Alan Holt

Network Performance Analysis

Using the J Programming Language

Springer

Network Performance Analysis

Alan Holt

Network Performance Analysis

Using the J Programming Language

Springer

Alan Holt, PhD
agholt@gmail.com

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2007936013

Printed on acid-free paper

ISBN 978-1-84628-822-7 e-ISBN 978-1-84628-823-4

9 8 7 6 5 4 3 2 1

Springer Science+Business Media

springer.com

Chi and Emily

Preface

The J programming laguage is rich in mathematical functionality and ideally suited to analytical computing methods. It is, however, a somewhat terse language and not entirely intuitive at first, particularly if one is used to more conventional programming languages such as C, Pascal or Java. J functions (called *verbs*) are denoted by punctuation symbols. New functions can be developed by combining sequences of existing verbs into *phrases*. The compositional rules of J govern how the functions are combined and thus how they are applied to data objects. *Conjunctions* provide the programmer with additional control over the behavior of verbs and the composition of phrases.

The J programming language was conceived by Kenneth Iverson (also the author of APL) and Roger Hoi in 1991 (note that J is not in anyway related to Java). It is an interpretive language written in C. It is, therefore, highly portable and can run on a number of architectures and operating systems. Architectures currently supported are Intel, PowerPC and ARM, as well as 64 bit AMD. There are runtime versions of J for a number of Unix variants, such as Linux, Mac OS X, and Solaris, for example. Windows and Windows CE are also supported.

The last platform, Windows CE, is of particular interest. If, like the author, you have tried a number of scientific calculators, you may have found them packed with functionality but cumbersome to use. PDAs (personal digital assistants), on the other hand, have better user interfaces. They are also quite powerful and have relatively high-resolution colour graphics. However, in terms of mathematical features, they are somewhat limited and are little more than electronic organisers. For an engineer then, J is the "killer app" for the PDA. A PDA (running Windows CE) becomes the ultimate scientific calculator. In this environment, a terse language like J is a benefit, as the input method is typically limited to a touch screen with a stylus. Programming in more verbose languages, like C or Java makes the PDA impractical.

In this book, we use the J programming langauge as a tool for carrying out performance analysis of packet data networks. All the examples in this book were developed on the J.504 version of the interpreter. There are many excellent books on the market on network performance analysis. However, they tend to be heavy in mathematical rigour. Rather than covering these topics in terms of proofs and theorms (which the other books do so well anyway), it is the aim of this book to introduce concepts and principles of network performance analysis by example, using J.

J is available for download from `www.jsoftware.com`, where installation instructions are provided as well as online documentation. There are also a number of forums that you may subscribe to that will give you access to the wealth of experience and knowledge of the J community.

Acknowledgments

The author would like to thank the following people for the valuable contribution they made to this book: Dr Lin-Yi Chou (University of Waikato), Dr Sue Casson (Leeds University), Dr Adrian Davies, Michael Dewsnip (DL Consulting), Dr Chi-Yu Huang (DL Consulting), Zhiwei Liu (University of Waikato), and Prof. John Monk (Open University). Thanks also to Catherine Brett of Spinger for guiding me through this process.

I especially want to thank my wife for all her love and encouragement throughout this period. I could not have undertaken this project without her support.

Alan Holt

Contents

List of Figures

List of Tables

1 Introduction

For reasons of tractibility, classical queuing theory assumes that the properties of network traffic (arrival rates, service times) include the Markovian property. While the Markovian assumption is valid for telephony networks, it has been widely reported that, for data networks, traffic is fractal in nature [6, 13, 16, 37]. One way to represent a traffic flow is as a stochastic process. Modelling traffic as Markovian stochastic processes is attractive because flows can be characterised by only a small number of parameters. Models of non-Markovian processes, however, are more complex and analytical results are difficult to derive. Markovian traffic can be clearly distinguished from Internet traffic patterns which exhibit properties of *long-range dependence* and *self-similarity*. That is, traffic is bursty over a wide range of time scales. Backlog and delay predictions are frequently underestimated if the Markovian property is assumed when traffic is actually long-range dependent (and self-similar) [16, 28].

A number of methods for generating fractal-like traffic processes have been put forward. Examples include Fractional ARIMA processes, Fractional Brownian motion, chaotic maps and the superposition of heavy-tailed on/off sources. Network performance can then be investigated using simulation techniques. Some of these fractal traffic generating techniques are described in this book. We use J to simulate network systems and analyse their performance under certain traffic conditions.

An alternative approach to simulation is *network calculus*. Network calculus is a recent development where the exact properties of flows are unknown. The mathematical foundations are based upon *min-plus* algebra, whereby the *addition* and *multiplication* operators of conventional algebra are exchanged for *minimum* and *plus* operators. Instead of representing a traffic flow as stochastic process, a flow is characterised by an *envelope* function. Service curves can also be expressed in this way. Performance bounds can be derived through the application of min-plus algebraic methods on flow arrival curves and network system service curves.

Another approach is to consider *closed-loop* methods of modelling networks. Internet transport protocol such TCP (Transmission Control Protocol), and more recently DCCP (Datagram Congestion Control Protocol [34]), adjust their transmission rates according to the congestion state of the network [29]. TCP sends data a "window" at a time. TCP initially probes the network for bandwidth by increasing its window size each time it receives an acknowledgment. TCP's transmission rate is regulated by the acknowledgment round-trip time. In the event of congestion, TCP reduces its window size (and thus its transmission rate) and begins probing the network for bandwidth once again. Congestion control algorithms can be modelled as dynamical feedback systems, and flow patterns are governed by transport protocols reacting to network events.

The aim of this book is to present network performance analysis techniques that do not rely on the Markovian assumption. Topics in network performance analysis are presented with the aid of the J programming language. J is rich in mathematical functionality, which makes it an ideal tool for analytical methods. Throughout the book a practical approach is favoured. Functions in J are developed in order to demonstrate mathematical concepts, this enables the reader to explore the principles behind network performance analysis.

The topics covered in this book are motivated by the author's own research and industrial experience. The book outline is as follows. Chapters 2 and 3 provide an introduction to the J programming language. Chapter 2 introduces the basic concepts such as the data types and built-in J functions. Chapter 3 covers more advanced topics, where the focus is programming in J.

Network calculus is introduced in Chapter 4. We demonstrate how arrival and service curves can be expressed as J functions. We show how to derive upper bounds for backlog, delay and output flow. Effective bandwidth and equivalent capacity techniques are introduced as a means of deriving the network resource requirements for deterministic QoS bounds.

Chapter 5 focuses on statistical analysis and stochastic processes. The concepts of short-range and long-range dependence are introduced and we implement models for generating time series with these properties. We introduce Autoregressive (AR) and Moving Average (MA) models for generating short-range dependent time series. Fractional Autoregressive Integrated Moving Average (FARIMA) models are presented for generating time series with long-range dependence and self-similarity.

In Chapters 6 and 7, we show how to simulate traffic with both short-range and long-range dependence properties. In Chapter 6, we simulate traffic with discrete on/off models using various random number generators to generate traffic with the desired correlation properties. In Chapter 7, we show how chaotic maps can be used to generate simulated traffic.

ATM QoS is covered in Chapter 8. Leaky bucket and virtual scheduling algorithms are developed in J. We show how cell conformance can be analysed using these algorithms by running them on simulated traffic from continuous on/off models.

Chapter 9 examines Internet congestion control. We use J to build binomial congestion control algorithms and explore the parameters that govern congestion window increase and decrease.

1.1 Quality of Service

The Internet was principally designed to offer a *best-effort* service [12]. While the network makes a sincere attempt to transmit packets, there are no guarantees with regard to reliable or timely delivery. Packets may be delayed, delivered out of order or dropped. The end-to-end protocols (such as TCP) are given the responsibility of recovering from these events. Packets incur delays during transmission due to link speeds, and propagation delays. Delays are also incurred by packets waiting in buffer queues. Memory buffers are used to resolve contention issues when two (or more) packets arrive simultaneously at a communications link. A packet's waiting time is determined by the number of packets that are scheduled ahead of it; thus waiting times increase with traffic volume. Furthermore, if traffic volumes are excessive, contention for buffer memory arises, which is resolved by dropping packets.

Traffic management methods (and associated policies) can help to alleviate the trade-off between QoS and network optimality. Traffic management is a complex technical issue, but it is also an economical, political and ethical one. Like any limited resource, the allocation of network capacity can be somewhat controversial. Traffic management policies are selected either to implement a *neutral* network, or a differentiated one. In a neutral network, there is parity between users and services in terms of how the network treats their packets. A neutral network however, does not necessarily support fairness. Due to the flat-rate pricing policy of the Internet, heavy users do not incur any additional financial cost over light users. Yet the burden of congestion is borne by all.

In a differentiated network, there is discrimination between users and/or services. The network provider controls the allocation of resources, electing to give some users a better than best-effort service (and others a less than best-effort). Better than best-effort services are covered extensively in the literature ([22, 73], for example). Certain classes of user require either priority allocation or an assurance of a minimum allocation of resources during periods of congestion. Resources are allocated according to the particular QoS requirements of the user's application. The aim is either to guarantee or at least optimise QoS metrics, such as delay, jitter, throughput or loss.

Less than best-effort (LBE), services are not so clearly defined. Typically, best-effort is considered the lowest level of service [23]; that is, high-priority users are allocated resources according to their needs, and any remaining resources are allocated to lower-priority users. One may argue, however, that if a particular set of users is receiving a preferential service relative to another, then any "best-effort" commitment to the low-priority users is not being met. The service they are receiving, therefore, is almost certainly *less than* best-effort.

Nevertheless, the literature maintains that a best-effort service can be supported despite the differentiation in flows and preferential resource allocation. Furthermore, low-cost LBE services may be offered alongside best-effort and high-priority services to users of applications that are tolerant to high loss rates, delay and jitter

[19, 25]. During periods of congestion, LBE packets are delayed or dropped in preference to best-effort or high-priority traffic. Applications with LBE delivery can make use of off-peak periods when network resources are underutilised. This enables network providers to use otherwise spare capacity, without affecting best-effort or higher priority traffic.

LBE services however, extend beyond low-cost service offerings by providers as a means of selling available capacity on under-utilised links. There have been cases where network providers have attempted to “discourage” the use of certain services over their network. Voice over IP (VoIP) is a typical example. There have been reports of provider discrimination against VoIP services. At the most basic level, providers simply *block* IP ports such that VoIP traffic cannot pass between the sender and receiver. A more subtle form of discouraging VoIP applications is delay packets such that the QoS is degraded beyond a usable level [50]. VoIP users may be unaware they are being discriminated against, believing that the low voice quality is merely a by-product of a best-effort service, when in actual fact the provider is inflicting a less-than best-effort service on them.

Less-than best-effort services can also arise from network provider business practices. The Internet is no longer a network formed through the “cooperative anarchy” [65] of government agencies and academic institutions. The infrastructure is divided amongst many competitive Internet Service Providers (ISPs). Tier 2 ISPs “buy” routing tables wholesale from transit providers; thus traffic is exchanged between ISPs via transit networks. Rival ISPs, however, may elect to cooperate and exchange traffic through a mutual *peering* arrangement [48]. The benefits of peering are (potentially) two-fold. Firstly, ISPs in a peering arrangement reduce their transit costs. Secondly, by eliminating the transit hop and taking a more direct route, traffic latency is reduced. A peering arrangement between two ISPs may be attractive to one ISP but not the other. Without the consent of both parties, peering will not occur. This has given rise to a number of dubious business practices, whereby one ISP tries to encourage other to peer with it. For example, if ISP B receives traffic from ISP A through some prior peering arrangement with another ISP, then ISP A could, using policy routing, forward its traffic to ISP B through a transit provider. Furthermore, ISP A may compound ISP B’s transit costs by “replaying” traffic using a traffic generator [49]. Such practices do not constitute a sincere attempt to deliver packets, and thus packet delivery is *not* best-effort.

Textbooks describe the Internet Protocol as a best-effort packet delivery mechanism [12] and Internet commentators talk of “retaining” a neutral network [9]. According to Sandvig, however, “the Internet isn’t neutral now” [60]. Nor has it been for a long time. As the Internet grew in the 1990s, it suffered from the *tragedy of the commons* [46] and was subject to “overgrazing.” Furthermore, the Internet carries content that some find undesirable. Providers adopt differentiated traffic management policies in order to address these issues. Consider the following scenarios:

- According to reports, 75 to 95 percent [18, 74] of electronic mail is unsolicited (SPAM). Providers go to a great deal of trouble trying to distinguish SPAM from legitimate e-mail so that it can be purged from the network.
- Users vary considerably when it comes to their consumption of network resources. In order to prevent a small number of heavy users causing congestion, some ISPs impose *caps* on their subscribers. Subscribers that exceed their caps are either blocked or charged extra.
- The Internet has undergone a revolution in the People's Republic of China, yet it is a prime example of how "the net can be developed and strangled all at once" [67]. Content providers, in accordance with legislation, cooperate with the Chinese government to censor traffic that carries information deemed to be "sensitive."

