and refinement as much as possible. The proposed method applies to a limited, but significant class of systems characterized by a hierarchical computation flow (typical representatives are control systems). In fact, the methodology exploits such system property to

set up a modular and compositional approach, both for the model design process and for the solution process. The resulting efficiency and easiness of the overall evaluation activity makes this method very attractive, when applicable. The CAUTION++ system architecture has been also briefly introduced, to show a practical case study for the described methodology.

The solution scheme has been presented in the most simple and efficient case where all components at the same level of the hierarchy are equal. Considering a more general case would imply a higher number of models to be individually set up and solved, but still retaining the same ability of modularity and compositionality. As future work, we intend to refine our approach, by providing a completely general description to fully address the target systems class.

Acknowledgments

This work has been partially supported by the European Community through the IST-2001-38229 CAUTION++ project and by the Italian Ministry for University, Science and Technology Research (MURST), project "Strumenti, Ambienti e Applicazioni Innovative per la Societa dell'Informazione, Sottoprogetto 4".

References

1. M. Balakrishnan and K.S. Trivedi. Componentwise decomposition for an efficient reliability computation of systems with repairable components. In *Int. IEEE Symp. Fault-Tolerant Computing (FTCS-25)*, pages 259–268, 1995.
2. G. Balbo. Introduction to stochastic Petri nets. In J.-P. Katoen, H. Brinksma, and H. Hermanns, editors, *Lectures on Formal Methods and Performance Analysis: First EEF/Euro Summer School on Trends in Computer Science Berg en Dal, The Netherlands, July 3-7, 2000, Revised Lectures*, volume 2090 of *Lecture Notes in Computer Science*, pages 84–155. Springer-Verlag, 2001.
3. G. Balbo, S.C. Bruell, and S. Ghanta. Combining queuing networks and generalized stochastic Petri nets for the solution of complex models of system behavior. *IEEE Trans. Computers*, 37(10):1251–1268, 1988.
4. S. Porcarelli, F. Di Giandomenico, A. Bondavalli, M. Barbera, and I. Mura. Accurate Availability Estimation of GPRS. *IEEE Transactions on Mobile Computing*, 2(3):233–247, 2003.
5. C. Betous-Almeida, and K. Kanoun. Stepwise construction and refinement of dependability models. In *Proc. IEEE International Conference on Dependable Systems and Networks DSN 2082*, Washington D.C., 2002.
6. J.F. Meyer and W.H. Sanders. Specification and construction of performability models. In *Workshop on Performability Modeling of Computer and Comm. Systems*, pages 1–32, 1993.
7. I. Mura and A. Bondavalli. Markov Regenerative Stochastic Petri Nets to Model and Evaluate Phased Mission Systems Dependability. *IEEE Computer Society Press*, 50, 2001.

8. S. Porcarelli, F. Di Giandomenico, A. Bondavalli, and P. Lollini. Model-based evaluation of a radio resource management system for wireless networks. In *Computing Frontiers*, pages 51–59, Ischia, Italy, April 2004.
9. CAUTION++ IST Project. Capacity Utilization in Cellular Networks of Present and Future Generation++. http://www.telecom.ece.ntua.gr/CautionPlus/.
10. I. Rojas. Compositional construction of SWN models. *The Computer Journal*, 38(7):612–621, 1995.
11. W. H. Sanders, and J. F. Meyer. A Unified Approach for Specifying Measures of Performance, Dependability and Performability. In *Dependable Computing for Critical Applications*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*, pages 215–237. Springer Verlag, 1991.
12. K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley and Sons, New York, 2001.

Rolling Upgrades for Continuous Services

Antoni Wolski and Kyösti Laiho

Solid Information Technology, Merimiehenkatu 36D,
FIN-00150 Helsinki, Finland
{antoni.wolski, kyosti.laiho}@solidtech.com

Abstract. With the advent of highly available systems, a new challenge has appeared in the form of the requirement for *rolling upgrade* support. A rolling upgrade is an upgrade of a software version, performed without a noticeable down-time or other disruption of service. Highly available systems were originally conceived to cope with hardware and software failures. Upgrading the software, while the same software is running, is a different matter and it is not trivial, given possible complex dependencies among different software and data entities. This paper addresses the needs for rolling upgradeability of various levels of software running in high-availability (HA) frameworks like the Availability Management Framework (AMF) as specified by SA Forum. The mechanism of a controlled switchover available in HA frameworks is beneficial for rolling upgrades and allows for almost instantaneous replacement of a software instance with a new version thereof. However, problems emerge when the new version exposes dependencies on other upgrades. Such dependencies may result from new or changed communications protocols, changed interfaces of other entities or dependency on new data produced by another entity. The main contribution of this paper is a method to capture the code, data and schema dependencies of a data-bound application system by way a directed graph called Upgrade Food Chain (UFC). By using UFC, the correct upgrade order of various entities may be established. Requirements and scenarios for upgrades of different layers of software including applications, database schemata, DBMS software and framework software are also separately discussed. The presented methods and guidelines may be effectively used in designing HA systems capable of rolling upgrades.

1 Introduction

The concept of *service continuity* embraced in the goals of the Service Availability Forum[1] is based on the notion that very short breaks in operation of service-providing applications are tolerable to a certain extent. This extent is specified using the availability measure *A* (percentage of the time a service is operational, as related to the total time the service is supposed to be operational) and, possibly, a maximum duration of a break (equal to mean time to repair, MTTR) or a frequency of breaks (represented with mean time between failures, MTTF). The three quantities are bound together with the formula:

[1] www.saforum.org

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp.175–189, 2005.
© Springer-Verlag Berlin Heidelberg 2005

$$A = \frac{MTBF}{MTBF + MTTR} \bullet 100\%$$

In the view of high availability standards like those of SA Forum, the main culprits preying on service continuity are failures—both of hardware and software. To deal with them, the system embodies redundancy both in hardware and software, managed by a high availability framework like AMF (Availability Management Framework) [4] of SA Forum.

According to the SA Forum AIS (Application Interface Specification) model [1], redundancy is maintained at the level of *service units* that may comprise of one or more components. In the simplest redundancy model, called 2N, the two units, *active* and *standby* make up a *mated pair*, and the redundant application components are organized in pairs in the corresponding units. Should a failure occur, the failed active (service) unit (hardware or software) is quickly replaced with a corresponding standby (service) unit. This operation is called a *failover*. Switching of the roles of units may be done also on request, in a no-failure situation, and this we will call a *switchover*. Switchovers are useful in various maintenance situations as will be seen in the sequel. Service continuity is preserved if, in the presence of failures, the required service availability level is maintained. If a standby system fails, it is repaired and brought back into synchrony with the active unit. Such a failure does not normally cause an interruption to the service.

All systems face a need for component replacement and upgrades from time to time. The need to facilitate software upgrades is demanding because a system with continuous service uptime expectation can not be just stopped for maintenance and upgrade. In order to provide service continuity, the hardware and software upgrades have to be performed on a running system in such a way that the availability requirements are met. We will call such upgrades *rolling upgrades*.

The concept of the rolling upgrade incorporates the notion of using the standby units present in a HA system and thus may be considered a special case of a *dynamic upgrade* in general [11]. It should be noted, hover, that engaging standby units in the upgrade process may temporarily jeopardize the availability level of the overall service because it may happen that the standby unit may not be available for failover, should this be needed. For this reason, we propose to use *spare units*, in place of standby units, whenever the service availability in endangered. Spare units are units that are not assigned any active or standby role. Such units are available in many HA system platforms. The choice whether to use the spare unit or not depends on the anticipated upgrade duration and the criticality of the component being upgraded. In the update scenario examples presented below, we make some educated decisions about using the spare units. In reality, such decisions have to be made on the basis of more accurate information about the required availability level and the duration of the upgrade.

Given the complexity of modern telecommunications systems where implementations are becoming increasingly software-driven, several interrelated software layers have to be recognized. In this paper, we are concentrating on systems utilizing database-centric applications, and thus the software layers considered for rolling upgrades are:

- Operating system and availability framework
- Database management system
- Database schema
- Database applications

The above layers are schematically shown in Fig. 1, together with the relevant interfaces among them.

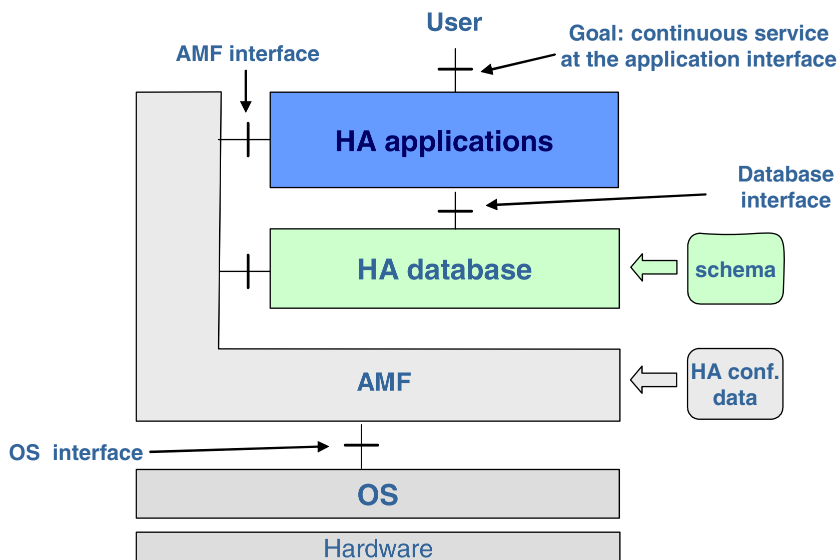


Fig. 1. Layers of software in an HA system

In Section 2, we survey the related work. In Section 3, the Upgrade Food Chain diagram is introduced with the purpose of capturing upgrade dependencies. In Section 4, requirements and scenarios associated with upgrades at different software layers are discussed. We conclude by summarizing the methods and guidelines produced.

2 Related Work

From the outset of uninterruptible systems, the needs for evolutionary changes, in a running system, have been recognized [7]. Consequently, various methods of dynamic (or live) upgrading (or updating) have been proposed (for review of early dynamic upgrading systems, see [11]). Researchers strived for achieving automatic upgrading systems and thus the proposed methods dealt with homogeneous components of low granularity. The update granules were abstract data types in Argus [2], procedures in PODUS [11] and tasks (or transactions) in Conic [7]. The emergence of well-defined component-based frameworks, like CORBA, J2EE and .NET, has offered new opportunities because of the unified component management and a possibility to represent component metadata in a natural way. There are methods for dynamic upgrading of CORBA components [14][8], Java RMI servers [12] and methods adaptable to J2EE EJB components [3]. Following the generally perceived needs, OMG has started an effort to produce the CORBA online upgrade specification [9], too.

Traditionally, the dynamic upgrades are expected to be *unattented* (i.e. automatic) and *safe* [3], i.e. not disrupting other components of the system. When building such a system, one has to answer two questions:

1) How to obtain and represent the necessary change and dependency information (upgrade metadata)?
2) How to execute the upgrade?