Discriminatory practices on the Internet are a reality; it is merely a question of which practices one finds acceptable. As Sandvig points out, network neutrality is not the issue, it is "who discriminates and for what purpose" [60]. Furthermore it may not be appropriate to view QoS in terms of the service levels centred around *best-effort*, but rather as varying degrees of the preferential allocation of resources.

As the amount of real-time traffic on the Internet increases, such as voice and video, it is essential that support for QoS is provided. First and foremost, QoS is about capacity planning. Network resources need to meet traffic demands. Users, however, have a sporadic need for resources, and peak demand is very rarely sustained for long periods. If the network capacity level is set to the peak demand, then the network will be idle for long periods. Fortunately, many applications are not entirely intolerant to some delay or loss. The network, therefore, does not have to be excessively overprovisioned. Nevertheless, it is not easy, given the stochastic nature of network traffic, to balance QoS requirements and resource efficiency.

1.2 Network Utilisation

In this section, we discuss the effects of network utilisation on performance. We take the opportunity to introduce J and use it to explore some of the concepts presented. Network utilisation is the ratio of demand over capacity. The load on a network link cannot exceed 100 percent of capacity. It is possible, however, for the *offered* load to exceed this figure, in that case the available buffer memory temporarily stores excess traffic. Excessive backlog, and thus delay, can be avoided by limiting the level of network utilisation. This is achieved by monitoring the traffic volumes and setting the capacity relative to the offered load, so that the demand/capacity ratio (utilisation) is sufficiently low so that the backlog is kept within acceptable bounds. The problem is finding the utilisation level which yields "acceptable bounds" for the backlog.

A commonly cited rule of thumb for network utilisation is 30 percent [61, 68]. There appears to be some confusion in some literature between *achievable utilisation* and the *level of utilisation* that can support a given QoS. Passmore conveys the performance "limitations" of Ethernet [51]:

> shared-media CSMA/CD Ethernet LANs where thirty percent is the effective utilisation limit.

Ethernet, however, has no such limitation, it is capable of much higher utilisation levels, as reported in [7]. Furthermore it is fairly straightforward to demonstrate empirically the utilisation capabilities of Ethernet. One of the origins of this "magic number" is possibly the (slotted) Aloha wireless protocol, which, due to the contention mechanism in the system, achieves a maximum throughput of 37 percent. Here, we carry out an analysis of the throughput of Aloha using J. We also demonstrate how throughput can be improved using carrier sensing techniques. The throughput S for slotted Aloha is the function of the offered load G and is given by the expression [66]:

$$S = Ge^{-G} \tag{1.1}$$

The mathematical expression in Equation (1.1) can be implemented by the J command line:

```
   (*^@-) 0 1 2 3
0 0.367879 0.270671 0.149361
```

From the example above, the indented line is the J command that is entered by the user (the J prompt is a three-space indentation). The sequence of characters `*^@-` (enclosed in brackets) form a function; in this case they define the relationship between S and G in Equation (1.1). The list of values that follows is the argument passed by the function (in this case the argument represents various values of G). The function is applied to each value in the argument list and outputs a corresponding value of S on the nonindented line below. The symbols `^`, `*` and `-` are arithmetic operators and represent the exponential function, multiplication and negation respectively. The `@` primitive is called a *conjunction* in J and acts as a sequencing operator. Without going into detail at this point, think of `@` as an *apply* operator. Thus, the exponential function is applied to the negated values of the arguments (G). The result of this operation is then multiplied by the argument values. J composition rules may not seem intuitive at first, but they will be covered in more detail later. It can be seen that the throughput reaches a maximum of approximately $S \approx .0.37$ for $G = 1$. For pure Aloha (nonslotted), the maximum achievable throughput is even lower:

$$S = Ge^{-2G} \tag{1.2}$$

Equation (1.2) can be realised by the J expression below:

```
   (*^@-@+:) 0 0.5 1 2 3
0 0.18394 0.135335 0.0366313 0.00743626
```

where `+:` is the *double* primitive. Pure Aloha achieves maximum throughput, $S \approx 0.18$ when the offered load $G = 0.5$.

Carrier sense multiple access (CSMA) techniques achieve greater efficiency than Aloha protocols. The fundamental problem with Aloha is that stations send data whether the carrier is busy or not. Transmitting on a busy channel results in collisions and, therefore, low throughput levels. Systems that use CSMA "sense" the carrier before transmitting. If the carrier is idle, then the station transmits; otherwise it backs off. Here, we analyse the *nonpersistent* varaiant of CSMA, where the throughput is given by:

$$S = \frac{G}{1+G} \tag{1.3}$$

The J expression for Equation (1.3) can be implemented as:

```
   (%>:) 0 1 2 3
0 0.5 0.666667 0.75
```

where `%` and `>:` are the division and increment operators. We can see from the results that CSMA yields a higher throughput than Aloha. A nonpersistent CSMA station in back-off, defers a packet transmission for a random period after a busy carrier becomes idle. This is to avoid two (or more) stations which have backed off, from transmitting at the same time (causing a collision). Ethernet is a *1-persistent* CSMA protocol, which does not defer a random amount of time (transmits immediately the carrier becomes idles). While 1-persistent CSMA yields a lower theoretical throughput than nonpersistent, Ethernet employs collision detection, which make higher throughput levels possible. Indeed, Metcalfe and Boggs [47] showed that (shared-media) Ethernet is capable of sustaining 95 percent utilisation.

The graph in Fig 1.1 shows the throughput curves for slotted Aloha, pure Aloha and nonperistent CSMA. It is possible to plot graph in J. The plotting functions have to be loaded:

```
   load 'plot' NB. load plot functions
```

The example below shows briefly how to plot a simple graph for the throughput of CSMA. Note, however, that J's graphing capabilties is outside the scope of this book. Refer to the *J User Manual* for more details on the plot command. For convenience assign G and S:

```
   G =: 0 1 2 3
   S =: (*^@-) G
```

Ensure that you are running J interpreter as *jw* rather than *jconsole*. Then plot S against G:

```
   plot G;S
```

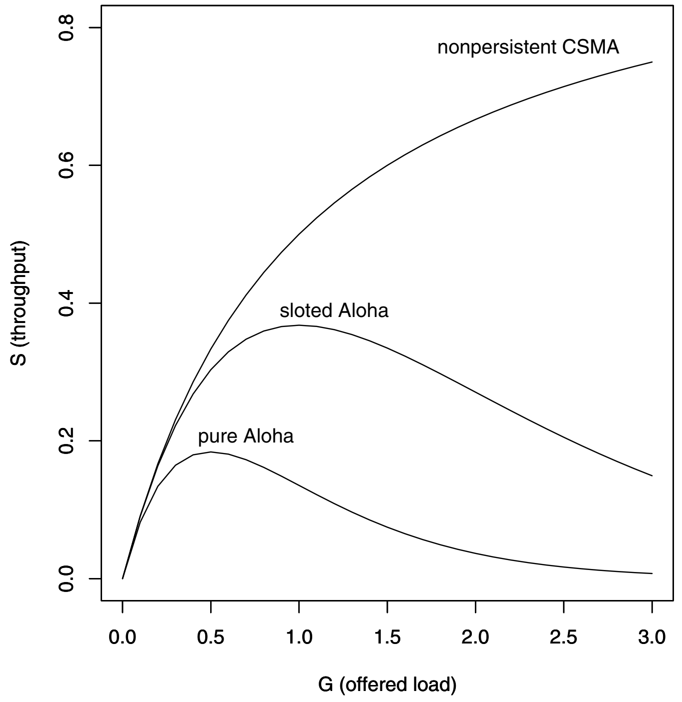

Fig. 1.1. Maximum achievable throughput for slotted Aloha

When setting utilisation levels, some sense of the time scales to which they apply is required. At the packet, cell or bit level time resolutions, a communications channel is either occupied or not occupied it cannot be *30 percent* occupied. Only at time resolutions greater than the packet (or cell) transmission time can communication links be *fractionally* occupied. When we view traffic flows graphically, we do so for a particular monitoring period and aggregation interval. For the purpose of illustration we present graphs (in Fig 1.2) showing network traffic for various levels of aggregation and monitoring periods. This data was taken from the University of Waikato Internet access link.[1]

For "calendar" time scales, such as days, weeks and years, traffic loads are non-stationary. The top-left graph in Fig 1.2 shows the volume of traffic aggregated over a weekly period for just over a year. Given that this link carries traffic from an academic institution, you can see two troughs at weeks 4 and 56, which are the Christmas/New Year periods (one year apart). There is also a trough centred around week 32, which is the intersemester break. It can be seen from the top-left graph that there is a rising trend in time, where the dashed line is a linear least-squares regression of the weekly traffic volumes.

[1] See `http://www.wand.net.nz/` for details.

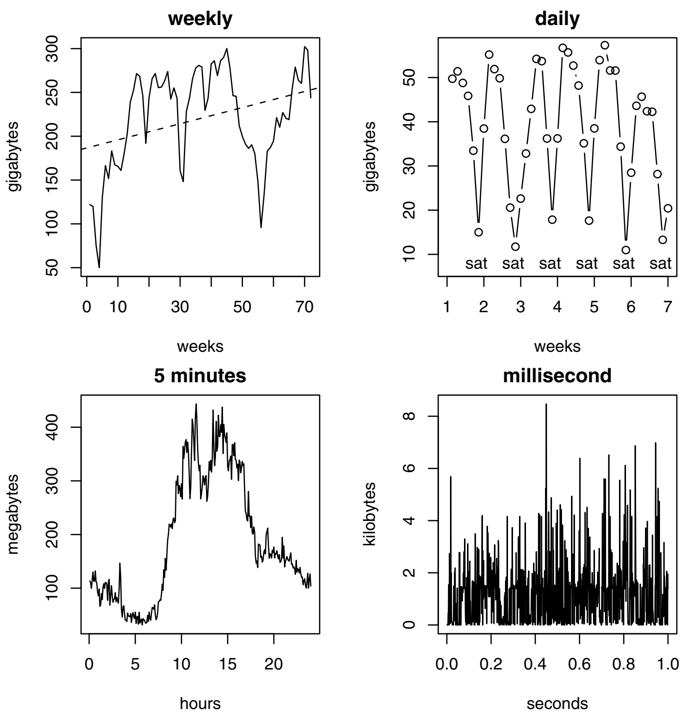

Fig. 1.2. Measured network traffic at various time scales

The top-right graph of Fig 1.2 shows the aggregate daily traffic volumes over a seven-week period. It shows the periodicity of peaks and troughs for weekday and weekend traffic respectively. There appears to be anomalously low traffic on Friday of the second week and Monday of the third week; this corresponds to the Easter bank holiday. The bottom-right graph (Fig 1.2) shows the five-minute traffic volumes over one day. The busy and off-peak periods of the day are evident.

Traffic flows on these time scales do not provide much useful insight into queue dynamics. Rather they are of more interest to the forecaster for the purpose of predicting future resource requirements or identifying peak and off-peak periods. While useful for long/medium-term forecasting, the time intervals over which traffic is aggregated are too long and the degree of "smoothing" does not convey any detailed burst-level behaviour.

The bottom-right graph of Fig 1.2 shows burst-level traffic. The traffic aggregates for each millisecond are shown for a one-second period. At these time scales, traffic is (typically) *stationary*. In traditional telephony, traffic arrival processes smooth exponentially quickly with the increase in aggregation interval. However, traffic patterns in packet data networks have complex structures, even for periods of stationarity, when packet arrivals can be highly correlated in time, exhibiting local trends and cycles [6, 13, 16, 28].

It is unlikely that there is some panacea utilisation figure that meets all capacity planning requirements and delivers QoS. It is not that the “30 percent” rule is wrong; in the absence of a better figure, it is as good as any. However, the QoS requirements will vary according to the users and the applications they run.

1.3 Traffic Management

The network processes packets according to three categories of traffic management policy:

- forwarding policy
- scheduling policy
- drop policy

If network providers do not apply any active traffic management, the respective forwarding, scheduling and drop policies are (typically): shortest-path/lowest-cost to destination, first-come-first-served (FCFS) and drop-from-tail. In a neutral network environment, all packets, regardless of user and service, are treated the same. Users compete with each other for resources. The (neutral) network provider does not protect users from one another, thus greedy users may get a larger proportion of resources, while the pain of congestion is shared equally. Network providers may elect to employ traffic management policies, such as Fair-Queuing [68] and Random Early Detect (RED), [20] in order to distribute congestion costs amongst users in proportion to their contribution to it.