It is easier to answer the latter question once there is a satisfactory answer to the former one. Efforts have been made to extract the necessary metadata from the component interface specifications [14]. However, as the authors of [11] point out: "[fully automatic dynamic updating] cannot work properly if semantic information is needed to perform any aspect of the updating". Consequently, human input is needed to provide some of the metadata. An example is the ENT model (ENT stands for: Exports, Needs, Tags) [3] where the interface metadata is annotated with the changes in *provided-requested* relationships among components. Once the sufficient amount of metadata is produced, it can be used in unattended upgrading.

Inter-component dependency diagrams were introduced in [7]. In our work, we go further by introducing the Upgrade Food Chain (UFC) diagram that captures the version-specific change information only. This does not mean that the full dependency information is not needed: the change information is obtained by way of the differential analysis of the full dependency information.

A requirement for the component to be quiescent before it can be upgraded is often presented [14]. However, we argue that, in the presence of an HA framework like AMF, the components need not be necessary quiesced because they are not quiesced when a failover happens.

Similarly, it is required that the internal state is passed over to the new version of the component, to preserve the component correctness [11]. Our position is that we do not have to take care of that because the inherent nature of an HA component incorporates the notion of preserving the state in the presence of failover (or switchover). The means for achieving the preservation of state are application checkpoints [4] and writing the state into an HA database [5].

We are not aware of any work related to dynamic upgrades in large and diversified systems lacking a common component framework. In this work, we utilize the HA characteristics of a system, to ease the implementation of dynamic upgrades.

3 Rolling Upgrades: Dependencies and Requirements

3.1 Dependency Types

A major problem in facilitating rolling upgrades is that components of a system are interrelated. To picture the dependencies among system components, we choose to represent three different types of software components: *executable code* (standalone or library), *data* and *metadata*. *Code* represents independently startable applications and subsystems, and libraries to which they are linked. *Data* represents application data stored in a database or other persistent or run-time storage. Metadata means database schema declarations, such as table/data structures and integrity rules. One application version is typically bound to one version of schema, and may not work properly with a changed schema.

We introduce the Upgrade Food Chain (UFC) diagram to picture the dependencies among the software components discussed above. A possible UFC diagram may have the form shown in Fig. 2.

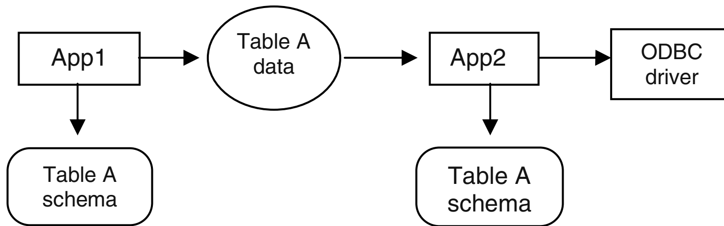


Fig. 2. Example UFC (Upgrade Food Chain) diagram

Consider a situation where two applications, *App1* and *App2* are upgraded to version *x+1*. *App1* uses data stored in a new table A. It thus needs also an upgraded database schema incorporating table *A*. The data in table *A* used by *App1* is produced by the upgraded *App2*. Additionally, *App2* needs a new version of an ODBC driver to function properly. The dependencies shown in the diagram are upgrade dependencies. Upgrade dependencies are special cases of inter-component function dependencies, as explained below.

Definition: Function Dependency
A component *A* is said to be *function-dependent* on component *B* if it requires some services or characteristics of component *B* to function properly.

If component *A* uses services of component *B*, it is said to be a *consumer* of services produced by *B*. Function dependencies among components are usually static and version-invariant. The reason is that, from the time of the component inception, its purpose and nature implies the related function dependencies. For example, all database-bound applications are function-dependent on the database schema, by definition. Exceptions from this rule may happen if the functionality of a component is changed significantly.

Knowledge of function dependencies is a sufficient, but not necessary, condition for execution of a safe multicomponent upgrade. Given an existing version *x* and the target version *x+1*, the necessary condition is the knowledge of version-specific function dependency, called upgrade dependency.

Definition: Upgrade Dependency
Assume we are upgrading components *A* and *B* from version *x* to *x+1*. Component *A* is *upgrade-dependent* on component *B* if the upgraded component *A* requires the functionality or characteristics increment, introduced in the upgrade of *B*, to function properly.

One can see that the purpose of upgrade dependability is to represent new dependencies that are introduced with a new version. If the two components involved are versioned in a different way, both new versions should be indicated in the dependency. On the other hand, if the function dependency of one component on another has not changed or is disappearing, with a given upgrade, it is not considered to be an upgrade dependency.

Definition: Upgrade Food Chain (UFC) Diagram
Upgrade Food Chain diagram is a directed graph, with each nodes being an instance of one of the three component types (code, data and metadata), and edges pointing to upgrade-dependent components.

Intuitively, the components should be upgraded in the reverse order of directed edges, starting from outmost components. All the components captured in a single UFC are considered a part of an *upgrade suite*. Upgrading of components in an upgrade suite has to be coordinated (ordered) so that the components can function properly during the upgrade process.

3.2 Assumptions About the System

Upgrade Granularity. The upgrade granularity we consider for SA-aware software is between (and including) the component and the service unit. A component is the smallest entity recognized by the AMF and also a natural unit of software development. A service unit (that comprises of components) is a unit of redundancy and thus switchovers are performed at this level.

Because both the concepts are irrelevant at the level of the operating system and the HA framework, the upgrade granularity for both is that a of a (cluster) node.

Distribution. An HA system is inherently distributed, not the least because of the hardware redundancy. Besides, the AMF has been planned for multi-computer clusters. Otherwise than assuming that components of one unit are co-located on the same cluster node, we do not make any references to the distributed nature of the system. We assume the function dependencies among components do not depend on the fact whether the components are co-located on a node or not.

Upgrade Transparency. When switchovers happen, the related component network addresses (service access points) change at each switchover. Upgrade transparency means that the consumer of the service, that is not upgrade-dependent on the upgraded service, should not be affected in any way by the upgrade. Because the upgrades we discuss are based on switchovers, the means for achieving upgrade transparency are the same as the means for achieving failure transparency, in an HA system, and we do not discuss it any further.

3.3 Trivial Upgrade: Independent Component

If a component upgrade is not dependent on any other component upgrade, it can be upgraded on its own because its upgrade suite does not comprise any other components.

To upgrade an independent component, a plain switchover may be applied. In this case, the procedure is shown below, given App_a^n and App_s^n are application instances of version n running as components in active and standby units, respectively.

To upgrade an independent code component App from version x to version x+1:

1) Stop the component App_s^x in the standby unit.
2) Install the new version of the component in the standby unit.
3) Restart the component as App_s^{x+1}.
4) Perform controlled switchover of units (App_a^x becomes App_s^x)
5) Stop App_s^x in the new standby unit.

6) Install the new version of the component in the standby unit.
7) Restart the component as App_s^{x+1}.
8) (Optional) Perform one more switchover if the original assignment of active and standby units was a preferable one.

Requirements. After performing step 4, the instances App_a^{x+1} and App_s^x have to interwork as a mated pair. If the active/standby operation at the component level involves communications between the active and standby component (e.g. to perform application state checkpoints), care should be taken of the need of the new version App_a^{x+1} to be able to communicate with the old version App_s^x, and possibly vice versa. If there is no intra-pair communications, e.g. if the component instances exchange data via a database, this concern is irrelevant.

Note that between steps 2 and 7, the system is vulnerable because it is running in stand-alone mode: there is no available standby component that can take over from a failed active component. For this reason, special precautions have to be taken if the period between steps 2 and 7 is protracted. Typically, you utilize a *spare unit* (hardware or software) to do the installation if it requires more time. Spare units are units that are not assigned any active or standby role.

3.4 Cycles in UFC

There may be a case as depicted in Fig.3. The two applications are dependent on data produced by the other one. An example may be that *App2* produces some statistical data based on transaction data produced by *App1*. On the other hand, *App1* is using the statistical data to optimize its own operation.

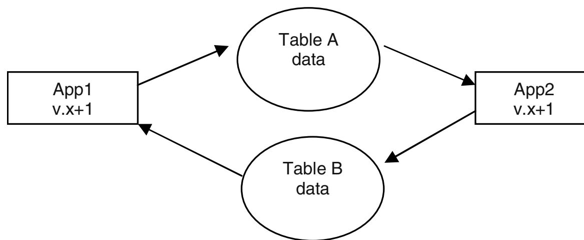


Fig. 3. Example of a cyclic UFC diagram

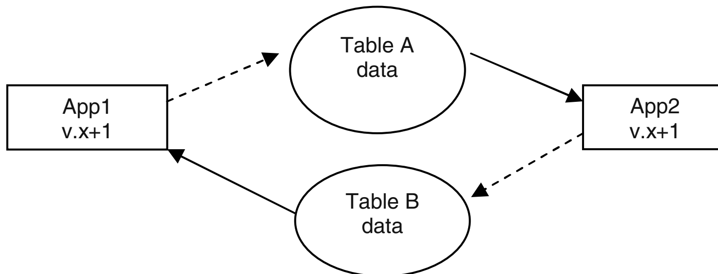


Fig. 4. Introducing weak dependencies (dashed)

If the depicted dependencies are *strong*, i.e. an application cannot operate without the data it is dependent on, we face a problem, because neither application will be able to operate. Therefore, the cycle has to be broken during the implementation of the application upgrade. One way is to implement the upgrade in such a way, that an application may operate, in a limited way, although the new data is not available. In such a case, the upgrade dependency between the application and the data is called a *weak upgrade dependency*.

In Fig. 4, weak dependencies are introduced, allowing to upgrade the two applications in any order.

Requirements. If a UFC cycle is detected, it has to be broken up during the upgrade implementation phase by introducing weak upgrade dependencies. Also, if there are dependencies among components of the same unit, it is preferable to change the dependencies to weak ones, because the actual order of setting the components to the active state may be *a priori* unknown.

Introducing weak dependencies is preferable also otherwise, to ease or remove the ordering requirements in the upgrade execution. There may be, however, some additional cost involved in making components weak-dependable on other components.

3.5 Acquiring and Using UFCs

The information captured in a UFC is mostly based on the incremental changes in the application semantics. If there exists component function dependency information captured in the component metadata similar to the ENT model in [3], the UFC may be extracted automatically by way of differential analysis of the metadata (between the current and the target version). In large diversified systems such metadata is not readily available. Therefore we assume the information pertaining to UFCs have to be acquired from the application developers when they are developing an upgrade. Once UFCs are available they may be used in constructing upgrade scripts to be run on a production system, or even used by an automatic upgrade facility. For this purpose, UFC graphs may be converted to a computer-readable form, e.g. using XML.