Assuming dynamic routing protocols are being used (rather than static routing), the traditional IP forwarding policy is the shortest-path (or least-cost) to destination. Upon receiving the packet, each router examines the destination address in the IP header and looks up the next hop of the corresponding router. The “shortest-path” depends on the routing protocol. For example, RIP [12] uses number-of-hops as a routing metric, whereas OSPF [69] uses a cost metric based upon the speed of the link. Shortest-path routing algorithms can lead to traffic hotspots where links on the shortest-path are congested, while those that are not, are underutilised.

Load balancing can be achieved by manually manipulating the routing metrics. This, however, is not an ideal method, as a slight change in operating conditions can significantly degrade the stability of the network.[2]

A better method is to use *policy routing* [44], whereby packets are routed based on some attributes of the packet header other than the destination address (source

[2] Speaking from bitter experience.

address or port numbers for example). Packet routes are determined through network management functions and traffic engineering policies. Policy routing does present a number of problems. First of all, keeping track of routing policies currently in effect is administratively difficult. Secondly, each packet incurs extra processing overheads in determining the flow to which it belongs.

Protocols, such as Multiprotocol Label Switching (MPLS), [2] provide an infrastructure for the management of predefined paths through the network. MPLS routers attach *tags* to packets at the edge of the network according to the flow to which they belong. Flows can be identified by routers in the core with minimal overhead and packets can be forwarded/scheduled/dropped accordingly.

There are two QoS frameworks for IP networks, namely, Differentiated services (DiffServ) and Integrated services (IntServ), Differentiated services involve classifying flows at the edge of the DiffServ "cloud" by setting the Differentiated Services Code-Point (DSCP) in the IP header. Each code-point is associated with some class of service. Thus, flows *mapped* to a particular class are forwarded, scheduled and dropped according to a set of policies that implement the service class. DiffServ specifies a *coarse-grained* QoS. Given that the DSCP is only one byte, there is a significant restriction on the number of flows that can be represented. QoS, therefore, is applied to *aggregates* of flows.

In contrast, IntServ is a *fine-grained* architecture. Resources are reserved in the network for individual flows using a signalling protocol, namely, the Resource reSerVation Protocol (RSVP) [73]. The Traffic SPECification (TSPEC) specifies the parameters for a leaky-bucket algorithm, such as the token arrival rate and bucket depth. A flow $A = \{A(t), t = 0, 1, \ldots\}$, where $A(t)$ is the amount of traffic in the interval $[0, t]$, and conforms to the traffic specification function α if:

$$A(t) - A(s) \leq \alpha(t - s) \quad \forall t \geq s \tag{1.4}$$

The TSPEC parameters $\{M, p, r, b\}$ describe a dual leaky-bucket, where p is the peak rate, M is the maximum packet size, r is the sustainable rate and b is the burst tolerance. Thus $\alpha(t) = \min[pt + M, rt + b]$.

The Resource SPECification (RSPEC) specifies the resources requirements of the flow. For a given flow A, the resource function β should yield an output B, such that:

$$B(t) - A(s) \geq \beta(t - s) \quad \forall t \geq s \tag{1.5}$$

In order to support any degree of QoS, something must be known about the nature of the flows entering the network so that resources can be allocated accordingly. Consider a single server where packets arrivals are distributed according to a Poisson distribution with mean rate λ. Furthermore, the number of packets serviced per unit time is also Poisson distributed with mean rate μ. For a flow of traffic intensity $\rho = \lambda/\mu$, the probability that the backlog equals or exceeds n is ρ^n. For a traffic intensity of $\rho = 0.3$, the J expression shows that the backlog diminishes exponentially with n:

```
   0.3^(1 2 3 4 5)
0.3 0.09 0.027 0.0081 0.00243
```

The average delay for a single $M/M/1$ queue (where the M means Markovian) is given by the expression:

$$E[D] = \frac{1/\mu}{1-\rho} \tag{1.6}$$

If $\mu = 5$, then the delay increases exponentially with traffic intensity ρ:

```
   rho =: 0 0.2 0.4 0.6 0.8 0.9
   0.2 % 1-rho
0.2 0.25 0.333333 0.5 1 2
```

Traffic flows in data networks, however, are not necessarily Poisson. Representing traffic as a wide-sense increasing envelope curve lends itself to analytical treatment using network calculus methods. For example, in Equations (1.4) and (1.5), α and β represent arrival and service curves respectively. Given these two (wide-sense increasing) curves, the upper bound for the backlog can be derived by the network calculus result: $\max[\alpha(s) - \beta(s)]$, $\forall s \geq 0$. Bounds for delay and output traffic can also be derived using network calculus, as will be discussed in detail in this book.

1.4 Queue Analysis

We complete this chapter by carrying out an analysis of the queue dynamics of a working conserving link. We show how J can be used to build models of network systems for the purpose of analysis.

Consider a network traffic source transmitting packets across a communications link of capacity c. It is assumed that the scheduling policy is FCFS and the buffer queue is sufficiently large that packets are never dropped. Packets arriving at the interface of the communications link, form a *discrete-time* arrival process $a = \{a(t), t = 0, 1, 2, \ldots\}$. Packets depart from the link at each time interval at a maximum rate c. If $a(t) > c$, then packets in excess of c are buffered. In the next time interval $t+1$, the link attempts to clear the backlog $q(t)$ and then forwards any new arrivals $a(t+1)$ up to the bound c. The backlog is given by the Lindley equation [39] below:

$$q(t+1) = (q(t) + a(t+1) - c)^{+} \tag{1.7}$$

where the $(x)^{+}$ is $\max(0, x)$ and $q(0) = 0$. Here, we show how to implement the Lindley equation in J, such that we can examine the queuing dynamics of a work conserving link. The function requires two parameters; the initial queue size $q(0)$, and the arrival process over a (finite) sequence of discrete time intervals $t = 1, \ldots, N$. The J function *qnext* returns the backlog $q(t+1)$, and is defined thus:

```
   qnext =: 0: >. qprev + anext - c
```

Note that the terms: `0:`, `>.`, *anext*, +, *qprev*, − and *c* in the J expression above are all functions. J supports a *tacit* programming style; that is, there is no explicit reference to arguments. The built-in functions + and − are the addition and subtraction operators respectively. The *larger-of* function `>.`, is also a built-in funtions, and returns the argument with the greater value:

```
   0 >. 1  NB. max
1
```

Note that the `NB.` construct denotes a comment. In order to make the *qnext* more recognisable, we can define the *max* function, and redefine *qnext*, thus:

```
   max =: >.  NB. define max function
   qnext =: 0: max qprev + anext - c
```

The function `0:`[3] is one of J's *constant* functions, it returns a value zero, irrespective of its arguments:

```
   0: 1
0
```

The function *anext* ($a(t+1)$) returns the number of arrivals, *qprev* returns the backlog ($q(t)$) and *c* is the capacity of the link. These functions will be defined shortly, but first we must introduce a few more built-in J functions. We cannot make explicit reference to arguments when programming tacitly, although we can access arguments through the the functions *left* `[` and *right* `]` which return the left and right arguments, respectively:

```
   0 1 2 [ 3 4 5  NB. return left argument
0 1 2
    0 1 2 ] 3 4 5 NB. return right argument
3 4 5
```

We use `[` and `]` to extract a and q, respectively. However, we need the values of $a(t+1)$ and $q(t)$ from these vectors. The term $q(t)$ is relatively straight forward, as it is the last element of vector q. The *tail* function `{:` returns the last element of a list:

```
   {: 0 1 2 3 4 5
5
```

[3] In addition to `0:` there are a number of corresponding integer *constant* verbs that range from -9 to 9.

The definition of the function *qprev* is:

```
   qprev =: {:@]
```

Extracting the term $a(t+1)$ is more involved as we have to compute the index of its position in the list a. We can calculate the value of $t+1$ from the length of the backlog vector. The *tally* function # gives the length of a list:

```
   # 0 1 2 3 4 5
6
```

The value of $t+1$ is given by the function:

```
   tnext =: #@]
```

The function *ind* decrements (with `<:`) the value returned by *tnext*:

```
   ind =: <:@tnext
```

In J, elements in a list are indexed from zero, hence the need to decrement the value of *tnext*. We can now extract the $(t+1)^{th}$ element from a using the *from* function `{` thus:

```
   anext =: ind { [
```

Note that we do not need to use the "apply" operator `@`. This is due to the *composition* rules of J. Finally we define the function c. In order to simplify this introductory example, we use one of the *constant* functions. For a communications link with a capacity of one, we use the expression:

```
   c =: 1:
```

This implementation of the Lindley equation is of somewhat limited usefulness as we can only analyse link capacities of integer values between one and nine. Ideally, we should prefer to pass the value of the capacity as parameter (along with a). However, this would add to the complexity of the final function. Later on in the book we will present a more flexible implementation of the Lindley equation where c is passed as a parameter. We can test *qnext* by passing $a(t+1)$ and $q(t)$ as arguments:

```
   1 qnext 0 NB. a(t+1) = 1, q(t) = 0
0
   0 qnext 1 NB. a(t+1) = 0, q(t) = 1
0
   1 qnext 1 NB. a(t+1) = 1, q(t) = 1
0
   2 qnext 0 NB. a(t+1) = 2, q(t) = 0
1
```

Excessive delays and dropped packets are a natural outcome of a best-effort service. Any impact on an individual user's "quality of service" is caused by other users sharing the same networking resources. A degradation of service is a result of congestion periods in the network.

Ideally, a network provider would wish to utilise network resources fully. The network user, however, judges performance in terms of response time. High throughput and low response time are conflicting requirements. This is primarily due to users' demands for resources being unevenly distributed over time. We can use the Lindley equation to explore queuing dynamics for various traffic types.

The function *qnext* computes $q(t+1)$; however an algorithm is required to perform the operation over an entire arrival sequence. In J, this is fairly straightforward and requires none of the conventional iteration constructs, such as *for* or *while* loops. Our algorithm, which we will call *lindley*, is simply:

```
   lindley =: ] , qnext
```

We pass the arrival sequence as the left argument to *lindley* and the backlog history up to time t as the right argument. The backlog history up to time $t+1$ is returned. Consider a determinstic arrival process a_1 of rate three:

```
   a1 =: 3 3 3 3 3 3 3 3 3 3
```

We set the capacity of the link accordingly:

```
   c =: 3:
```

The first iteration of the *lindley* function, with backlog history $q(0) = 0$ is:

```
   a1 lindley 0
0 0
```

For subsequent iterations, we can use the J power construct `^:`. Functions can be iterated N times by raising them to the power N, that is:

$$f^3(x(0)) = f(f(f(x(0))))$$

Thus we can run the *lindley* function for ten iterations with the following command-line:

```
   a1 lindley^:(10) 0
0 0 0 0 0 0 0 0 0 0 0
```

Not surprisingly, given the deterministic arrival rate $a(t) = 3, \forall t$, the backlog is zero. Furthermore, the link is fully occupied. It would not be advisable to set the link capacity any lower, as the backlog would grow, assuming an infinite queue, without bound:

```
   c =: 2:
   a1 lindley^:(10) 0
0 1 2 3 4 5 6 7 8 9 10
```

Here we examine another, deterministic traffic arrival process. The "average" arrival rate is still three, however; traffic arrives in bursts of six, followed by a period of inactivity:

```
   c =: 3:   NB. set the capacity back to 3
   a2 =: 6 0 6 0 6 0 6 0 6 0
   a2 lindley^:(10) 0
0 3 0 3 0 3 0 3 0 3 0
```

Traffic in excess of the capacity $c = 3$ is buffered until the next period. The next time interval constitutes an off-period, so the link is able to clear the backlog. For deterministic flows, setting the capacity for a given QoS is trivial. For example, if for a_2 we wish to set an upper bound of two for the backlog, then c needs to be set to 4. Network traffic, however, tends to be stochastic. Given burst of traffic, it is not so easy to predict that an off (or quiet) period will follow (or even if one will occur soon). Consider the arrival sequence a_3:

```
   a3 =: 3 5 2 4 5 2 3 1 3 2
```

The J expression below confirms that the average arrival rate of a_3 is three:

```
   +/ a3 % 10
3
```

The `+/` is the summation function in J and `%` is *divide*. Run the *lindley* algorithm:

```
   a3 lindley^:(10) 0
0 0 2 1 2 4 3 3 1 1 0
```

A backlog builds up during persistent bursts (when $a(t) > c$). Eliminating the backlog is relatively straightforward. We merely need to set the capacity of the link to the peak rate of the flow, thus:

```
   max a3
5
   c =: 5:
   a3 lindley^:(10) 0
0 0 0 0 0 0 0 0 0 0 0
```

However, a zero backlog appears at a cost of overprovisioning the network. For much of the time, network resources remain idle. If some backlog can be tolerated and the network provider can accept some amount of idle capacity, then a compromise solution may be reached by:

```
   c =: 4:
   a3 lindley^:(10) 0
0 0 1 0 0 1 0 0 0 0 0
```

1.5 Summary

The purpose of network performance analysis is to investigate how traffic mangement mechanisms deployed in the network affect the allocation of resources amongst its users and the performance they experience. We can construct models of traffic management mechanisms and observe how they perform by applying them to some flow of network traffic. The stochastic nature of network traffic presents the network capacity planner with a challenge. Clearly, the capacity must lie between the average and the peak of the traffic flow. Precisely where, is determined by the QoS requirements. This book is about addressing these challenges.

2

Fundamentals of the J Programming Language

In this chapter, we present the basic concepts of J. We introduce some of J's built-in functions and show how they can be applied to data objects. The pricinpals presented in this book are supported by examples. The reader therefore, is strongly advised to download and install a copy of the J interpreter from `www.jsoftware.com`.

2.1 Data Objects

Here, we give a brief introduction to data objects in J. Data objects come in a number of forms: scalars, vectors, matrices and higher-dimensional arrays. Scalars are single numbers (or characters); the examples below show the values assigned to variables names:

```
   i =: 3         NB. integer
   j =: _1        NB. negative integer
   x =: 1.5983    NB. real
   y =: 22r7      NB. rational
   z =: 2j3       NB. complex number
   m =: _         NB. infinity
   n =: __        NB. negative infinity
   v =: 5x2       NB. exponential notation 5 * exp(2)
```

Note that negative numbers are denoted by an underscore (_) preceding the value rather than by a hyphen (-). An underscore on its own denotes infinity ∞, and two underscores denotes negative infinity $-\infty$. Variables are not limited to numerical values:

```
   c =: 'hello world'
```

Variables can be evaluated by entering the variable name at the command prompt:

```
   c
hello world
```

Scalars are called *atoms* in J or *0-cells*, and vectors are called lists or 1-cells. The sequence of numbers below is an example of a list and is assigned to a variable name:

```
   dvc =: 1 1 2 3 5 8 13 21 34 55
   dvc
1 1 2 3 5 8 13 21 34 55
```

The `i.` verb generates ascending integers from zero $n-1$, where n is the value of the argument. For example:

```
   z1 =: i.10 NB. generate list 0 to 9
   z1
0 1 2 3 4 5 6 7 8 9
```

The associated verb `i:` generates integers from $-n$ to n, thus:

```
   z2 =: i:5  NB. generate list -5 to 5
   z2
_5 _4 _3 _2 _1 0 1 2 3 4 5
```

Matrices (tables in J) are 2-cell objects. Here is a table of complex numbers:

```
   j =: (i.6) j./ (i.6)
   j
0 0j1 0j2 0j3 0j4 0j5
1 1j1 1j2 1j3 1j4 1j5
2 2j1 2j2 2j3 2j4 2j5
3 3j1 3j2 3j3 3j4 3j5
4 4j1 4j2 4j3 4j4 4j5
5 5j1 5j2 5j3 5j4 5j5
```

Matrices (of ascending integers) can be generated with the `i.` verb. The example below shows a 3×2 matrix:

```
   i. 2 3 NB. generate a matrix
0 1 2
3 4 5
```

Higher-dimensional arrays are also possible. The expression below generates an array of reciprocals with ascending denominators:

```
   %i. 2 4 3
        _         1       0.5
 0.333333      0.25       0.2
 0.166667  0.142857     0.125
 0.111111       0.1 0.0909091

0.0833333 0.0769231 0.0714286
0.0666667    0.0625 0.0588235
0.0555556 0.0526316      0.05
 0.047619 0.0454545 0.0434783
```

This 3-cell data object is a three-dimensional array with three columns, four rows and two *planes*, where the planes are delimited by a blank line.

Scalars/atoms, vector/lists and matrices/tables are just special instances of arrays with respective ranks zero, one and two. Table 2.1 shows the J terms for data objects and their mathematical equivalents. Throughout this book, we will use the J and mathematical terms interchangeably.

J term	Mathematical term	Dimension	Object type
atom	scalar	0	0-cell
list	vector	1	1-cell
table	matrix	2	2-cell
array	array	n	n-cell

Table 2.1. J versus mathematical terms

2.2 J Verbs

In J, functions are called *verbs*. J has a number of built-in verbs/functions; for example, the basic arithmetic operators are: `+`, `-`, `*`, `%` and `^`, which are addition, subtraction, multiplication, division and power functions, respectively. Here are some (trivial) examples:

```
   2 + 3      NB. addition
5
   2 - 3      NB. subtraction
_1
   7 * 3      NB. multiplication
21
   2 % 7      NB. division: "%" is used instead of "/"
0.285714
```

```
      2^3      NB. power
8
```

In J, there is a subtle (but important) difference between `_1` and `-1`. The term `_1` is the value minus one, whereas `-1` is the negation function applied to (positive) one.

Arithmetic can also be performed on lists of numbers:

```
    2 % 0 1 2 3 4  NB. divide scalar by a vector
_ 2 1 0.666667 0.5
   3 2 1 0 - 0 1 2 3 NB. pointwise vector subtraction
3 1 _1 _3
```

The example above demonstrates how J deals with divide by zero. Any division by zero returns a _ (∞). In addition to the basic arithmetic operators, J has many more primitives, for example:

```
   +: 1 2 3 4      NB. double
2 4 6 8
   -: 1 2 3 4      NB. halve
0.5 1 1.5 2
   %: 1 4 9 16     NB. square root
1 2 3 4
   *: 1 2 3 4      NB. squared
1 4 9 16
   >: _1 0 1 2 3   NB. increment
0 1 2 3 4
   <: _1 0 1 2 3   NB. decrement
_2 _1 0 1 2
```

Verbs are denoted by either a single character (such as *addition* +) or a pair of characters (such as *double* `+:`). Some primitives do use alphabetic characters, for example, the *integers* verb `i.` and *complex* verb `j.`, which were introduced above.

2.3 Monadic and Dyadic functions

Each verb in J possesses the property of *valance*, which relates to how many arguments a verb takes. *Monadic* verbs take one argument (and are therefore of valance one), whereas *dyadic* verbs take two arguments (valance two).

Monadic verbs are expressed in the form: `f x`. The (single) argument x is passed to the right of the function f. In functional notation, this is equivalent to $f(x)$. In its dyadic form, f takes two arguments and are passed to the function on either side: `y f x`, equivalent to $f(y,x)$ in functional notation.

J's verb symbols are *overloaded*; that is, they implement two separate (often related, but sometimes inverse) functions depending upon the valance. We use the `%` primitive to demonstrate. We have already seen it used in its dyadic form as a *division* operator. However, in its monadic form, `%` performs a *reciprocal* operation:

```
   % 2    NB. used monadically is reciprocal
0.5
   3 % 2  NB. used dyadically is division
1.5
```

Let us look at a few more examples. The monadic expression `^x` is the *exponential* function of x: e^x. The dyad `y^x`, however, performs y to the power x, that is: y^x. To illustrate:

```
   ^ 0 1 2 3     NB. used monadically is exp(x)
1 2.71828 7.38906 20.0855
   2 ^ 0 1 2 3   NB. used dyadically is y^x
1 2 4 8
```

Used monadically `<:` performs a *decrement* function:

```
   <: 1 2 3 4 5 6
0 1 2 3 4 5
```

However as a dyad it performs *less-than-or-equal-to*[1]:

```
   4 <: 1 2 3 4 5 6
0 0 0 1 1 1
```

2.4 Positional Parameters

The meaning of a *positional parameter* is given by virtue of its relative position in a sequence of parameters. J does not really have a concept of positional parameters; however, we can pass positional parameters to functions as an ordered list of arguments. In this section, we introduce verbs for argument processing: *left* `[` and *right* `]`. These verbs return the left and right arguments, respectively:

```
   2 [ 3
2
   2 ] 3
3
```

[1] Similarly `>:` performs *increment* and *greater-than-or-equal-to*.

The *right* provides a convenient means of displaying the result of an assignment:

```
   ]x =: i.10
0 1 2 3 4 5 6 7 8 9
```

The *left* verb can be used to execute two expressions on one line:

```
   x =: i.10 [ n =: 2 NB. assign x and n
   x ^ n
0 1 4 9 16 25 36 49 64 81
```

The verbs *head* `{.` and *tail* `{:` return the first element and the last element of a list:

```
   {. x
0
   {: x
10
```

Conversely, *drop* `}.` and *curtail* `}:` remove the head and tail of a list and return the remaining elements:

```
   }. x
2 4 6 8 10
   }: x
0 2 4 6 8
```

The *from* verb `{` is used to extract a particular element (or elements) within a list, by passing the index of the required element in the list as a left argument:

```
   0 { x
0
   2 { x
4
   3 1 5 { x
6 2 10
   0 0 0 1 2 2 2 3 3 4 5 5 5 5 { x
0 0 0 2 4 4 4 6 6 8 10 10 10 10
```

Lists can be combined with *raze* `,.` and *laminate* `,:` in columns or rows, respectively. The J expressions below yield two matrices m_1 and m_2:

```
   ]m1 =: 1 2 3 4 ,. 5 6 7 8
1 5
2 6
3 7
4 8
```

```
   ]m2 =: 1 2 3 4 ,: 5 6 7 8
1 2 3 4
5 6 7 8
```

We cover higher-dimensional objects in more detail in Section 2.6. Here, we look briefly at applying the *from* verb to matrices:

```
   2 { m1
3 7
   1 { m2
5 6 7 8
```

Here, *from* returns the third row of m_1 in the first example and the second row of m_2 in the second example. If we wish to reference an individual scalar element, then we need to use *from* twice:

```
   0 { 1 { M2
5
```

In order to reference a column, we need to be able to change the *rank* of the verbs. The concept of rank will be covered in the next section. Two data objects can be concatenated with *ravel* (`,`), for example:

```
   v1 =: 1 2 3 4
   v2 =: 5 6
   ]v3 =: v1,v2
1 2 3 4 5 6
   0 3 4 5 { v3
1 4 5 6
```

This creates a single list of eight elements (v_3). There is no separation between the two original lists v_1 and v_2. If we wished to retain the separation of the two initial lists, then we combine them with the *link* verb `;`, for example:

```
   ]v4 =: v1;v2
+-------+---+
|1 2 3 4|5 6|
+-------+---+
```

The lists are "boxed" and therefore exist as separate data objects. We can reference the two lists in the usual way:

```
   0 { v4
+-------+
|1 2 3 4|
+-------+
```

The data object returned is a single element; we cannot get at any of the individual scalar elements in the box:

```
   1 { 0 { v4
|index error
|   1    {0{v4
```

Use *open* > to unbox the object:

```
   > 0 { v4  NB. unbox v1
1 2 3 4
   1 { > 0 { v4
2
```

There is a corresponding inverse function, namely the monadic verb *box* <, which "groups" elements:

```
   <1 2 3 4
+-------+
|1 2 3 4|
+-------+
```

Using the primitives described above, we define a number of functions for referencing positional parameters. These functions will be used a great deal in developing functions later in this book. Note that a couple of conjunctions are used here (`&` and `@`) will be covered later, in Section 3.1.4

```
lhs0 =: [         NB. all left arguments
lhs1 =: 0&{@lhs0  NB. 1st left argument
lhs2 =: 1&{@lhs0  NB. 2nd left argument
lhs3 =: 2&{@lhs0  NB. 3rd left argument

rhs0 =: ]         NB. all right arguments
rhs1 =: 0&{@rhs0  NB. 1st right argument
rhs2 =: 1&{@rhs0  NB. 2nd right argument
rhs3 =: 2&{@rhs0  NB. 3rd right argument
```

The functions `lhs0` and `rhs0` evaluate the left and right arguments, respectively. The other functions are programmed to return positional parameters, thus `lhs1` (respectively `rhs1`) returns the first positional parameter on the left-hand side (respectively right-hand side). We illustrate the use of positional parameters with the following example. Consider the classical M/M/1 queuing model given in Equation (1.6). We can write a function that takes the parameters μ and ρ as left-hand arguments and right-hand arguments, respectively:

```
   mm1 =: %@lhs1 % 1:-rhs0
   3 mm1 0.5 0.6 0.7 0.8 0.9
0.666667 0.833333 1.11111 1.66667 3.33333
```