3.6 Other Assumptions

In the following sections, when we discuss upgrade scenarios, we make certain assumptions about the quality of upgrades:

- The upgraded software has been tested properly on a test system incorporating all know dependencies.
- The upgrade procedures have been also tested on a test system or on spare units of the production system.
- Because the process of generating UFCs from application semantics is human-centered, and therefore error-prone, one must prepare for the worst and have a backup plan for the situation where the upgrade (despite all proper preparations) is not successful, and the system has be returned to the state, that existed before the upgrade was started. We assume here that system backup images can be and are taken before the start of the upgrade process and that the backup state can be restored if needed.

4 Upgrade Scenarios

4.1 Operating System Upgrade

Operating system upgrade is slightly outside the scope of this paper, as the operating system is, typically, independent of the HA software running in the system. However the capability to perform the service unit switchovers may be utilized in OS upgrades, too. Because installing of a new version of an operating system may be a time-consuming process, spare nodes should be used to perform the installation in the background, without jeopardizing availability of the currently running services. Once the spare is upgraded, the standby node can be brought down and rapidly replaced with the spare, reducing the period of vulnerability of the system.

Similarly to all other software, we assume the compatibility and operation of the new version of the operating system has been tested on a separate test system.

Upgrade Scenario: Operating System

1) Install the operating system on a spare node
2) Install the HA framework, DBMS and applications if necessary.
3) Disconnect current standby node (i.e. the node running standby units) from the active node (resulting in a temporary standalone operation).
4) Transfer the database of the standby node to the upgraded spare node.
5) Assign the spare node the role of new standby node. The old standby node becomes a spare node.
6) The framework initializes the components and the active/standby operation resumes. The active and standby databases become reconnected and resynchronized.
7) Perform a controlled switchover.
8) Repeat steps 1-7 starting with the new spare node and new standby node.

The above scenario should be repeated for all pairs, in a 2N+M redundant system, where M is the number of spare nodes. If there are no spare nodes in a system, the periods of standalone operation will be longer, as the operating system is being upgraded on a standby node.

4.2 HA Framework Upgrade

An HA framework (like SA Forum's AMF) has interconnected instances running on each node. The HA framework upgrades may be dependent on the system model schema updates and new configuration files if they exist (see notes about monotonic schema upgrades in the following subsection). Another difficulty is that all SA-aware (meaning, in the SA Forum parlance, highly available) components are dependent on the framework software because they are typically linked to the framework's libraries. The UFC diagram for framework upgrade is shown in Fig. 5. Because the re-linking the applications make take some considerable amount of time, using of spare nodes is preferable, as in the previous case.

Upgrade Scenario: HA Framework

1) Perform the (monotonic) schema upgrade in the system model database to support the HA framework upgrade (if applicable)
2) Upgrade the HA framework at the spare node.

3) Re-link other SA-aware subsystems and applications with the upgraded framework libraries, at the spare node.
4) Disconnect current standby node (i.e. the node running standby units) from the active node (resulting in a temporary standalone operation).
5) Transfer the database of the standby node to the upgraded spare node (if applicable).
6) Assign the spare node to be a new standby node. The old standby node becomes a spare node.
7) Perform a controlled switchover.
8) Repeat steps 2-7 starting with the new spare node and new standby node.

The above scenario should be repeated for all node pairs, in a 2N+M redundant system, where M is the number of spare nodes.

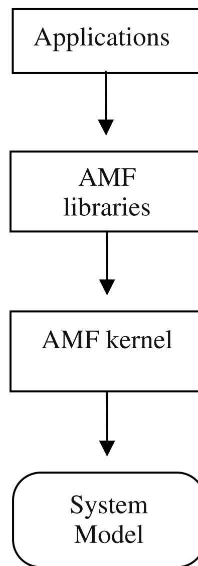


Fig. 5. UFC for HA Framework Upgrade

Requirements. In order for the presented scenario to succeed, the HA framework upgrade has to be engineered in such a way that the instances of the old version and new version of the framework can coexist in the same system. Should this turn out untrue, the rolling upgrade of the HA framework will be impossible, and closing down of the whole system will be required.

4.3 DBMS Upgrade

An HA DBMS must be engineered in such a way that rolling upgrade of the DBMS software is possible. Additionally, the involved dependencies and requirements have to be taken into account. The dependencies related to the DBMS upgrade are shown

in Fig. 6. The weak dependency of applications on upgraded driver libraries (such as ODBC) is explained in the way that the upgraded DBMS should be upward compatible with respect to drivers: the drivers of the old version can be used with the upgraded DBMS. Therefore, drivers may be upgraded at any later time (if a driver upgrade exists). The fact that there is a dependency of applications on new drivers may be explained by possible performance improvements in the drivers.

We assume that the database runs in the active/standby redundancy configuration. Given the assumed short time of performing the upgrade, the scenario does nor employ the spare node.

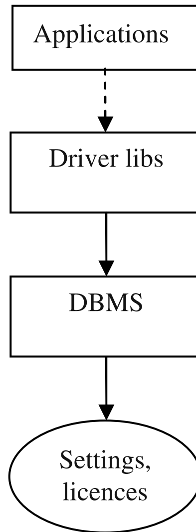


Fig. 6. Dependencies of the DBMS Upgrade

A DBMS upgrade scenario may be very much vendor-specific. The scenario shown below is supported in the Carrier Grade Option of the Solid Database Engine [13].

Upgrade Scenario: HA DBMS

1) Stop the standby DBMS server.
2) Upgrade the DBMS software at the standby node. This involves loading program media, necessary settings and license files into installation directories.
3) Start the upgraded server in the standby mode, with optional conversion mode enabled to convert the database to the format supported by the upgraded DBMS (if applicable). Note: if there are applications that are directly linked to the DBMS, they should be re-linked and restarted, too.
4) Reconnect the servers so they resume the active/standby operation. The necessary database catchup (state resynchronization) is performed automatically.
5) Perform the controlled switchover. The active node runs now the new version.
6) Stop the DBMS server running at the new standby node.
7) Install DBMS at the new standby node.

8) Start the upgraded server at the new standby node, with the optional conversion mode enabled to convert the database to the format supported by the upgraded DBMS (if applicable). Note: if there are applications that are directly linked to the DBMS, they should be re-linked and restarted, too.
9) Reconnect the servers so they resume the active/standby operation, although in the reverse active/standby node configuration. The necessary database catchup is performed automatically.
10) Perform the controlled switchover if the starting active/standby node configuration was the preferable one

Requirements. The crucial characteristics of a DBMS that is needed here is the capability of the new version to maintain the data replication stream with the old version. The minimum requirement is that the upgraded version may take up the standby role while the old version is running as an active. In order to make the upgrade painless for the applications, the new DBMS version must be totally upward compatible with the old one: there should be no change in the old functionality, although new functionality may be added. Also, assuming that there are a set of applications (on other nodes of the system) that should be able to use both older version and the newer version of the database (before and after the switchover), then the newer version of the database server needs to be compatible with the older version of the client API - such as ODBC and JDBC.

4.4 Schema Upgrade

Application upgrades are often dependent on schema upgrades as the new application functionality requires an enhanced data model. Thus, schema upgrades have to be installed before any depending application upgrades. The problem of schema upgrades (or, *schema evolution*) in production systems has been a recognized issue [10]. Typically, the objective of schema evolution is to satisfy the needs of new applications or application updates without jeopardizing the pre-existing applications.

In an HA environment, schema updates have to be performed on a live database, while the applications are running, because bringing the database totally off-line would endanger the overall availability goal. Fortunately, contemporary relational database systems typically support dynamic schema changes. Tables and columns may be added and dropped, referential integrity constraints may be redefined, etc. In an active/standby database pair, the schema changes have to be propagated from the active to the standby database.

Another problem is how to ensure that schema upgrade does not invalidate running applications. To do this, stringent limitations have to be enforced over schema upgrade design and application development. A schema upgrade that is upward compatible with the existing applications (with certain assumptions) is called a *monotonic schema upgrade*.

Definition: Monotonic Schema Upgrade
A schema upgrade is monotonic if and only if:

i. None of the first-class objects[2] is removed or renamed.
ii. None of the existing columns is removed or renamed

[2] First-class objects (in a relational database) are named schema objects created with the SQL CREATE statement, such as tables, views, constraints, triggers, etc.

iii. None of the existing integrity constraints is changed
iv. None of the existing active objects (stored procedures, triggers and events) is redefined

One can see, that a monotonic schema upgrade is, essentially, a schema extension. Objects like tables, views and triggers, table columns and related constraints may be added.

The fact that a schema upgrade is monotonic is not a sufficient guarantee that running applications are not invalidated with the upgrade. The applications themselves have to be built following the *schema-upgrade-safe* rules.

Rules for Schema-Upgrade-Safe Application Development
An application is unaffected by a monotonic schema upgrade if

i. It does not take advantage of any implicit column ordering.
ii. It does not take advantage of table dimensionality (number of columns).
iii. Its error processing (especially of DELETE statements) anticipate possible referential enhancements.

The effect of (i) and (ii) is that statements like SELECT *, and INSERT without explicit columns names, are forbidden. The reason for (iii) is that, as new tables may be associated with existing tables as referencing tables (having foreign keys pointing to existing tables), referential integrity violations may emerge. For example a DELETE statement on an existing table may produce a referential integrity error if there are dependent rows in a referencing table. Normal defensive programming (anticipating errors wherever errors are theoretically possible) will suffice. Additionally, new integrity rules may be added to the new foreign key definitions, like … ON DELETE CASCADE to guarantee that no new referential integrity errors emerge.

Given that the schema upgrade is monotonic and the application are built following the rules for schema-upgrade-safe development, rolling schema upgrades should be possible.

The monotonic schema upgrade should satisfy most needs of the normal application life cycle. Should there be a need for a non-monotonic upgrade involving renaming and changing of the schema semantics, a more careful approach is needed. In such a case, applications have to be scanned for possible change dependencies and reprogrammed accordingly, before the schema upgrade is applied.

The schema upgrade scenario may depend on the HA DBMS implementation used. If an active/standby HA DBMS is capable of propagating the schema changes, as well as data, from the active to the standby database (as does the Solid CarrierGrade Option of the Solid Database Engine), then the upgrade scenario is trivial.

Upgrade Scenario: Schema Upgrade

1) Apply the schema upgrade, dynamically, to the active database. The schema changes are automatically propagated to the standby database.

After creating the new schema elements, such as tables and columns, these are unpopulated (empty). It is often the case that portions of the existing data need to be migrated to the new schema, or that the new schema elements will need some default values or other seed data. Assuming that the applications are developed in a schema-upgrade-safe fashion, e.g. the existing applications can continue using the changed

database schema, the data migration and the new schema population can be applied to the operational active/standby database without causing downtime to service. For example, in the case of Solid Database Engine Carrier Grade Option, data migration tasks would be executed against the active database, and automatically synchronized to the standby, after which the schema upgrade is complete, and both database nodes are ready for use (for the new database client application versions). Note: in the case of large data migration requirements, the data migration itself may have a performance effect and lead to temporary service level degradation. This needs to be taken into account and tested properly when designing the rolling upgrade process.