2.5 Adverbs

The (default) behaviour of verbs can be altered by combining them with *adverbs*. We have already encountered an adverb with the summation function `+/`. The application of `/` causes + to be inserted between the elements of the argument, in this case, the individual (scalar) numbers in the list.

```
   +/ i.6                  NB. as we've seen before
15
   0 + 1 + 2 + 3 + 4 + 5   NB. and is equivalent to this
15
```

The dyadic case results in a matrix of the sum of the elements of the left argument to each element of the right argument.

```
   (i.6) +/ (i.6)
0 1 2 3 4  5
1 2 3 4 5  6
2 3 4 5 6  7
3 4 5 6 7  8
4 5 6 7 8  9
5 6 7 8 9 10
```

The *prefix* adverb `\` causes the data object to be divided into sublists that increase in size from the left; the associated verb is then applied to each sublist in turn. We can see how the sublist is generated using the *box* verb:

```
   <\ i.5
+-+---+-----+-------+---------+
|0|0 1|0 1 2|0 1 2 3|0 1 2 3 4|
+-+---+-----+-------+---------+
```

A cumulative summation function can be implemented using the *insert* and *prefix* verbs:

```
   +/\ i.6
0 1 3 6 10 15
```

This function will be useful later on when we wish to convert interval traffic arrival processes to cumulative traffic arrival processes. The *suffix* `\.` operates on decreasing sublists of the argument:

```
   <\. i.6
+-----------+---------+-------+-----+---+-+
|0 1 2 3 4 5|1 2 3 4 5|2 3 4 5|3 4 5|4 5|5|
+-----------+---------+-------+-----+---+-+
```

The monadic *reflexive* adverb ~ *duplicates* the right-hand argument as the left-hand argument. So the J expresssion `f~ x` is equivalent to `x f x`, for example:

```
   +/~ i.6   NB. (i.6) +/ (i.6)
0 1 2 3 4  5
1 2 3 4 5  6
2 3 4 5 6  7
3 4 5 6 7  8
4 5 6 7 8  9
5 6 7 8 9 10
```

The ~ verb also has a dyadic form (*passive*). This means that the left and right-hand side arguments are swapped; that is, `y f x` becomes `x f y`, as an illustration:

```
   2 %~ i.6  NB. equivalent to (i.6) % 2
0 0.5 1 1.5 2 2.5
```

2.6 Rank, Shape and Arrays

Arithmetic can be performed between a scalar and a list or between two lists, for example:

```
   2 * 0 1 2 3 4 5
0 2 4 6 8 10
   0 1 2 3 4 5 + 1 2 3 4 5 6
1 3 5 7 9 11
```

Notice that the lists have to be the same length; otherwise the J interpreter throws a "length error":

```
   9 8 - 0 1 2 3 4 5
|length error
|   9 8    -0 1 2 3 4 5
```

J can also perform arithmetic on higher-dimensional objects. In this section, we introduce arrays as well as the concepts of *rank* and *shape*. Rank is synonymous with dimensionality; thus a two-dimensional array has rank two, a three-dimensional array has rank three. Verbs have rank attributes which are used to determine at what rank level they should operate on data objects. We will explore this later. First, let us consider at how we can define array objects by using the dyadic *shape* verb `$`:

```
   ]x2 =: 2 3 $ i.6
0 1 2
3 4 5
```

As we have already seen, we could have defined this particular array, simply by:

```
   i. 2 3
0 1 2
3 4 5
```

This is fine for defining an array with ascending integers (as returned by `i.`), but if we wanted to form an array using some arbitrary list of values, then we need to use `$`. We will continue to use the `$` method, although we acknowledge that it is not necessary, as the data objects used in these examples are merely ascending integers. The shape is specified by the left arguments of `$` and can be confirmed using the `$` in its monadic form:

```
   $ x2
2 3
```

The data object x_2 is a (3×2) two-dimensional array, or, in J terms, a rank two object (of shape `2 3`). Arithmetic can be applied in the usual way. This example shows the product of a scalar and an array:

```
  2 * x2
0 2  4
6 8 10
```

Here we have the addition of two arrays (of the same shape):

```
   x2 + (2 3 $ 1 2 3 4 5 6)
1 3  5
7 9 11
```

J can handle this:

```
    2 3 + x2
2 3 4
6 7 8
```

But apparently not this:

```
   1 2 3 + x2
|length error
|   1 2 3    +x2
```

J of course, *can* handle this, but we need to understand more about the *rank* control conjunction " which will be covered in Section 3.1 below. Consider a $3 \times 2 \times 2$ array:

```
   ]x3 =: 2 2 3 $ i.12
0  1  2
3  4  5

6  7  8
9 10 11
```

x_3 is a three-dimensional array and, therefore, of rank three. J displays this array arranged into two *planes* of two rows and three columns, where the planes are delimited by the blank line. We can confirm the structure of x_3 by using `$` as a monad, where it peforms a *shape-of* function:

```
   $ x3
2 2 3
```

Now, let us apply summation to x_3:

```
   +/ x3
 6  8 10
12 14 16
```

Here, the individual elements of the two *planes* have been summed; that is:

$$\begin{aligned}\begin{pmatrix}0&1&2\\3&4&5\end{pmatrix}+\begin{pmatrix}6&7&8\\9&10&11\end{pmatrix}&=\begin{pmatrix}0+6&1+7&2+8\\3+9&4+10&5+11\end{pmatrix}\\&=\begin{pmatrix}6&8&10\\12&14&16\end{pmatrix}\end{aligned}$$

It is important to understand why `+/` sums across the planes rather down the columns or along the rows. First consider this example:

```
   % x3
       _        1       0.5
0.333333     0.25       0.2

0.166667 0.142857     0.125
0.111111      0.1 0.0909091
```

Aside from the difference between the arithmetic functions `+/` and `%` perform, they also operate on the argument in a different way. Where as `+/` operated on the two planes, here `%` is applied to each individual scalar element. The difference in the behaviour of the two verbs `+/` and `%` is governed by their respective rank attributes. We can query the rank attribute of verbs with the expressions below:

```
   % b. 0
0 0 0
   +/ b. 0
_ _ _
```

Three numbers are returned. The first number (reading left to right) is the rank of the monad form of the verb. The second and third numbers are the ranks of the left and right arguments of the dyadic form of the verb. When a verb performs an operation on an object, it determines the rank of the cell elements on which it will operate. It does this by either using the rank (dimension) of the object or the rank attribute of the verbs, whichever is smaller. In the example above, x_3 has a rank of three, and the (monadic) rank attribute of `%` is zero. So `%` is applied to x_3 at rank zero. Thus it is applied to each 0-cell (scalar) element. However, the rank attribute of `+/` is infinite, and, therefore, the rank, at which the summation is performed, is three. Thus, `+/` applies to each 3-cell element of x_3, resulting in a summation across the planes.

Consider another example. We define the data object x_0 as a list of six elements and then apply the fork `($,#)` which (simultaneously) returns the shape and the number elements.

```
   ]x0 =: i.6
0 1 2 3 4 5
   ($;#) x0
+-+-+
|6|6|
+-+-+
```

Here both # and $ return six as x_0 consists of six atoms, or 0-cell elements. Now try this on x_2 which was declared earlier:

```
   ($;#) x2
+---+-+
|2 3|2|
+---+-+
```

The resultant shape is as expected but the number of elements returned by *tally* may not be. To make sense of the result, we need to know what “elements” the *tally* is counting: 0-cell, 1-cell or 2-cell? This depends upon the rank at which # is operating. The data object x_2 is clearly rank two. The command-line below shows us that the (monadic) verb attribute of # is infinite:

```
  # b. 0 NB. monadic rank attribute is infinite
_ 1 _
```

In this particular case we may ignore the dyadic rank attributes. Tally (#) is applied to the 2-cell elements which are the rows. Consider this example:

```
   ]x1 =: 1 6 $ i.6
0 1 2 3 4 5
   ($;#) x1
+---+-+
|1 6|1|
+---+-+
```

Data objects x_0 and x_1 may appear the same but they are actually different by virtue of their shape and rank. x_1 is a (6×1) two-dimensional array, and, therefore, of rank two (with shape `1 6`). It also has only one element (one row) because # still operates on the 2-cell elements. In contrast, x_0 is a list of six 0-cell elements. In actual fact, x_0 is equivalent to y_0, defined below:

```
   ]y0 =:  6 $ i.6
0 1 2 3 4 5
   x0 = y0  NB. x0 and y0 are equivalent
1 1 1 1 1 1
   x0 = x1  NB. but x0 and x1, as we know, are not
|length error
|   x0    =x1
```

The difference between x_0 and x_1 becomes more apparent when we perform some arithmetic operation on them:

```
   x1 - x0
|length error
|   x1    -x0
```

The interpreter is trying to subtract the first element from x_1 from the first element of x_0, then subtract the second element from x_1 from the second element of x_0, and so on. However, while x_0 has six 0-cell elements, x_1 only has one element, which is a 2-cell. However it does not seem unreasonable to want to perform arithmetic operations on x_0 and x_1, as they both contain six numbers. We can control the rank attribute of a verb, thereby enabling us to perform arithmetic on both x_0 and x_1. We will return to this example in Chapter 3 when we cover conjunctions.

2.7 Summary

In this chapter we have introduced some of the basic concepts of programming in J. J functions are called verbs. The primitive verbs are (mostly) designated by a single punctuation character (+) or a pair of punctuation characters (`+:`), though a few use alphabetic characters (`i.`). Verbs can be monadic (one argument) or dyadic (two arguments). Data objects have properties of rank (dimension) and shape. All objects can be thought of as arrays. Atoms (0-cell), lists (1-cell) or tables (2-cell) are merely special instances of arrays ranked (dimensioned) zero, one and two, respectively. Furthermore any n ranked array can be thought of as a *list* of $n-1$ cell objects. Note that, an atom has an empty shape and is different from a one-item list. Furthermore, a list is different from a $1 \times n$ table. Verbs have rank attributes that determine at what *cell* level they operate.

3

Programming in J

New verbs can be defined from existing ones by combining them in sequences called *phrases*. In this chapter, we focus on programming in J and describe in detail the rules of *composition*. We begin by using J to explore the concept of z-transforms. This exercise serves as a brief introduction to composition contructs and *conjunctions*. Subsequent sections cover these concepts in greater detail.

The z-transform converts discrete signals in the time domain into a complex frequency time domain. The z-transform involves the summation of an infinite series of reciprocals of z^{-n} [41]:

$$\mathcal{Z}\{x[n]\} = \sum_{n=0}^{\infty} x[n]z^{-n} \tag{3.1}$$

where z can be complex and $x[n]$ is a sequence or function indexed by n. Consider the unit step function $u[n]$:

$$u[n] = \begin{cases} 1 & \text{if } n \geq 0 \\ 0 & \text{if } n < 0 \end{cases} \tag{3.2}$$

The corresponding z-transform for $u[n]$ is:

$$\mathcal{Z}\{u[n]\} = \frac{z}{z-1}, \qquad |z| > 1 \tag{3.3}$$

The J expression below implements the z-transform in Equation (3.3) for $u[n]$:

```
   z =: 2 3 4 5 6 7 NB. define |z| > 1
   (%<:) z
2 1.5 1.33333 1.25 1.2 1.16667
```

The expression above consists of a two verb phrase (enclosed in brackets) followed by an argument z. The pair of verbs, namely, `%` and `<:`, form a compositional construct called a *hook*. The J interpreter parses the phrase right to left. The argument

z undergoes a transformation by the first verb (reading from the right). The second verb is then applied to the argument *and* the transformation. Thus, all the values in z are decremented by `<:`, then the *division* operator `%` is applied, where the numerator is z and the denominator is the result of the decrement operation ($z - 1$).

The *step* verb can be implemented by the mathematical function *less-than-or-equal-to*, where a value of one is returned if true, and zero otherwise. In J, the <, `+.` and = are the *less-than*, *or* and *equals-to* verbs, respectively. Thus, the phrase in the expression below implements the step[1] function for $n = 3$:

```
   n =: i.10
   3 (<+.=) n
0 0 0 1 1 1 1 1 1 1
```

A sequence of three verbs forms a *fork*. The arguments undergo transformations by both the left and right verbs (< and =, is this case). The middle verb (`+.`) is applied to the resultant tranformations. Functions in J may be *unnamed* phrases, enclosed in brackets, as in: `(<+.=)`. For convenience, however, we may assign the phrase to a name, defining a new verb thus:

```
   step =: < +. = NB. less than or equal to
```

The unit step function $u[n]$ in Equation (3.2) is realised by invoking *step* with a left argument of zero. The unit step function can be defined by attaching the constant `0` to the *step* function:

```
   ustep =: 0&step
   ustep n
1 1 1 1 1 1 1 1 1 1
```

In J, `&` is a *conjunction*, and is used to *bind* nouns to verbs. Mathematically, *ustep* implements the function *less-than-or-equal-to-zero*, The expression below yields the z-transform for $u[n]$, confirming the result in Equation (3.3). The tranformation in Equation (3.1) is a summation to $n = \infty$. For practical purposes, instead of $n = \infty$, we choose a sufficiently large value for n:

```
   n =: i.100
   +/ (ustep * z&^@-) n
2 1.5 1.33333 1.25 1.2 1.16667
```

Note the use of the *bind* conjunction again, whereby the term `z&^` forms a *z-to-the-power* verb. Another conjunction is also used, namely *atop* `@`. The composition of one verb *atop* another, constitutes a transformation of a transformation. Here, the *z-to-the-power* verb is applied *atop* the negation `-` yielding the mathematical term z^{-n}.