After the schema upgrade is performed, next come the dependent application upgrades.

4.5 Application Upgrade

Because of possible dependencies, application upgrades should be carefully planned. As some applications may be producers and the other consumers of data, UFC diagrams may useful to capture the dependencies of the type shown in Fig. 2 and Fig.3. The cyclic dependencies have to be discovered early in the upgrade development cycle to allow for reprogramming the applications and introducing weak dependencies. Some of the weak dependencies may be then broken in the UFC graph, allowing for an acyclic graph. An acyclic UFC graph indicates the correct upgrade installation sequence, starting from the outer (leaf) nodes. Note that the ordering of the upgrade is that of partial ordering: any pair of mutually independent upgrades may be installed in any order. If several mutually independent or weakly dependent upgrades are comprised in a single service unit of a HA system, they may be installed in the same installation step. The UFC diagram may be then organized into a set of partially ordered upgrade steps

A single step of application upgrades is performed using the controlled switchover:

Upgrade Scenario: Application Upgrade

i. In a standby service unit of the system, stop the applications awaiting the upgrade.
ii. Install and start the upgraded applications in the standby mode.
iii. Perform a controlled switchover.
iv. Perform steps 1-3 in the new standby unit

Once an installation step is executed, the dependent upgrade steps may be performed. Throughout the time of the upgrade procedure, the applications are continuously available, with the exception of short breaks during switchovers. This way, the goal of providing continuous services is achieved in the presence of system upgrades.

4.6 Other Application System Architectures

In the presentation, we have mostly assumed two-tier (client/server) application architectures. In reality, more complex architectures may be used, including transaction processors, application servers and messaging frameworks like Web Services. In those architectures, the principles of the UFC diagram creation and usage remain the same although new component types may emerge in analysis.

5 Conclusions

Performing rolling upgrades on a continuously operating HA system is a demanding task. It can be successfully performed given proper methods and technologies are used. The prerequisites for a successful rolling upgrade at any level of the system are:

1) Finding out upgrade dependencies and capturing them with, for example, Upgrade Food Chain (UFC) diagrams.
2) Programming the upgrades in a way that allows for existence of weak dependencies and satisfies the rules of schema-upgrade-safe application development.
3) Assuring monotonic schema upgrades.
4) Using an HA DBMS that supports dynamic and uninterruptible schema update.
5) Using an HA DBMS capable of rolling upgrade of the DBMS software.
6) Using a HA framework software capable of doing a rolling upgrade of its own.

There are also several unresolved issues that require further study. Among them are: analysis of the performance impact of rolling upgrades, dealing with ad-hoc environments that do guarantee neither monotonic schema upgrades nor upgrade-safe applications, and satisfying the need to have a possibility to downgrade as well.

References

1. Application Interface Specification, SAI-AIS-A.01.01, April 2003. Service Availability Forum, available at www.saforum.org.
2. Bloom, T., Day, M.: Reconfiguration and module replacement in Argus: theory and practice. Software Engineering Journal, March 1993, pp. 102-108.
3. Brada, P.: Metadata Support for Safe Component Upgrades. COMPSAC 2002: 1017-1021.
4. Brossier, S., Herrmann, F., Shokri, E.: On the Use of the SA Forum Checkpoint and AMF Services. ISAS 2004, May 13-14, 2004 Munich, Germany.
5. Drake, S., Hu, W., McInnis, D.M., Sköld, M., Srivastava, A., Thalmann, L., Tikkanen, M., Torbjørnsen, Ø.,Wolski. A.: Architecture of Highly Available Databases. ISAS 2004, May 13-14, 2004 Munich, Germany.
6. Jokiaho, T., Herrmann, F., Penkler, D., Moser, L.: The Service Availability Forum Application Interface Specification. The RTC Magazine, June 2003, pp. 52-58.
7. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Trans. Software Engineering 18(11), pp. 1293-1306 (November 1990).
8. Van de Laar, F., Chaudron, M.R.V.: A Dynamic Upgrade Mechanism Based on Publish/Subscribe Interaction. COMPSAC 2002, pp. 1034-1037.
9. Moser. L.E., Melliar-Smith, P.M., Tewksbury, L.A.: Online Upgrades Become Standard. COMPSAC 2002, pp. 982-988.
10. Roddick. J.F.: Schema Evolution in Database Systems - An Annotated Bibliography. SIGMOD Record 21(4), pp. 35-40 (1992).
11. Segal M., Frieder O.: On-the-fly Program Modification: Systems for Dynamic Upgrading. IEEE Software, March 1993, pp. 53-65.
12. Solarski, M., Hein Meling, H.: Towards Upgrading Actively Replicated Servers On-the-Fly. COMPSAC 2002, pp. 1038-1046.
13. Solid High Availability User Guide, Version 4.1, Solid Information Technology, February 2004, available at http://www.solidtech.com.
14. Tewksbury, L.A., Moser, L.E., Melliar-Smith, P.M.: Live Upgrades of CORBA Applications Using Object Replication. ICSM 2001, pp. 488.

First Experience of Conformance Testing an Application Interface Specification Implementation

Francis Tam and Kari Ahvanainen

Nokia Research Center, P.O. Box 407, FIN-00045 NOKIA GROUP, Finland
{francis.tam, kari.ahvanainen}@nokia.com

Abstract. This paper describes our first attempt of conformance testing an implementation of the Service Availability Forum Application Interface Specification on a carrier-grade service platform for mobile communications applications. The requirements and guidelines of the IEEE Standards for measuring conformance to POSIX® have been adapted for the Application Interface Specification. The Test Method Specification structure is explained and assertions of the component registration function is shown as an example. A description of the Implementation Under Test is included, together with an explanation of the Test Method Implementation. Our experience suggests that this approach is indeed feasible and repeatable. The thorough level of testing appears to have the right balance of confidence and manageability for test cases. The intermediate test results also unexpectedly provide the developers with some useful insight into future implementations.

1 Introduction

With the advent of end customers in the telecommunications market wanting more new services at an increasing rate, and the dependability level of such services as high as the traditional ones, network infrastructure equipment manufacturers and application developers are faced with the challenge of delivering such services in ever shorter cycles. In December 2001, ten leading communications and computing companies announced an industry-wide coalition [1] to create and promote an open standards for Service Availability. By standardising programming interfaces for developing and deploying highly available applications, the Service Availability Forum attempts to tackle the issue of reducing development time and costs for highly available services.

The focus of the Service Availability Forum is to build the foundation for on-demand, uninterrupted landline and mobile network services. In addition, the goal is to come up with a solution that is independent of the underlying implementation technology. This can be achieved by first of all identifying a set of building blocks in the context of the application domain, followed by defining their interfaces and finally, obtaining a majority consensus among the suppliers.

To date, the Service Availability Forum has already published the Hardware Platform Interface Specification [2] and the Application Interface Specification [3]. The Hardware Platform Interface (HPI) primarily deals with the monitoring and controlling the physical components of a carrier-grade computing platform. By

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 190–199, 2005.
© Springer-Verlag Berlin Heidelberg 2005

abstracting the platform specific characteristics into a model, an HPI implementation provides the users with standard methods of monitoring and controlling the physical hardware via this abstraction.

The Application Interface Specification (AIS) defines the capabilities of a high-availability middleware interfacing with applications and the underlying carrier-grade computing platform. The AIS abstracts the high-availability characteristics into a model upon which an implementation provides standard methods to the application developers to respond and manage events such as failures. The expected end result is to maintain service continuity by an application even in the presence of failures. The AIS defines an extensive Application Programming Interface (API) for an application. The availability management and control capability is captured in the Availability Management Framework (AMF). In addition, services such as cluster membership, checkpoint, event, message and lock have been defined to support the development of such system.

As with any open standards solution, the publication of a specification is just the first step towards its goal. The next step is to establish whether a product, claimed to be standard compliant, does conform to the published specification. Measuring the conformance of an implementation against a published specification is one of such means. In this paper, we give our first experience of conformance testing an early implementation of the AIS on an internal carrier-grade platform product for mobile communications applications.

2 IEEE Standards for Measuring Conformance

The objective of conformance testing is to establish whether an implementation being tested conforms to the specification as defined in a standard. We attempt to follow, as much as appropriate, the requirements and guidelines offered by the IEEE standard 2003-1997 [4] for measuring conformance to the POSIX® standards. This is based on the observation that measuring the conformance of a Service Availability Forum AIS implementation is quite similar to measuring that of a POSIX® implementation. In particular, at the API level there is a collection of C functions. In addition, there is no compelling reason why we should not follow an existing standard where it is applicable.

Figure 1 depicts the basic model for conformance assessment as defined by the IEEE standard. In a Test Method Specification, a number of assertions are written for each element according to the base standard. An assertion defines what is to be tested and is stated in such a way that a test result code of `PASS` indicates conformance to the base standard. For each assertion, a set of allowable Test Result Code is stated. The set of Test Result Code associated with an assertion that a test program can report for a conforming implementation is known as Conforming Test Result Codes. However, the specification is mainly expressed using an informal, natural language such as English.

Each assertion in the Test Method Specification is then turned into one or more instances of a Conformance Test Software for that assertion. Together with the necessary procedures for conducting the conformance test, they form the Test Method Implementation.

During the execution of the Conformance Test Software, Intermediate Test Result Codes may be produced to provide more information on why a specific test result code is issued. A Final Test Result Code is then determined for an assertion test. If all the Final Test Result Codes match with the specified Conforming Test Result Codes, the Implementation Under Test (IUT) is considered to be conforming to the base standard.

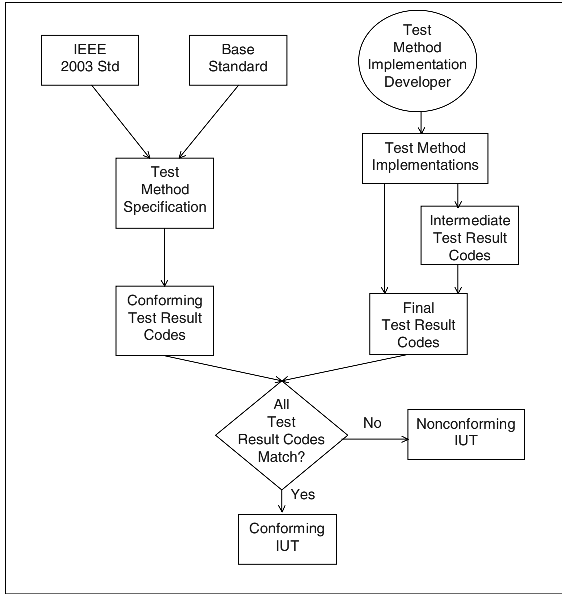


Fig. 1. Basic model for conformance assessment

The results of Test Method Implementation against an IUT is summarised in a Test Report containing information such as the identifications of the base standard, test method specification and test method implementation, result of each assertion test and the date when the IUT was tested.