[1] We implement the *step* function in this way merely for illustration, J provides a built-in *less-than-or-equal-to* verb: `<:`.

3.1 Verb Composition

In the introductory example above, we briefly demonstrated the rules of composition. We showed how verbs are combined, sometimes with conjunctions, to implement new verbs. In this section, we cover verb composition in more detail and present the concept of *tacit* programming.

3.1.1 Hooks

A sequence of two verbs forms a *hook*. If f_1 and f_2 are verbs, then the monadic phrase: `(f1 f2) x`, where x is an argument, is equivalent to the function expression: $f_1(x, f_2(x))$. The verb f_2 performs a monadic operation on x. Then f_1 performs an operation on the result of x *and* $f_2(x)$. The function f_1 behaves as a dyad, as it operates on two "arguments." To illustrate this we provide a number of examples. The hook below implements the expression $x^2 + x$:

```
   x =: i.6    NB. assign a vector of numbers to x
   (+*:) x
0 2 6 12 20 30
```

Each value in x is squared (by `*:`). This result is then added to the vector x; thus the expression above is equivalent to:

$$\{0+0, 1+1, 4+2, 9+3, 16+4, 25+5\}$$

In the following example, each value of x is doubled (`+:`) and then multiplied by x, which is equivalent to $x \times 2x$, or $2x^2$:

```
   (*+:) x  NB. double then multiply
0 2 8 18 32 50
```

The expression $x - 1/x$ is implemented by:

```
   (-%) x  NB. subtract reciprocal from each value
0 1.5 2.66667 3.75
```

The dyadic form of the hook is given by: `y (f1 f2) x`, where x and y are arguments. Here, f_2 performs a monadic operation on x. Then f_1 performs a dyadic operation on y and the result of $f_1(x)$. Expressed in functional notation, the dyadic hook is equivalent to: $f_1(y, f_2(x))$.

The example below implements the expression $y + \sqrt{x}$:

```
   y =: 2
   y (+%:) x
2 3 3.41421 3.73205 4 4.23607
```

3.1.2 Forks

A monadic *fork* is a three-verb phrase `(f1 f2 f3) x`. Expressed in functional notation, this is equivalent to $f_2(f_1(x), f_3(x))$. Verbs f_1 and f_3 operate on x as monads. The results of $f_1(x)$ and $f_3(x)$ are then operated on by f_2 as a dyad. The dyadic form of the fork is: `y (f1 f2 f3) x` which in functional notation is: $f_2(f_1(y, x), f_3(y, x))$. The verb f_1 operates on y and x, as a dyad, as does f_3. The results of $f_1(y, x)$ and $f_3(y, x)$ are then operated on by f_2 as a dyad.

A ubiquitous (but nevertheless good) example of a (monadic) fork is the arithmetic mean function:

```
   (+/%#) x    NB. arithmetic mean
2.5
```

The tally verb `#` returns the number elements in the vector x (in this case 6). The `+/` returns the summation of x. Then the verb `%` is applied to the result of verbs `+/` and `#`, thus dividing the sum of the elements by the number of elements. Note that, here, `%` is applied dyadically, so it performs a division function rather than the (monadic) reciprocal form of the verb.

The example below shows a dyadic fork. The expression $yx - y/x$ can be implemented by:

```
   y (*-%) x
___ 0 3 5.33333 7.5 9.6
```

3.1.3 Longer Phrases

To reiterate, phrases are evaluated right to left. The hook and fork rules of composition are applied. Consider the example below:

```
   (-+/%#) x
_2.5 _1.5 _0.5 0.5 1.5 2.5
```

A fork is formed by the three primitives: `+/%#` which, as we saw above, implement the arithmetic mean. This, in turn, forms a hook with `-`. We can make the example a little clearer if we define an arithmetic mean verb:

```
   mean =: +/%#   NB. define mean
   (-mean) x      NB. deviation from the mean
_2.5 _1.5 _0.5 0.5 1.5 2.5
```

We can see that `-` and *mean* form a hook, which subtracts $\overline{x}$ (the mean of x) from each element of x; that is, $x_i - \overline{x}$. The phrase *-mean* implements the *deviation-from-mean* function.

3.1.4 Conjunctions

In this section, we discuss verb composition with conjunctions. The *bond* verb `&` is used to combine verbs and nouns to form new verbs. For instance, if `^.` is used monadically, it performs the natural logarithm function (ln); dyadically, however, it performs the logarithm to the base of left argument. This is illustrated in the example below:

```
   ^. 1 2 4 10 100     NB. natural log
0 0.693147 1.38629 2.30259 4.60517
   2 ^. 1 2 4 10 100   NB. log to the base 2
0 1 2 3.32193 6.64386
   10 ^. 1 2 4 10 100  NB. log to the base 10
0 0.30103 0.60206 1 2
```

For convenience, we can define the verb $\log_2$ by binding the 2 to verb `^.`, that is:

```
   log2 =: 2&^.    NB. define log to the base 2
```

Here the literal 2 is bonded to `^.` to form a new verb which, used monadically, gives:

```
   log2 1 2 4 10 100
0 1 2 3.32193 6.64386
```

Similarly, $\log_{10}$ can be defined:

```
   log10 =: 10&^.  NB. define log to the base 10
```

For readability we can define the natural logarithm:

```
   ln =: ^.       NB. define natural log
```

The `&` conjunction can also be used to combine verbs. Used in this way it is called the *compose* conjunction. In functional notation, the dyadic phrase `(f1&f2) x`, is equivalent to $f_1(f_2(x), f2(x))$. For the dyadic form of the phrase: `y (f1&f2) x` the functional equivalent is $f_1(f_2(y), f_2(x))$. Consider the use of *not* verb `-.` in the example below which produces a list on integers between three and nine:

```
   (i.10) -. (i.3)
3 4 5 6 7 8 9
```

This can be achieved with the phrase below, where we pass the bounds of the list as parameters, though the value of the left argument is not included in the list. Thus:

```
   10 (-.&i.) 3
3 4 5 6 7 8 9
```

The *to* verb, defined below, is a useful function for generating a list of integers from the value of the left argument to the value of the right agrument:

```
   to =: -.&i.~,]
   3 to 10
3 4 5 6 7 8 9 10
```

The *passive* adverb allows us to reverse the arguments and the *right* verb `]` ensures the right argument is included in the list.

The *atop* conjunction `@` (and related *at* conjunction `@:`) provide a means of combining verbs in sequence. Consider the monadic phrase in the form: `(f1@f2) x`. The verb f_2 is applied monadically to x, then f_1 is applied monadically to the result. In functional notation, this is equivalent to: $f_1(f_2(x))$.

Suppose we wish to compute the arithmetic mean and then negate the result. We cannot use the expression $-mean$, as this forms a *hook* between the `-` and *mean* verbs. As demonstrated above, this returns the deviation from the mean; that is $x_i - \overline{x}$, whereas the operation we actually want is $-\overline{x}$. The *hook* causes the `-` to behave dyadically and thus performs *subtraction*. The *atop* conjunction `@` alters the compositional rules between `-` and *mean*, such that `-` is applied monadically to the result of *mean*. The phrase below gives the desired result:

```
   (-@mean) x
_3.5
```

If the *at* conjunction is used instead, we get, in this case, the same result:

```
   (-@:mean) x
_3.5
```

While the *atop* and *at* are similar, they do have a subtle but important difference. Suppose we wish to sum a list of reciprocals:

$$\sum_{i=1}^{i=n} \frac{1}{x_i} \tag{3.4}$$

In the example below `+/` is applied atop `%`, thus:

```
   (+/@%) x
1 0.5 0.333333 0.25 0.2 0.166667
```

However, this has not performed the operation Equation (3.4). In Section 2.6, we saw that the `+/` had an infinite monadic (and dyadic) rank and, therefore, should have operated across the entire object. However `+/` has combined with `%`, which has a (monadic and dyadic) rank of zero:

```
   % b. 0
0 0 0
```

For the phrase: `f1@f2`, f_1 *inherits* the rank of f_2. Thus, for `(+/@%)`, `+/` *inherits* the rank of zero from −. This can be confirmed by:

```
  (+/@-) b. 0
0 0 0
```

The *at* conjunction performs the same sequencing operation as *atop*, but without rank inheritance:

```
   (+/@:-) b. 0
_ _ _
```

The J expression below confirms that the phrase operates at the desired rank and correctly performs the operation in Equation (3.4):

```
   (+/@:%) x
2.45
```

The phrase below shows the *atop* conjunction used in dyadic form: `y (f1@f2) x`. In functional notation, this is equivalent to $f_1(f_2(y, x))$. The arguments y and x are operated on by f_2, which behaves as a dyad, f_1 then operates monadically on the result. The dyad below implements the expression $2y/x$:

```
   y (+:@%) x
_ 4 2 1.33333 1 0.8
```

Sometimes it is necessary to override the rank of a verb. Let us return to the example of the three dimensional array x_3 from Section 2.6. We redefine x_3 below:

```
   ]x3 =: i. 2 2 3
0  1  2
3  4  5

6  7  8
9 10 11
```

Recall that, when we applied +/ to x_3, it summed the elements across the planes:

```
   +/ x3
 6  8 10
12 14 16
```

If we wish to sum x_3 along the rows, we need to explicitly specify the rank attribute of the verb with the " conjunction:

```
   +/"1 x3
 3 12
21 30
```

Here the (monadic) rank attribute has been explicitly set to one. This can be confirmed by:

```
   +"1 b. 0
1 1 1
```

It is important that the rank attribute of +/ has *not* been changed. A new verb has been formed, one which performs the summation of the rows of, in this case, a matrix. To sum down the columns set the rank attribute to two:

```
   +/"2 x3
 3  5  7
15 17 19
```

Also, from Section 2.6, we can re-examine the addition of the two-dimensional array x_2 to a three element list. Redefine x_2:

```
   ]x2 =: i. 2 3
0 1 2
3 4 5
```

An error is returned if we use the expression:

```
   1 2 3 + x2
|length error
|   1 2 3    +x2
```

Specifying a rank attribute of one gives the desired result:

```
   1 2 3 +"1 x2
1 3 5
4 6 8
```

The *gerund* ` provides a means joining a number of verbs in a list. It is commonly used on *agenda* conjunction @. for implementing selection.

$$x(t) = \begin{cases} 2t + 1 \text{ if } t > 0 \\ 0 \qquad\quad \text{otherwise} \end{cases} \tag{3.5}$$

We illustrate *gerund* and *agenda* with the following example. The function $x = 2t+1$ can be implemented with the J expression:

```
   (>:@+:) i:5
_9 _7 _5 _3 _1 1 3 5 7 9 11
```

However $x = 2t + 1 \; \forall t$, whereas for $t \leq 0$, in Equation (3.5), $x = 0$. This can be implemented by the phrase:

```
   0: ` (>:@+:) @. (0&<:) &> i:5
0 0 0 0 0 1 3 5 7 9 11
```

The function `(0&<:)` acts as a condition statement, returning one if the argument is greater than zero and zero otherwise:

```
   (0&<:) i:5
0 0 0 0 0 1 1 1 1 1 1
```

The first function in the list is invoked if the condition statement returns a zero, which in this case is the constant function `0:`. The next function is invoked, in this case `(>:@+:)`, if one is returned.

Note the use of the `&>`. This ensures that the phrase is applied to each element in the list. From the expression below we can see that the phrase in question has infinite rank and, therefore, does not apply at the 0-cell level as we wish:

```
   0: `(>:@+:) @. (0&<:) b. 0
_ _ _
```

We could explicitly specify a zero rank:

```
   0: ` (>:@+:) @. (0&<:)"0 i:5
0 0 0 0 0 1 3 5 7 9 11
```

We can avoid setting the rank by using *each* construct `&>`. This method is frequently used in this book.

3.2 Examples

In this section we present a number of worked examples to demonstrate how to implement some mathematical expressions related to data networks.