The Test Method Implementation is documented with information such as how to install, configure and execute the Conformance Test Software, gather and interpret the results, and the known limitations.

Terminologies and Definitions

Assertion: It defines what is to be tested and is `TRUE` for a conforming implementation.
Base Standard: The standard for which the conformance assessment is sought. In this paper, the AMF part of the Service Availability Forum AIS.
Conforming Implementation: An implementation that satisfies all the relevant conformance requirements.

Conformance Log: A human readable record of information, produced as a result of a testing session, that is sufficient to verify the assignment of test results.
Conformance Test Software: Test software used to ascertain conformance to the base standard.
Element: A functional interface. In this paper, each function call as defined in the AMF of the Service Availability Forum AIS.
IUT: Implementation Under Test.
SUT: System Under Test. The system on which the Implementation Under Test operates.
Test Method Implementation: The means, typically a set of software and procedures, used to measure the conformance of an IUT to a base standard.
Test Method Specification: A document that expresses the required functionality and behaviour of a base standard. These are described as assertions and a complete set of conforming test result codes is provided.
Test Result Code: A value that describes the result of an assertion test.

Level of Testing

The IEEE standard distinguishes three major levels of testing. At the one end of the spectrum, exhaustive testing seeks to verify the behaviour of every aspect of an element, including all permutations. This however is normally infeasible due to the excessive number of tests required. At the other end of the spectrum is identification testing whereby only a cursory examination is required. In this case, simply having a match of the C function prototypes is considered to be conforming. This approach clearly lacks the confidence we seek in an implementation claiming to be conforming and is therefore considered not too useful.

In our first attempt of conducting conformance testing, we chose thorough testing that seeks to verify every aspect of an element but does not include all permutations. We frame our testing around all the possible return codes of an API, that is, we focus on all the success and error conditions. Since we exclude all the permutations of return codes, we therefore reduce the number of tests significantly and keep it down to a manageable set. For example, in `saAmfComponentRegister()` there are 11 return codes and therefore we only need 11 assertions. Under exhaustive testing however, we would have needed 2 to the power of 11 assertions, that is, 2048 of them.

3 Test Method Specification

The IEEE standard defines a generic assertion structure for describing a Test Method Specification. As a first attempt, we used a subset of the generic structure and we found that it was applicable. As such, we did not seek for any alternative structure. The following is the adapted structure in our conformance testing:

```
<Assertion_identifier>
    If <Option> then
          If <Test_Support> then
                (Setup: <Setup_Requirements>)*
```

```
                    Test: <Test_Text>
                   (TR: <Testing_Requirements>)*
                   (Note: <Notes>)*
             Else <No_Test_Support>
      Else <No_Option>
```

Clauses marked with * are optional. An `<Assertion_identifier>` is a unique label for an assertion within an element. It is made up of the following characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ ( ) -
```

The "**If** `<precondition>` **then** … **Else** `<outcome>`" is used to specify the precondition needed for testing an assertion.

`<Option>` represents the feature or behaviour defined in the base standard that need not be present in a conforming implementation.
`<Test_Support>` represents those facilities needed by a System Under Test to perform an assertion test.
`<Setup_Requirements>` are the steps that a test program must perform before performing a test of an assertion.
`<Test_Text>` specifies the test to be performed. The text usually contains action and result.
`<Testing_Requirements>` specifies the minimal testing required for an assertion.
`<Notes>` contains additional information of an assertion.

An Example

We have included below an example of the `saAmfComponentRegister()` API. Due to the space limitation, we have only shown three assertions.

ok

Setup: Use a valid AMF library handle in *amfHandle* and valid names for the component and proxy component in *compName* and *proxyCompName*.

Test: A call to *saAmfComponentRegister(amfHandle, compName, proxyCompName)* registers the component named *compName* to the AMF library represented by *amfHandle*. The call returns SA_OK.

TR: Test with local component registration.

error1

Setup: Use a valid AMF library handle in *amfHandle* and invalid names for the component and proxy component in *compName* and *proxyCompName*.

Test: A call to *saAmfComponentRegister(amfHandle, compName, proxyCompName)* returns SA_ERR_INVALID_PARAM.

TR: Test with local component registration.

error11

Setup: Use a valid AMF library handle in *amfHandle* and proxy component in *proxyCompName*, and a previously registered component name in *compName*.

Test: A call to *saAmfComponentRegister(amfHandle, compName, proxyCompName)* returns SA_ERR_EXIST.

TR: Test with local component registration.

In addition, there is a table detailing the assertions and their corresponding expected Conforming Test Result Codes for each element.

Element	Assertion	Conforming Test Result Codes
saAmfComponentRegister	ok	PASS
	error1	PASS
	error3	PASS
	error5	PASS
	error6	PASS
	error7	PASS
	error8	PASS
	error9	PASS
	error10	PASS, NO_OPTION
	error11	PASS
	error12	PASS, NO_OPTION

4 Implementation Under Test

The Implementation Under Test used in our work was a carrier-grade service platform for mobile communications. A high level block diagram of the platform is shown in figure 2.

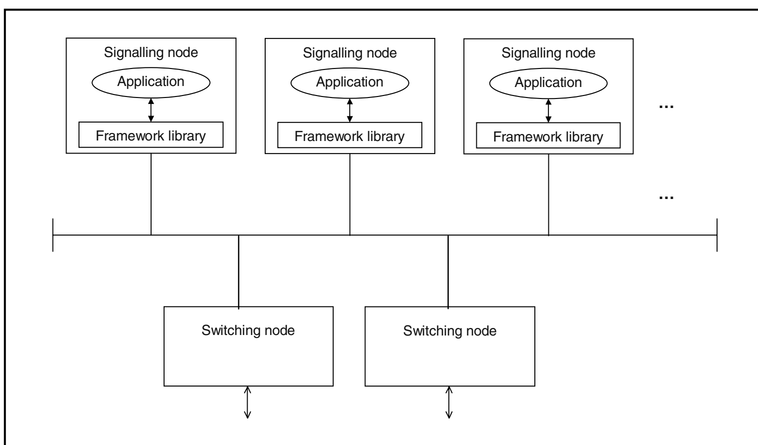


Fig. 2. Block diagram of the carrier-grade service platform

The service platform consists of a number of Signalling and Switching nodes interconnected via a high speed network. Collectively these nodes host 3G services. In addition, the Switching nodes interface with a routing platform that provides the routing and tunnelling functionality.

On this service platform, only a subset of the AIS AMF APIs was implemented on the Signalling node. However, it was based on a pre-release (draft version 0.8) of the specification.

5 Test Method Implementation

Figure 3 shows the positioning of the Conformance Test Software (CTS) with reference to the Test Method Specification during build time and the Implementation Under Test during run time. During the execution of the CTS the Framework APIs of the IUT are not accessed directly. Instead, an API and error code conversion is performed before and after the actual API call. This is because the IUT's APIs are based on an earlier version of the SA Forum API specification and some of the APIs, and the error codes they return, have changed in the published specification. The conversion is implemented as separate functions and can be removed later, for example, when it is not needed anymore after the IUT's APIs have been upgraded.

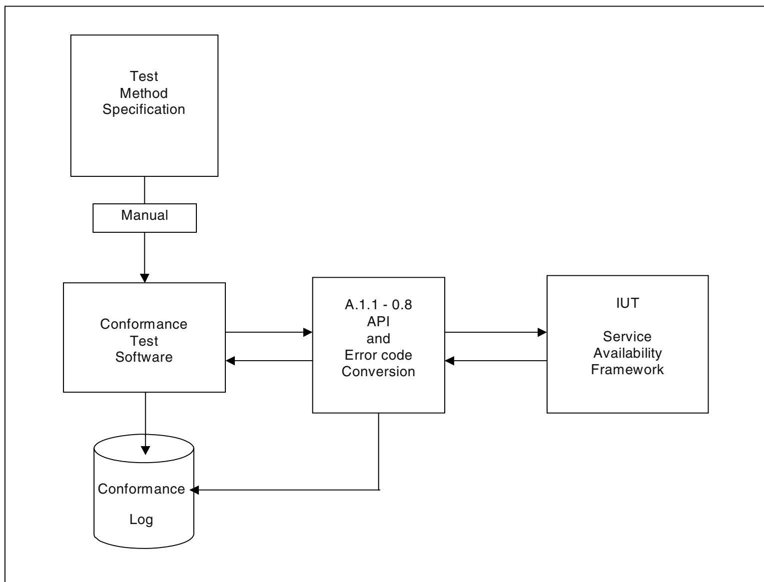


Fig. 3. Conformance test software

At build time, each assertion in the Test Method Specification is implemented as one or more tests in the CTS. The number of tests required to be implemented is typically derived from a set of conditions separated by the word *or* in an assertion.

The table below shows a subset of the test instances generated from the Test Method Specification for `saAmfComponentRegister()`. The parameters of the API under test are set accordingly reflecting the specified set of conditions in an assertion.

Tests	*amfHandle*	*compName*	*proxyCompName*	Expected Return Code
OK-1	valid	valid	not used	SA_OK
E01-1	valid	NULL	not used	SA_ERR_INVALID_PARAM
E03-1	uninitialised handle	valid	not used	SA_ERR_INIT
E03-2	finalised handle	valid	not used	SA_ERR_INIT
E09-1	NULL	valid	not used	SA_ERR_BAD_HANDLE
E11-1	Valid	previously registered	not used	SA_ERR_EXIST

The CTS, together with the conversion module, also produce a conformance log on the tested instances for off line examination. The following is the log output of one instance of assertion `error11` for `saAmfComponentRegister()`:

```
Timestamp: 11/11/2003 07:00:00
  Element: saAmfComponentRegister
Assertion: error11
 Instance: 1
DBG: saFwkInitialise returns 0
 Prologue: saAmfInitialize : OK
DBG: registering component name: CompRegE11
DBG: saFwkComponentRegister returns 0
 Prologue: saAmfComponentRegister : OK
DBG: registering component name: CompRegE11
DBG: saFwkComponentRegister returns 7
     TEST: OK
DBG: saFwkFinalise returns 0
 Epilogue: saAmfFinalize : OK
   RESULT: PASS
```

The log output has the following format:

"`Timestamp:`" followed by the test execution date and time in the format of "dd/mm/yyyy hh:mm:ss".
"`Element:`" followed by the name of the element, same as the name of the tested API.
"`Assertion:`" followed by the name of the assertion of this test instance.
"`Instance:`" followed by the instance name of the test assertion.
"`Prologue:`" followed by an API name that has been called to create the necessary preconditions for the test, followed by "`OK`" or "`NOK`" depending on the returned error code of such an API.

"TEST:" followed by "OK" if the return code is as expected, "NOK" otherwise.
"Epilogue:" followed by an API name that is called to create the necessary post conditions for the test, followed by "OK" or "NOK" depending on the returned error code of such an API.
"RESULT:" followed by "PASS", "FAIL" or "UNRESOLVED" depending on the test result and the success of the prologue and epilogue function return code. This result is an intermediate result and needs to be examined further.
"DBG:" These lines will appear in the log output only if "DEBUG" flag is defined in the test instance source files. These lines have some debugging information depending on the API being tested, for example:

- Name of the Framework API called for the pre- and/or post-conditions and its numeric return value
- Component name used as a parameter for the API
- Instance name used as a parameter for the API

6 Conclusions

The conformance testing we conducted involved 7 APIs of the AMF. The callback functions were deliberately excluded in the first phase of our work. Following the guidelines of the IEEE standard, we wrote 55 assertions and implemented 31 test instances. Due to the need for the API and error code conversion in our testing, we added an extra step to process the Test Result Code. If the expected return error code does not have a mapping to those in the base standard, the Test Result Code is set to UNRESOLVED in the final verdict. A test report recording all the verdicts of the conformance test for each API was produced.

Our experience of conformance testing an AIS implementation suggested that following the requirements and guidelines of the IEEE standard was indeed feasible. By following this process of developing and conducting conformance testing, we are able to repeat all the tests and obtaining consistent results. We also believe that at the thorough level of testing it has the right balance of confidence and manageability of tests. During the review of our test results with the IUT developers, we did not identify any falsely passed tests that would have indicated inadequate coverage of the test cases. On the contrary, one unexpected outcome was that the intermediate test results provided some useful insight into the future implementations for our developers.

Given the positive experience obtained from this work, we are continuing this effort to build a prototype for testing callback functions. In addition, we are exploiting the pre- and post-conditions, realised as prologue and epilogue in the implementation, in describing higher level scenarios involving multiple components and APIs.

We have also identified two areas for improvement. Framing the testing around all the possible return codes of an API appears to have achieved the goal of considering all the success and error conditions. However, there are error conditions, such as those indicated by SA_ERR_LIBRARY, that have a high level of implementation dependency. Therefore, there is a need to clearly define what should be included and excluded from the test definitions in order to avoid imposing any kind of structure onto an implementation. This of course has to be conducted within the Service Availability Forum in conjunction with the definition of the AIS compliance criteria.

The second improvement is related to the way in which the Test Method Specification is expressed. Using an informal, natural language such as English to capture the assertions is far from precise. We studied the use of ETSI's Testing and Test Control Notation version 3 (TTCN-3) [5] and the preliminary conclusion was that it was a potential candidate as a replacement. Although the core language itself is big, a subset of the constructs and the Runtime Interface [6] are thought to be applicable to our work. Further investigation is required and this may prove to be a candidate with the potential of bringing in the whole tool chain for the testing environment as well as the capability of automatically generating test cases from the Test Method Specification.

Acknowledgement

This work was funded by System Technologies, Nokia Networks under projects HA2003 and HARouter. We would also like to thank the iNOS Service Availability Framework development team for their technical support.

References

1. Service Availability Forum: http://www.saforum.org.
2. Service Availability Forum: Hardware Platform Interface Specification, SAI-HPI-A.01.01 (2002).
3. Service Availability Forum: Application Interface Specification, SAI-AIS-A.01.01 (2003).
4. IEEE Std 2003-1997: Requirements and Guidelines for Test Method Specifications and Test Method Implementations for Measuring Conformance to POSIX® Standards (1998).
5. ETSI ES 201 873-1 (V2.2.1): Methods for Testing and Specification; The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language (2003).
6. ETSI ES 201 873-5 (V1.1.1): Methods for Testing and Specification; The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (2003).

On the Use of the SA Forum Checkpoint and AMF Services

Stéphane Brossier, Frédéric Herrmann, and Eltefaat Shokri

Sun Microsystems
{stephane.brossier, frederic.herrmann, eltefaat.shokri}@sun.com

Abstract. In an environment where hardware and software errors may take an application down, high levels of service availability can be obtained by combining redundant hardware infrastructure with powerful application management frameworks which are able to detect these errors and recover from them. The Service Availability Forum (SAF) Application Interface Specification (AIS) defines interfaces used by the application management framework (AMF) to control applications placed under their supervision. The main goal of the AMF is to guarantee that for a given service there is always at least one instance of an application providing the service. In most cases the application implementing a service also encapsulates some data which is critical for the availability of the service. This data (called the application state) has to be preserved in a way that if the application providing the service fails, the replacing application can restore the application state and continue providing the service with minimal service disruption. The SAF AIS defines interfaces (named checkpoint interfaces) by which the application can save and restore its state. This paper describes the use of SAF AMF and checkpoint services in implementing applications providing highly-available services. The use of AMF and checkpoint services for implementing applications with the 2N or N-way active redundancy models is presented in the paper. It briefly discusses the interactions between the application, and AMF/checkpoint services during both (i) fault-free operations, as well as (ii) the error recovery procedures. The paper also suggests the most suitable checkpoint options for both of these redundancy models, depending on the tradeoff between protecting the integrity of the application state and the overhead of state saving/restoration by the application.

1 Introduction

This section introduces the need for state-full application failover as a mean to increase service availability. It also positions the Service Availability Forum (SAF) Checkpoint Service and SAF Availability Management Framework service as adequate facilities to support state-full application failover.

As stated in [1], the Service Availability Forum is an industry coalition of computer and communication companies dedicated to producing standard specifications that enables the development of carrier-grade systems based on commercial hardware platform, and operating systems. The SA Forum has developed an interface specification, named Application Interface Specification (AIS), for high-availability middleware. The AIS defines APIs needed for developing highly

M. Malek et al. (Eds.): ISAS 2004, LNCS 3335, pp. 200–212, 2005.
© Springer-Verlag Berlin Heidelberg 2005

available services. The Availability Management Framework and Checkpoint Services, the subject of this paper, are among the SA Forum AIS APIs.

1.1 Increase in Service Availability

In an environment where hardware or software errors can take an application down, high levels of service availability (for high availability systems, see for example [2,3]) can be obtained by combining a redundant hardware infrastructure with powerful application Availability Management Frameworks (AMF) [4]. The AMF can detect these errors and recover from them. The Service Availability Forum Application Interface Specification (AIS) defines the interfaces used by such AMF 's to control applications placed under their supervision. The AMF ensures that for a given service there is always at least one instance of an application active and providing that service. Consider applications as being grouped in two main classes:

- *single active instance" applications*: The application has been architectured in such a way that only one single active instance of the application can provide service at any given time. When this active instance fails, another instance must be activated (either by creating a new instance or by activating an existing instance). The SAF 2N and N+M AMF redundancy models support such applications.
- *"multiple active instance" applications*: The application has been architectured in such a way that multiple active instances of the application can provide service in parallel. When one active instance fails, other instances continue to provide the service. The AMF ensures that there is an appropriate number of active application instances. The N-way active and N-way AMF redundancy models have been designed to support such applications.

In most cases, a service encapsulates data: either global data for the entire service or specific data attached to each client of the service. A subset of this data needs to be persistent and stored on highly available disk subsystems (and backup archives). A second subset is more volatile and does not require this level of persistency, as in the case of call setup information or shopping cart contents. In the context of this paper, the volatile part of the service data is called the "application state".

To maximize service availability, both persistent application data and application state must be preserved through error recovery. Preserving application state through error recovery is often referred to as "state-full application failover capability".

1.2 Support for State-Full Application Failover

To support a state-full application failover, there should be a facility in the system for saving and restoring the application state across application and/or node failures. The application state can be preserved using the same techniques (and interfaces) as those used for the persistent data. However, the performance penalty for using these techniques is usually too high and so it is preferable for applications to replicate their state in the main memory of different nodes avoiding disk access.

Application state saving/restoration can be done in one of the following ways:

- *Application-transparent state save and restoration*: Under this approach, the memory segments (e.g., stack, and global variables) of the processes that constitute the application are considered the state of the application. This state is transparently transferred from one node to another node in the system, so that if the application on one node fails (e.g., crashes), then the underlying system on another node can

spawn another application (or more specifically the processes of the application). This new application has the same state as the failed application. Although transparency is a highly-desired feature of this approach, it usually imposes a non-negligible overhead. For complex applications, the state of the application is much smaller than the sum of all memory segments of its processes.

- *Message-based checkpointing*: In this approach, the application is fully responsible for transferring its state (incrementally or one-shot) through the use of messages sent directly to its standby partners. This approach naturally increases the complexity for the application developer to ensure that the application state is reliably transferred to the standby partners, even when failures occur. Moreover, bringing up a new standby instance of the application and making its state consistent with other active participants in the service without a major effect on the service performance is a very complex issue.
- *Storage checkpointing*: In this approach, highly-available applications save and propagate their state by one of the following approaches:
  - *Conventional database solutions (usually in-memory databases)*: This approach simplifies the application development compared to the message-based checkpointing solution. However, it might not support adequate performance for the save/restoration of the application states for time-critical applications such as telecommunication applications.
  - *Specialized checkpoint services*: The logic behind this approach is that the conventional database solutions might not fully satisfy the need of time-critical applications because they might demand higher performance which cannot be satisfied by general-purpose databases which enforce strong ACID properties [6].