3.2.1 More z-transforms

We continue with the z-transforms in the introduction to this chapter. The unit impulse function is closely related to the unit step function (Equation (3.2)), where $u[n]$ is the running sum of $\delta[n]$.

$$u[n] =: \sum_{n=0}^{\infty} \delta[n] \tag{3.6}$$

We can implement $\delta[n]$ with the J verb below:

```
   uimpls =: 0&=
```

The phrase forms an *equal-to-zero* function, as demonstrated in the command-line below:

```
   n =: i.10
   uimpls n
1 0 0 0 0 0 0 0 0 0
```

The z-transform for $\delta[n]$ is $1\ \forall z$. The implementation of the z-transform for the $\delta[n]$ function is trivial, as we can just use the `1:` constant verb:

```
   z =: 1 2 3 4 5 6
   1: &> z
1 1 1 1 1 1
```

The transform in Equation (3.1) is realised by the J expression below. It confirms that the $\delta[n]$ function yields a value of one:

```
   n =: i.100
   +/ (uimpls * z&^@-) n
1 1 1 1 1 1
```

Consider the transform of the function $a^n u[n]$:

$$\mathcal{Z}\{a^n u[n]\} = \frac{1}{1 - az^{-1}}, \qquad |z| > |a| \tag{3.7}$$

The left-hand side of the expression in Equation (3.7) is implemented by the phrase in brackets:

```
   a =: 2 [ z =: 3 4 5 6 7 8
   a (%@>:@-@*%) z
3 2 1.66667 1.5 1.4 1.33333
```

The corresponding z-transform $a^n u[n]$ can be found using the unit step function:

```
   n =: 100
   +/ (a&^ * ustep * z&^@-) n
3 2 1.66667 1.5 1.4 1.33333
```

3.2.2 Shannon's Result

Consider the example of Shannon's result for the maximum channel capacity C [62]:

$$C = W \log_2 \left(1 + \frac{S}{N}\right) \tag{3.8}$$

where W is the bandwidth of the channel and S/N is the signal to noise ratio. For a bandwidth $W = 3100$ Hz and signal to noise ration $S/N = 1000 : 1$ (30 dB), the channel capacity C is given by the J expression:

```
   3100 (*log2 @ >:) 1000
30898.4
```

Thus, the channel capacity C is approximately 30.9 kb/s. The *increment* verb `>:` *adds* one to the right argument, which, in this case, is the signal to noise ratio $S/N = 1000$. The verb log_2 is applied to the result of `>:` using the *atop* conjunction. Finally, the `*` verb performs the multiplication function between the left argument ($W = 3100$) and the result of $\log_2$.

3.2.3 Euler's Formula

Euler's formula [59] demonstrates the relationship between the exponential function and the trigonometrical functions:

$$e^{i\theta} = \cos\theta + i\sin\theta \tag{3.9}$$

where $i = \sqrt{-1}$ is the imaginary unit. J comes supplied with trigonometrical functions, but they have to be loaded:

```
   load 'trig' NB. load trigonometrical functions
   ]th =: 2p1 * (i.>:4)%4 NB. 2p1 = 2*pi
0 1.5708 3.14159 4.71239 6.28319
   sin th
0 1 1.22465e_16 _1 _2.44929e_16
```

The *complex* verb `j.` multiplies its argument by `0j1` so we can implement (the right hand side of) Euler's formula as:

```
   eul =: cos+j.@sin
   eul th
1 6.1232e_17j1 _1j1.2247e_16 _1.837e_16j_1 1j_2.4493e_16
```

However, with J, we can implement Euler's formula from the left-hand side of Equation (3.9):

```
   eu2 =: ^@j.
   eu2 th
1 0j1 _1 0j_1 1
```

From these results, *eu1* and *eu2* may not appear equivalent, but closer examination reveals that they (nearly) are. The problem is that *eu1* has suffered precision error and returned very small values for results that should actually be zero. For this reason, *eu2* is favoured over *eu1*:

```
   eu =: ^@j.
```

Moreover, this solution is far more elegant, and quite fitting, as Euler's formula is considered to be one of the most beautiful mathematical equations. To conclude this example, we demonstrate Euler's identity:

$$e^{i\pi} + 1 = 0 \tag{3.10}$$

We can confirm this with:

```
   >:@eu 1p1    NB. 1p1 = 3.14159 (pi)
0
```

Euler's formula is used extensively in transforms. We will make use of it later in developing a Fourier transform function.

3.2.4 Information Entropy

The concept of *entropy* in information theory relates to the degree of randomness in a signal [62]. For n discrete symbols in a signal, the entropy $\mathcal{H}$ is given by:

$$\mathcal{H} = -\sum_{i=1}^{i=n} p(i) \log p(i) \tag{3.11}$$

where $p(i)$ is the probability of symbol i occurring. We need the $\log_2$ function that was defined earlier. We define p_0 and p_1 as probability distributions for a signal that comprises four distinct symbols:

```
   ]p0 =: (#%) 4
0.25 0.25 0.25 0.25  NB. equal probability
   ]p1 =: 0.1 0.2 0.5 0.2
0.1 0.2 0.5 0.2
```

We will develop the function in stages, so that we can show how it works. First multiply $p_1(i)$ by $\log_2 p_1(i)$:

```
   (*log2) p1
_0.332193 _0.464386 _0.5 _0.464386
```

Then sum the resultant terms ($\sum_{i=1}^{i=n} p_1(i)\log_2 p_1(i)$):

```
   (+/@:*log2) p1
_1.76096
```

The final operation is to negate the result:

```
   (-+/@:*log2) p1
1.76096
   entropy =: -+/@:*log2 NB. define entropy
```

We can compute $\mathcal{H}_{max}$ by applying *entropy* to $p0$, the probability distribution for when all symbols are equally likely:

```
   entropy p0  NB. gives value of maximum entropy
2
```

We can compute the amount of *redundancy* $\mathcal{H}_{max} - \mathcal{H}$ for a particular signal (in this case with probability distribution p):

```
   -/@entropy p0,.p1
0.239036
```

3.3 Good Programming Practice

Before undertaking any serious program development in J it is worth looking briefly at good programming practices. Consider the expression for computing the sample variance σ^2:

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{i=n}(x_i - \overline{x})^2 \tag{3.12}$$

where $\overline{x}$ is the arithmetic mean. One implementation of the sample variance in J is:

```
   var1 =:   <:@#%~+/@(*:@-+/%#)
   var1 i.6
3.5
```

There are a couple of issues relating to this verb. Firstly, it is virtually unreadable. Secondly, it is very difficult to debug, as this is just one function made out of nine primitives and four conjunctions. In short, this is *not* how to write J code! In this respect, J is like any other programming language; it is quite easy to write unreadable code that is difficult to debug.

Here, we rewrite the variance function. In Subsection 3.1.3 we showed a phrase that implemented a *deviations-from-mean* function. We define it thus:

```
   mdev =: -mean    NB. deviations from the mean
   mdev i.6
_2.5 _1.5 _0.5 0.5 1.5 2.5
```

Then, we square the deviations ($(x_i - \overline{x})^2$):

```
   sqdev =: *:@mdev  NB. square of the deviations
   sqdev i.6
6.25 2.25 0.25 0.25 2.25 6.25
```

And compute the sum of squares:

```
   sumsq =: +/@sqdev  NB. sum of squares
   sumsq i.6
17.5
```

The degrees of freedom is just $n - 1$:

```
   dof =: <:@#   NB. tally decremented by 1
   dof i.6
5
```

The resulting function *var2* shows that the variance is merely the sum of square deviations divided by the degrees of freedom:

```
   var2 =: sumsq%dof
   var2 i.6
3.5
```

Furthermore, the standard deviation σ is just the square root (`%:`) of the variance:

```
   std =: %:@var2
   std i.6
1.70783
```

It can be seen that breaking down (complex) J functions into smaller ones aids readability. Furthermore, each component function can be tested, making debugging easier. As a rule of thumb, do not use more than four functions to implement any other function. Despite the benefits of breaking functions down into smaller functions there is one drawback, and that is the necessity that (sub) function names are unique. As the number of functions we develop grows the greater the chance that two functions names will clash. In the next subsection we introduce *locales* as a means of avoiding clashes in the name space.

3.3.1 Locales

Functions can be organised into modules or *locales*. Verb names need only be unique within the locale, which helps to avoid name space clashes. In the example below, we define two verbs with the same name: f_1. An extra identifier is appended to verbs to denote the locale.

```
   f1_myloc_  =: +*:  NB. define f1 in 'myloc'
   f1_newloc_ =: *-:  NB. define f1 in 'newloc'
```

The verb `f1_myloc_` is defined in locale `myloc`, while `f1_newloc_` is defined in `newloc`. Notice that the locale name appears between two underscores. In order to reference them, we need to specify their function name and their locale:

```
   f1_myloc_ i.6
0 2 6 12 20 30
   f1_newloc_ i.6
0 0.5 2 4.5 8 12.5
```

However, this is only because we are not *in* either the *myloc* or *newloc* locales. The default locale is called *base* We can verify the locale we are in by the following (somewhat obscure) command:

```
  18!:5 ''
+----+
|base|
+----+
```

We can change the locale with the following command:

```
   cocurrent <'myloc'
```

Now, when we reference any verb defined within this locale, we do not need to include the locale name as part of the function name:

```
   f1 i.6
0 2 6 12 20 30
```

Just to verify that we are indeed in the *myloc* locale. The phrase below returns the current locale:

```
   18!:5 ''
+-----+
|myloc|
+-----+
```

Similarly for the f_1 defined in *newloc*:

```
   cocurrent <'newloc'
   f1 i.6
0 0.5 2 4.5 8 12.5
```

If we define a new verb f_2 here (omitting the locale designation), then f_2 will be in the *newloc* locale:

```
   f2 =: 5&*
   f2 i.6
0 5 10 15 20 25
```

We can define and reference f_2 without explicitly specifying the locale; but if we then return to the *base* locale, references to f_2 require the locale:

```
   cocurrent <'base'
   f2 i.6 NB. f2 not defined in 'base' locale
|value error: f2
|       f2 i.6
   f2_newloc_ i.6
0 5 10 15 20 25
```

In addition to the *base* locale, there is the *z* locale, which is appended to the *search path* of the current locale. When a verb is referenced without an explicit locale qualification, J searches for it in the current locale (either the *base* locale of the locale specified by the *cocurrent* directive). If it is not found, then the *z* locale is searched. Typically the *z* locale is used to hold standard utilities. The convention used in this book is to write a complex function as a series of subfunctions. These subfunctions are defined in some new locale. The actual function is defined in the *z* locale and references its subfunctions using an appropriate locale qualification. Any "wrapper" functions are defined in the *z* locale. Wrapper functions are discussed in the next subsection where we deal with explicit programming.

3.3.2 Explicit Programming

The verb composition methods covered so far in this book are refered to as *tacit programming*. With tacit programming, there is no explicit reference to arguments. J does, however, allow *explicit programming* methods where arguments are explicitly referenced. The verb definition below implements an explicit addition function:

```
   add =: 4 : 'x. + y.'
```

The literal argument 4 specifies that the verb definition is dyadic (3 specifies monadic). Here we use explicit reference names, namely x., which is the left argument and y., which is the right argument. Explicit programming also allows for *conventional* programming constructs like branches (if statements) and flow-control (for and while loops). Here is a *conventional* definition of the summation function:

```
   sum =: 3 : 0
n =. 0 [ total =. 0
for_k. y.
do.
  total =. total + k
end.
total
)
```

Check that *sum* yields the same result as +/ with the expression:

```
   (+/;sum) i.6
+--+--+
|15|15|
+--+--+
```

While J supports explicit programming techniques, it is better to adopt tacit programming over explicit programming where possible. Verbs that are defined tacitly are interpreted at definition time, whereas explicit verb definitions are translated at execution time. However, there may be times when a function is too complex to be written entirely in tacit form and it is more convenient to write it in explicit form. Tacit programming methods (as we will show) are conducive to implementing the concepts in this topic area. Explicit programming is used when complexity necessitates this. Primarily, we use explicit functions as *wrappers* that call tacit functions that perform the core computational operations.

3.4 Scripts

Verb definitions are lost when the J session is closed down. Verbs definitions can be preserved by storing them in a *script* file. When the J interpreter is started, the scripts can be loaded in. Create a file *fscript.ijs* that contains the following function definitions:

```
NB. Script file for verb f
f =: *:+-:  NB. x^2 + x/2
```

Now start up a new J sesssion and try to run the function f:

```
   f i.6
|value error: f
|       f i.6
```

As this is a new J session, any previous function definitions will be lost. However, they can be *restored* by loading *fscript.ijs*:

```
   load 'fscript.ijs'
   f i.6
0 1.5 5 10.5 18 27.5
```

The directory pathname has to be specified if the script is in a different directory from which the J session was executed. Script files can be loaded at the beginning of the J session by using the *load* command in the *startup* script. For more details see, Appendix A.