- Checkpoint services emphasize fast save and restoration of application-specified state over other criteria such as stronger semantics. Checkpoint services normally offer high-level specialized interfaces for reading and updating checkpoints. They also ensure that the data stored in the checkpoint are transparently replicated on multiple nodes in the system.

The SAF Checkpoint API [5] is defined as a standard interface for such specialized checkpoint services. It is important to mention that it is not intended for the SAF checkpoint service to provide transactional properties. It is mainly targeting a fast transfer of state between applications.

If a 2N redundancy application uses AMF and checkpoint services, the application will have the following desirable capabilities:

- *Fast restart of state-full applications on the same node*: The active application keeps track of its state by writing into the checkpoint locally. If the active application fails and is restarted on the same node by the clustering software, the application can use the local copy of the checkpoint to re-create its state. Since the checkpoint is stored in memory on the local node, the restart operation is very efficient.
- *Seamless failover/switchover*: An active application keeps track of its state by writing into a checkpoint. The application state is transferred to the nodes where the standby partners reside. If the primary fails and the clustering software decides to failover the application, then one of the standby applications will take over and will use the local copy of the checkpoint to recreate its state, efficiently.

The benefits of using this combination of services will be discussed in Section 3.

2 Main Characteristics of the Checkpoint API

This chapter provides a brief description of the SAF checkpoint features and characteristics. Detailed description of these characteristics can be found in [5].

2.1 Structure of a Checkpoint

Checkpoints are implemented as a set of checkpoint replicas. Several instances of these replicas are kept by the checkpoint service to ensure checkpoint redundancy. There cannot be more than one replica of a given checkpoint on a cluster node. The number of replicas can be configured as part of the checkpoint configuration. The SAF Checkpoint API does not put any limitation on the number of replicas per checkpoint, however in the context of this discussion we assume that most implementations will limit the number of replicas to a small number per checkpoint as the cost of maintaining too many replicas would be prohibitive.

Checkpoints are structured as a set of sections that can be created and deleted dynamically by processes. A process can create sections of various sizes. Each section can be considered as a time-dependent and interrelated set of information (e.g., session information on an application server or call information in call processing applications). Each section has an expiration time which is used by the checkpoint service to automatically delete the section when it expires.

2.2 Checkpoint Options

Checkpoints can be created/opened with different options. These options can be combined together to fit different usage patterns.

The two classifications for the checkpoint options are:

(i) *Classification of checkpoint based on the control over the location of replicas* Checkpoints can be opened with the 'collocated' or 'non collocated' attribute [5]. When the collocated attribute is used, it means that a replica is created on the node where the open operation takes place (which is the node where the application is running). The application has control over the location (and number) of checkpoint replicas by opening the checkpoint on appropriate nodes. It should be noted that the maximum number of replicas per checkpoint supported by a particular checkpoint service implementation puts a limit on the number of nodes where applications can access a particular checkpoint in collocated mode. If an implementation supports a maximum of two replicas per checkpoint, that checkpoint can only be accessed in collocated mode on up to two nodes at a given time. The application is also responsible for choosing the node where the active replica (the replica where the update is written first) is located.

If the collocated attribute is not used, the locations, the number of the replicas and the choice of an active replica are the responsibility of the checkpoint service (but might be constrained by some checkpoint configuration attributes).

(ii) *Classification of checkpoint based on the semantics of updates* Checkpoints can be created as *synchronous* or *asynchronous*:

- In the synchronous mode, the update operation (write, creation of a new section, deletion of an existing section) is performed on all the replicas before returning the call.
- In the asynchronous mode, only one replica (called the active replica) is updated when the update call returns. The other replicas are updated asynchronously by the checkpoint service. The SAF Checkpoint API provides two variants of asynchronous mode but this paper does not go into that level of detail.

The important difference between the synchronous and asynchronous modes is that in the case of asynchronous updates the non active replicas could be slightly out of date compared to the active replica. If the node containing the active replica fails, the latest updates might not have been propagated to all replicas at the time of the failure. An application instance taking over the service on another node needs to be architectured to cope with this possible state loss (its internal state might be slightly out of sync with the latest interactions it had with some of its clients).

2.3 Combination of Different Checkpoint Options

This section compares the strengths and weaknesses of different combinations of checkpoint options.

- *Asynchronous and Collocated Checkpoint:*
  This combination achieves better update performance by relaxing the synchronicity of updates. It is the best suited combination for applications that are performance critical but can cope with occasional losses of checkpoint updates. A leading example is call processing applications. These applications demand sub-second fault-recovery latency but can cope with occasional loss of calls in the event of rare and complex failures in the system. The 2N redundancy model fits well with this combination and will be discussed further in Chapter 3. The asynchronous/non-collocated combination offers the following advantage:

  - *Best performance*: The combination of the asynchronous mode with the collocated attribute provides a very fast update capability compared to other combinations. The main reason for the significant improvement in the performance is because updates are done synchronously *only* on the local replica. This avoids any synchronous remote communication and therefore offers the maximum performance for update operations.

  However, this combination suffers from the following weaknesses:

  - *Possible loss of successfully-returned updates*: If the active replica is lost (for example due to a node failure), there is a possibility of losing some of the latest information since other replicas might not have received the latest updates. However, more effective implementations can reduce the probability of these losses.
  - *Limited application scalability:* The number of nodes where the application can open the checkpoint is limited to the maximum number of replica supported per checkpoint.
  - *Active replica chosen by the application:* Some complexity is added to the application as it is responsible for designating the location of the active replica.

- *Asynchronous and Non-Collocated Checkpoint:*
  This combination of options is suitable when an application's two most important requirements are update performance and application scalability. An example is an application with multiple instances active in parallel as in the N-way active redundancy model. This combination of options offers the following advantages:

  - *Improved performance*: The asynchronous mode still offers some performance since only one replica is updated synchronously. However, as the active replica location is chosen by the checkpoint service it might be located on a node other than that running the application. Also, the operation of updating this replica is more costly than in the collocated case. So, we do not recommend this for small-scale applications with high performance requirements.
  - *Application scalability*: The non-collocated attribute ensures that there can be any number of applications running on different nodes and accessing the same checkpoint even when the implementation limits the number of replicas per checkpoint to two. In other words, the redundancy level of the application is not bound to the redundancy level of the checkpoint.
  - *Replica management is transparent to the application:* Since the checkpoint service automatically handles operations such as creating a new replica, and selecting the active replica, the application is simplified.

  However, this combination suffers from the following weakness:

  - *Possible loss of successfully-returned updates: See* Asynchronous/Collocated.

- *Synchronous and Collocated Checkpoints:*
  This combination provides some performance improvement as replicas are collocated with the instances of the application using the checkpoint. It is best suited to applications that cannot cope with the possible loss of checkpoint updates during node failures. This combination offers the following advantage:

  - *No loss of successfully-returned updates*: Since the update operations are synchronous, when an update call returns the checkpoint service guarantees that the update is successfully propagated into all the replicas. In other words, when there are no ongoing update operations, all replicas are identical. Therefore, if one replica is lost (for example, as a result of a node crash), other replicas can be still used and the service can continue with the latest updates and with no, or minimal, delay in the service.

  However, this combination suffers from the following weaknesses:

  - *Performance penalty*: As discussed earlier, the synchronous semantics impose a performance penalty for update operations. This penalty is minimized because one replica is local. Checkpoint read operations, performed when a standby takes over, are optimal as there is a guarantee of a local replica.
  - *Limited application scalability:* See Asynchronous/Collocated case.
  - *Active replica chosen by the application:* See Asynchronous/Collocated case.

- *Synchronous and Non Collocated Checkpoints:*

  This combination is the simplest for applications to use because it provides the strongest update semantics and the application does not need to handle the management of the checkpoint replicas. Additionally, the number of application instances is not limited to the number of replicas of a checkpoint. However, this combination might exhibit weaker performances compared to previous combinations. This combination offers the following advantages:

  - *No loss of successfully-returned updates*: See Synchronous/Collocated case.
  - *Replica management is transparent to the application*: As discussed above for Asynchronous/Non Collocated.
  - *Application scalability*: As discussed above for Asynchronous/Non Collocated.

  However, this combination suffers from the following weakness:

  - *Performance penalty*: Update operations take longer since all the replicas have to be updated synchronously before the update call can return to the application. This makes the service less performant in fault-free environments. Also the taking over of the service by the standby application can take longer as reading the checkpoint might require access to a remote replica.

3 Checkpoint Services in 2N Redundancy Applications

Under the 2N redundancy model, in its simplest form, there is one active instance of the application providing service, with one standby instance running and ready to take over when the active application fails. The vast majority of 2N redundant applications have states, and for the standby to continue the required service effectively, the state of the application should be retrieved by the standby before it can take over the service. An effective way of implementing state-full 2N redundant applications is to use of the SAF AMF and checkpoint services, as illustrated in this chapter.

3.1 Suitability of Checkpoint Options to 2N Applications

The semantics of the SAF checkpoint service is more suited to the model where the standby application reads the checkpoint *only* when it is asked to take over the service. This type of standby is usually called a *warm standby*. The SAF checkpoint service does not offer an easy way of implementing a *hot standby* application where the standby application has the most updated state as soon as the state is stored in the checkpoint service by the active application. This is mainly because the standby application does not get notification of checkpoint updates.

As denoted in the SAF Specification (and discussed briefly in Chapter 2 of this paper), the checkpoint service offers various checkpoint options. The most suitable option for a 2N redundant application is a collocated checkpoint with the asynchronous update option for the following reasons:

- *Fast update of the checkpoint by the active application*: Assuming that the active replica (the one on which the update operation is performed synchronously) of the checkpoint is collocated with the active application, then a checkpoint update will be very quick compared to the other options, because an update does not involve heavy operations such as inter-node communications and inter-replica synchronization. This increases the service performance in a fault-free environments. In general, it is very important for any high-availability solution to maintain the service performance in fault-free environment.
- *Fast restoration of the application state by the newly-elected active application*: Since a replica of the checkpoint is located in the standby application node, then when the standby becomes active, it can quickly read the most up-to-date state of the application from the local replica. No network operation is needed for this.

However, as explained in Section 2, with the asynchronous option there is no guarantee that all replicas of the checkpoint have been updated when an update operation returns. Therefore, there is a risk that the newly-elected active node does not have all of the latest updates made by the failed active application. As a result, the service developer has to make a tradeoff between synchronous and asynchronous updates: “preventing the loss of state” or “fastest write in the checkpoint”. In the discussions in the rest of this chapter we will assume an asynchronous update mode.

3.2 Application Interactions with the Checkpoint Services

This section describes the normal interactions of the active and standby applications with the checkpoint and AMF services.

- *Active application*: When an application instance is about to become active, it opens the checkpoint for the purpose of writing, if it has not opened the checkpoint before. If there is no local replica for this checkpoint, the checkpoint service will automatically create a local replica before returning from the open call. The active application should set the local replica to be the active replica of the checkpoint. At this point the active application is ready to provide the specified service. When the application receives a request from a client to perform a service, it executes the request and updates the checkpoint when appropriate.
- *Standby application*: The standby application opens the checkpoint for the purpose of reading its content. If there is no local replica of the checkpoint on the node, then the checkpoint service automatically creates a local replica before returning from the open call. After opening the checkpoint, the standby application waits for an order from the AMF to become active. The standby application is not involved in keeping its local replica up-to-date as this is done automatically by the checkpoint service. It is important to note that the standby application may read the checkpoint whenever it wishes, but the checkpoint service does not have any information for the reader application about the updates on the checkpoint. Therefore, it is difficult for the standby to obtain incremental changes in the state.

As shown in Figure 1, the active application is responsible for setting the active replica of the checkpoint. If the local replica of the active application is not set active by the application, then the checkpoint updates will not be optimized for performance.

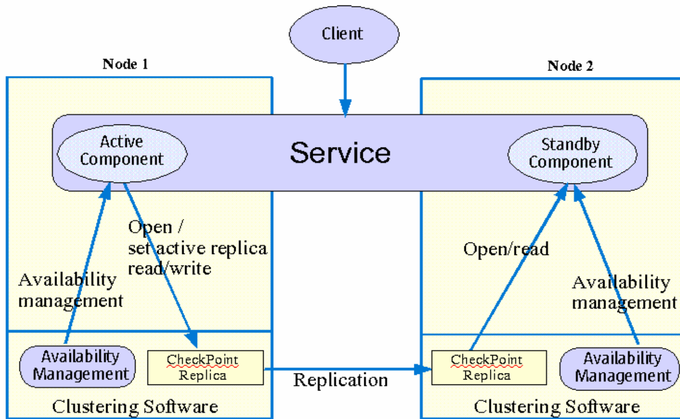


Fig. 1. The Use of AMF and Checkpoint Service for 2 N Redundant Applications

3.3 Responses to Failures or Administrative Operations

This section describes how a change of HA state by the AMF affects the way a particular application instance accesses its checkpoint according to its new HA state.

3.3.1 Failure of the Active Application

As specified in the SAF AMF Specification, the system administrator can configure the AMF so that the first few failures of an active application do not cause an application failover but simply a local application restart (meaning that the failed application does not lose its active status). However, after a configurable number of consecutive failures in the active application, the active application will lose its active state. We discuss here how the application and the clustering software (i.e., AMF and checkpoint service) react in each of these two cases:

- Restart of the failed active application: In this case, the active application is restarted on the same node and is made active by the AMF. The active application then reads the checkpoint and updates its internal state accordingly. Assuming that the retention time of the checkpoint was properly setup, the checkpoint replica should still be present (and up-to-date) on the node when the application restart completes. So in this case, the checkpoint service does not need to fetch any data from a remote node and the recovery of the application state is local and fast.
- Failover of the service: Based on the service configuration (or some other global information), the AMF may decide that the service should be failed over from the failed active application to its standby partner. In this case, the standby application gets a notification from the AMF to become active. Upon receipt of such a notification, the standby application performs the following actions:
  - Read the content of the checkpoint and update its internal state accordingly.
  - Set the local replica to be the active replica.
  - Start acting as the active application and resume provision of service to its client. This might trigger new updates in the checkpoint where needed. After the recovery completes (i.e., the standby took over the active role), the AMF might start another instance of the application (either in the failed

application node or another node in the cluster) and make it a new standby. When this newly started application is asked by the AMF to become standby, the application opens the checkpoint. If there is no replica of the checkpoint on the node where the standby application runs, then the checkpoint service creates a local replica of the checkpoint before returning from the open call. Therefore, after returning from the open call, the application becomes standby and waits to be told by the AMF to take over the active role.

3.3.2 Failure of the Standby Application

When the standby application fails, depending upon the service configuration, the AMF may perform one of the following recovery operations:

- Restart of the failed application: In this case, the failed application will be restarted and be made standby by the AMF.
- Failover of the standby application: In this case, another instance of the application will be made standby (during which the application will open the checkpoint). If there is no local replica of the checkpoint in the node where the new standby application is located, then a local replica will be created there.

3.3.3 Application Switchover

There are situations where the AMF decides to move the active service to the standby application, even when the current active application does not demonstrate any sign of ill health. Such a situation could be to perform a hardware upgrade on the nodes where the active application runs. In these situations, both active and standby applications collaborate with the AMF for a seamless transfer of the active service from one node to another. The steps that an active application (A1) and a standby application (A2) go through are as follows:

1. A1 made QUIESCED: Before entering the QUIESCED state (See [4] for the definition), A1 updates the checkpoint so that its last state is recorded in the checkpoint. It, then, notifies the AMF of its success in going into QUIESCED state.
2. A2 made active: After successful completion of Step 1, the AMF orders A2 to become active. A2 goes through the procedure of becoming active as discussed in Section 3.3.1 and notifies the AMF of its success.
3. A1 made out-of-service: Finally after successful completion of Step2, the AMF removes the service from A1. In this step, A1 closes the checkpoint.

The cases where some of these steps fail are out of the scope of the paper.

3.4 Redundancy Models Similar to 2N

Based on this usage example in the context of a simple the 2N redundancy model, it should be noted that the usage of collocated checkpoints (in asynchronous or synchronous mode) is best suited to execution environments where there is a single application writing to the checkpoint and where the location of the standby applications is already known before recovery takes place.

Without going in too much details about AMF concepts, we want to recommend that applications use one separate checkpoint for each *component service instance* [4]. An application handling several component service instances in parallel would manage as many checkpoints as the number of component service instances.

In all 2N, N+M and N-way redundancy models currently defined in the AMF Specifications, only one component is active for a component service instance at a given time and a few other components are standby for the same component service instance. The number of components on standby is likely to be very small, consistent with the limitations put on the number of replicas per checkpoint by the checkpoint service. Hence in all these models, using collocated checkpoints for each component service instance seems the most appropriate choice.

4 Checkpoint Service in N-Way Active Redundancy Model

The N-way active redundancy model differs from other models defined by the AMF in that several instances of the application can be active in parallel providing a single service. All active applications open the same checkpoint and use it to save/retrieve their state as needed.

In a call processing environment or in a web server environment for instance, the clients connect to one of the active applications for the duration of the session. The active application creates a temporary zone (section) in the checkpoint, which is used to save the content of the current session.

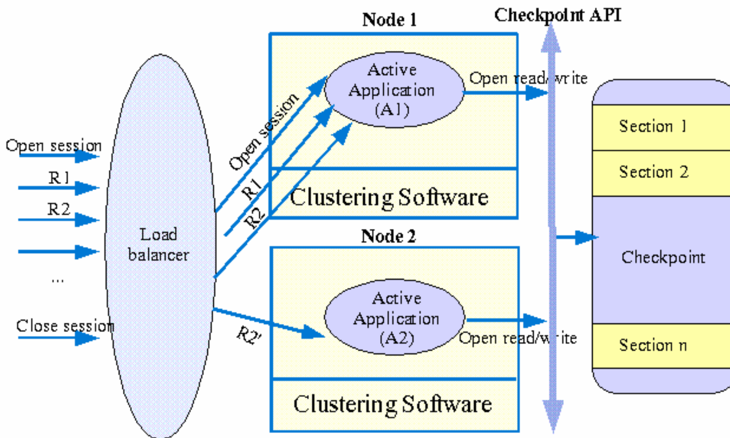


Fig. 2. The Use of Checkpoint in N-Way Active Redundancy Model (Client View)

Figure 2 depicts a cluster where a client opens a session 'S1'. The load balancer chooses the active application on Node 1 (A1) to handle the current session. A1 creates a new section 'Section1' in the checkpoint. Each new request emitted by the client contains an identifier for the opened session 'S1'. The load balancer forwards the request to the application A1, and this one updates the section 'Section1' and replies to the client. If everything goes fine, the client ends up by closing the session, and A1 destroys the section 'Section1' in the checkpoint.

If A1 crashes before the session terminates, for instance if it crashes after the request R1, the AMF can either restart A1 on the same node, or restart it on a different

node. The client does not see this failure, and when it sends the request R2, the load balancer either forwards R2 to A1 if it has been restarted on the same node, or to a different active application (A2 on node 2). The active application which receives R2 (or R2') uses the session identifier to figure out that there is a current open session related to this request. It uses the checkpoint API to read the section 'Section1', and retrieves the current state of the session. At this point, it handles R2, updates 'Section1', and replies to the client

In this model, only one given active application handles a session at any given time. Therefore, the section in the checkpoint associated to the current session is modified by only one active application at a given time, and there is no need for synchronization between the active applications. In this N-way active model, the application should use non-collocated checkpoints so that application scalability is not limited by the number of replicas supported by a particular checkpoint service implementation.

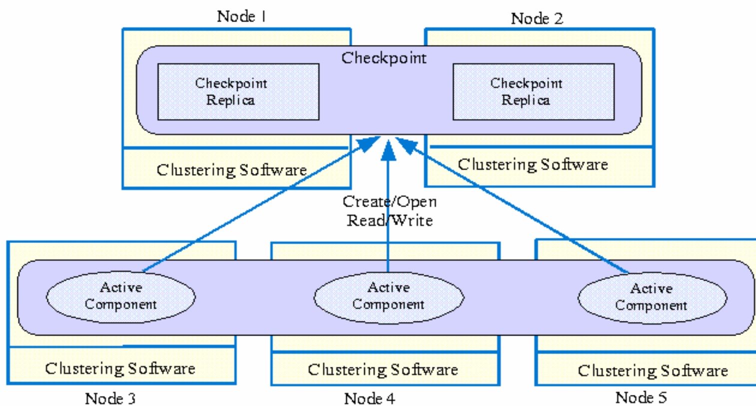


Fig. 3. Use of Non Collocated Checkpoints in an N-Way Active Redundancy Model

Figure 3 depicts a five node cluster. The service is composed of three active components relying on the checkpoint service to maintain the checkpoint replicas. In this case, the checkpoint service maintains two replicas. If the active applications create a synchronous checkpoint, both replicas are updated synchronously. If it creates an asynchronous checkpoint, only one replica (i.e., the active replica) is updated synchronously and the other is updated by the checkpoint service in the background.

5 Concluding Remarks

The paper discusses the position of the SAF Checkpoint API relative to other checkpointing alternatives. It claims that the checkpoint service provides optimum performance for time-critical applications that can cope with occasional losses of checkpoint updates that can occur due to node failures. The paper also identifies several combinations of checkpoint options and discusses the pros and cons in each

case. The paper also illustrates how to combine the SAF AMF and the checkpoint service for implementation of the 2N and the N-way active redundancy models. As shown in the paper, the SAF Checkpoint API is well suited to support cold and warm standby models.

References

1. Jokiaho, T., Herrmann, F., Penkler, D., and Moser, L.: The SA Forum Application Interfaces Specification, RTC, June 2003.
2. Laprie, J. C., Arlat, J., Beounes, C., Kanoun, K.: Definition of Hardware and Software Fault-Tolerance: Definitions and Analysis of Architectural Solutions, IEEE Computer, July, 1990.
3. Powell, D.: Delta-4: A generic Architecture for Dependable Distributed Computing, in Research Notes ESPRIT (Vol. 1), Springer-Verlag, May 1991.
4. Service Availability Forum Application Interface Specification: Availability Management Framework API , April 2002.
5. SA Forum Application Interface Specification: Checkpoint Service, April 2002.
6. Garcia-Molina, H., Ullman, J. D., Widom, J.: Database Systems, Prentice Hall, 2001.

Author Index

Ahvanainen, Kari 190
Arun, S.G. 17

Bondavalli, Andrea 160
Bozinovski, Marjan 33
Brossier, Stéphane 200

Dague, Sean 48
Di Giandomenico, Filicita 160
Drake, Sam 1

Herrmann, Frédéric 200
Horbank, Matthias 134
Hu, Wei 1

Ibach, Peter 134

Jhumka, Arshad 148
Johansson, Andréas 148

Kwon, Min–Hee 118

Laiho, Kyösti 175
Larsen, Kim 33
Liu, Xiliang 101
Lollini, Paolo 160

McInnis, Dale M. 1

Neises, Jürgen 73

Park, Jong–Tae 118
Porcarelli, Stefano 160
Prasad, Ramjee 33

Ravindran, K. 101
Reinecke, Philipp 86
Reisinger, Heinz 61
Renier, Thibault 33

Sârbu, Adina 148
Schwefel, Hans–Peter 33
Seidl, Robert 33
Shokri, Eltefaat 200
Sköld, Martin 1
Srivastava, Alok 1
Suri, Neeraj 148

Tam, Francis 190
Thalmann, Lars 1
Tikkanen, Matti 1
Torbjørnsen, Øystein 1

van Moorsel, Aad 86

Wolski, Antoni 1, 175
Wolter, Katinka 86