3.5 Summary

In the chapter we have primarily focused on verb composition. A sequence of verbs forms a phrase that combine to define a new verb. The two basic compositional constructs are the hook and fork. A hook is a sequence of two verbs: `(f1 f2)` and a fork is a sequence of three verbs: `(f1 f2 f3)`. Conjunctions such *bond* `&` and *atop* `@` (and *at* `@:`) are also compositional constructs. The `&` conjunction enables constants to be attached to verbs. The `@` is a sequencing construct: `(f1@f2)`, where f_1 performs a transformation on the transformation of f_2 on its arguments. However the `@` conjunction causes rank inheritance, that is f_1 inherits the rank attributes of f_2 and will operate at *that* rank. In cases where we wish to avoid rank inheritance, we use the *at* conjunction `@:` instead.

4

Network Calculus

Network calculus is a framework for analysing network traffic flows. In classical queuing theory [33] traffic arrivals are expressed as stochatic processes. In network calculus a flow is characterised as a wide-sense increasing function, which defines an upper bound on the cumulative number of arrivals over a time interval. Any other characteristics of the flow beyond this are unknown. Similarly, network resources can also be expressed in terms of wide-sense increasing functions. The foundations of network calculus is *min-plus algebra*, in which the "addition" and "multiplication" operators are replaced by minimum and plus, respectively. Furthermore, $+\infty$ is included in the set of elements over which min-plus algebra is performed. By applying this algebraic structure to wide-sense increasing *arrival* and *services* curves, upper-bounds on QoS metrics, such as delay and backlog, can be derived

Only a few pre-defined functions are required for this chapter. They are:

```
min_z_ =: <./  NB. minimum
max_z_ =: >./  NB. maximum
max0 =: 0&max  NB. max(0,x)
ceil =: <.     NB. ceiling function
```

4.1 Characterising Traffic Flows

In Chapter 1, a traffic flow was represented as a sequence a indexed by $t = 1, 2, \ldots$, where $a(t)$ is the number of arrivals in the interval t. Alternatively, traffic flows can be characterised by a cumulative sequence $A = \{A(t), t = 0, 1, \ldots\}$, where $A(t)$ is the number of arrivals in the interval $[0, t]$ and $A(0) = 0$; that is:

$$A(t) = \sum_{s=1}^{s=t} a(s) \qquad A(0) = 0$$

We can characterise $A(t)$ as being f-upper, constrained by the expression:

$$A(t) - A(s) \leq f(t-s) \quad \forall 0 \leq s \leq t \tag{4.1}$$

We define the arrival sequence a_1 as a vector:

```
   a1 =: 3 5 2 4 5 2 3 1 3 2
```

The J expression below gives A_1, which is the sequence of cumulative arrivals of a_1:

```
   ]A1 =: 0, +/\ a1   NB. prepend zero because A(0) = 0
0 3 8 10 14 19 21 24 25 28 30
```

For reasons that will become apparent later, it is convenient to define $A_1(t)$ as function instead of as a data object. We can use the *seq* verb thus:

```
   seq_z_ =: {~  NB. sequence verb
   A1 =: 0 3 8 10 14 19 21 24 25 28 30&seq
```

We can reference $A_1(3)$, for example, with the expression:

```
   A1 3
10
```

Using the CBR (constant bit rate) link example in Section 1.4, $A_1(t)$ is bounded by $f(t) = ct$, that is A_1 is c-uppper constrained iff:

$$A_1(t) - A_1(s) \leq c \cdot (t-s) \qquad \forall 0 \leq s \leq t \tag{4.2}$$

Here, we show how to use J to analyse the characteristics of traffic flows. We define the time index terms t and s as verbs:

```
   t_z_ =: ]
   s_z_ =: i.@>:@t
```

The use of t and s is demonstrated below:

```
   hd1 =: 't';'s';'t-s'
   hd1,: (t;s;t-s) 10
+--+----------------------+----------------------+
|t |s                     |t-s                   |
+--+----------------------+----------------------+
|10|0 1 2 3 4 5 6 7 8 9 10|10 9 8 7 6 5 4 3 2 1 0|
+--+----------------------+----------------------+
```

Set the CBR link peak rate to $c = 3$:

```
   c =: 3:
```

We can realise the inequality in Equation (4.2) with the following expression:

```
   ((A1@t - A1@s) <: c*(t-s)) 10
1 1 1 1 1 1 1 1 1 1 1
```

The result shows that A_1 is c-upper constrained over the interval $[0,t]$ for $t = 10$. Indeed, the mean rate of the flow $A(t)/t = 3$, is equivalent to the speed of the CBR link. However, for A_1, traffic does not arrive at a constant rate. For example, the flow is not c-upper constrained when $t = 5$:

```
   ((A1@t-A1@s) <: c*(t-s))  5
0 0 0 0 0 1
```

We can see from the expression below, $\forall t$ and $c = 3$, $A_1(t)$ is not c-upper constrained:

```
   */@ ((A1@t-A1@s) <: c*(t-s))  &> i.11
1 1 0 0 0 0 0 0 0 0 1
```

A CBR link does not allow for bursts of traffic above the peak rate function. The amount of traffic that departs from the link at any time interval t is bounded by the peak rate. That is, traffic $A(t) - A(t-1)$ in excess of c is buffered (or dropped) at the link. This is a consequence of operating a CBR link at the mean arrival rate of the flow. Backlog can be avoided by operating the link at the peak rate of the traffic flow. The peak rate of A_1 is derived:

```
   max@(}.@A1 - }:@A1) i.11
5
```

As expected, the flow A_1 is c-upper constrained for $c = 5$:

```
   c =: 5:
   */@ ((A1@t-A1@s) <: c*(t-s))  &> i.11
1 1 1 1 1 1 1 1 1 1 1
```

Traffic flows are typically bursty, such that operating a link at the peak rate of the flow, typically results in the link being underutilised. Leaky bucket algorithms can be used to characterise VBR (variable bit rate) flows. A leaky-bucket is defined in terms of a rate ρ and burst tolerance σ. Thus an arrival curve f is given by:

$$f(t) = \rho t + \sigma \tag{4.3}$$

Thus $A_1(t)$ is (ρ, σ)-upper constrained if:

$$A_1(t) - A_1(s) \leq \rho \cdot (t - s) + \sigma \tag{4.4}$$

We define ρ and σ, respectively as:

```
   rho   =: 3:
   sigma =: 4:
   f =: 0&<* sigma + rho*t
```

The `0&<*` term ensures $f(0) = 0$. From the J expression below, it can be seen that the inequality in Equation (4.4) is satisfied:

```
    */@ ((A1@t - A1@s) <: f@(t-s)) &> i.11
1 1 1 1 1 1 1 1 1 1 1
```

We have shown that we can characterise bursty traffic as a CBR or VBR flow. A_1 is bounded by a CBR flow where the rate c is set to the flow's peak rate, hence $c = 5$. As a VBR flow, A_1 is bounded rate $\rho = 3$ (which is the flow's mean rate) and a burst tolerance of $\sigma = 4$. Other arrival curves also bound A_1:

```
   rho   =: 4:
   sigma =: 1:
   */@ ((A1@t - A1@s) <: f@(t-s)) &> i.11
1 1 1 1 1 1 1 1 1 1 1
```

There are, therefore, a number of options in terms of arrival curve for A_1.

4.2 Min-Plus Algebra

Network calculus applies system theory to the problem of analysing traffic flows within networks. When system theory is used to analyse electronic circuits, conventional algebra $(\mathbf{R}, +, \times)$ can be used. However, when analysing network flows, *min-plus* algebra is adopted. With min-plus algebra, *addition* is replaced by *min* and the *product* is replaced by *plus*. In conventional algebra, the output signal $y(t) \in \mathbf{R}$ of a circuit is the convolution of the input signal $x(t) \in \mathbf{R}$ and impulse response of the circuit $y(t) \in \mathbf{R}$. Thus the convolution $y \star x$, is given by:

$$(x \star y)(t) = \int_{0 \le s \le t} x(s) \cdot y(t-s) d(s) \tag{4.5}$$

In min-plus algebra, the convolution operator is:

$$(f \star g)(t) = \min_{0 \le s \le t} [f(s) + g(t-s)] \tag{4.6}$$

where f and g are non-negative, wide-sense increasing functions. The min-plus algebra is the dioid: $(\{\mathbf{R} \cup \infty\}, \oplus, \otimes)$ which is a *commutative semifield* with properties of:

- commutivity

  $a \oplus b = b \oplus a$

  $a \otimes b = b \otimes a$
- associativity

  $(a \oplus b) \oplus c = a \oplus (b \oplus c)$

  $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
- distributivity

  $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \oplus c)$

We demonstrate these three properties for the $\min$ ($\oplus$) operator in J (the $\otimes$ operator is also commutative, associative and distributive). The expressions below return a one if the equality is met and zero otherwise. Commutative property:

```
   3 min 2 = 2 min 3
1
```

Associative property:

```
   ((3 min 2) min 4) = 3 min (2 min 4)
1
```

Distributive property:

```
   ((3 min 2) + 4) = (3 + 4) min (2 + 4)
1
```

Matrix operations can also be performed under min-plus algebra. Thus, for the operation $\mathbf{P} \oplus \mathbf{Q}$, we have:

$$\begin{pmatrix} p_1 \\ p_2 \end{pmatrix} \oplus \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} \min[p_1, q_1] \\ \min[p_2, q_2] \end{pmatrix} \tag{4.7}$$

Which we can demonstrate in J:

```
   P =: 1 4
   Q =: 2 3
   P min"0 Q
1 3
```

For the operation $\mathbf{P} \otimes \mathbf{M}$, we have:

$$\begin{pmatrix} p_1 \\ p_2 \end{pmatrix} \otimes \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} \min[p_1 + m_{11}, p_1 + m_{12}] \\ \min[p_2 + m_{21}, p_2 + m_{22}] \end{pmatrix} \tag{4.8}$$

In regular algebra, the *inner product* function is implemented by the phrase `(+/ .*)`. The equivalent inner-product operation in min-plus algebra is `(min"1 .+)`. The min-plus inner-product is illustrated below:

```
   ]M =: i. 2 2
0 1
2 3
   P (min"1 .+) M
1 5
```

We are particularly interested in the application of min-plus algebra to functions or sequences (indexed by $t = 1, 2, \ldots$). Consider the two functions, f_1 and g_1:

$$f_1(t) = \begin{cases} t+4 & \text{if } t > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

$$g_1(t) = \begin{cases} 3t & \text{if } t > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

The functions f_1 and g_1 are defined below in J:

```
   f1 =: 0&<* (4: + t)
   g1 =: 0&<* (3: * t)
   (f1 ,: g1) i.11
   (f1 ,: g1) i.11
0 5 6 7  8  9 10 11 12 13 14
0 3 6 9 12 15 18 21 24 27 30
```

The pointwise minimum of two functions is:

$$(f \oplus g)(t) = \min[f(t), g(t)] \tag{4.11}$$

There are two ways of expressing Equation (4.11) in J. We illustrate with functions f_1 and g_1:

```
   min@(f1@t,g1@t) &> i.11
0 3 6 7 8 9 10 11 12 13 14
```

or:

```
   (f1@t min g1@t) &> i.11
0 3 6 7 8 9 10 11 12 13 14
```

The min-plus convolution $(f \star g)(t)$ (introduced in Section 4.2) is realised by the J expression:

```
   min@(f1@s+g1@(t-s)) &> i.11
0 3 6 7 8 9 10 11 12 13 14
```

In this particular case $f_1 \star g_1 = \min[f_1, g_1]$. We can show that both operations are commutative, that is: $f \oplus g = g \oplus f$ and $f \star g = g \star f$. We demonstrate commutativity in the pointwise minumum:

```
   min@(g1@t,f1@t) &> i.11
0 3 6 7 8 9 10 11 12 13 14
```

The convolution operator is also commutative:

```
   min@(g1@s + f1@(t-s)) &> i.11
0 3 6 7 8 9 10 11 12 13 14
```

Min-plus algebra is associative in $\oplus$, that is $(f \oplus g) \oplus h = f \oplus (g \oplus h)$: Define a new function $h_1(t) = 2t + 1$:

```
   h1 =: 0&<* (1: + 2: * t)
   h1 i.11
0 3 5 7 9 11 13 15 17 19 21
```

It can be seen that the two J expressions are equivalent:

```
   min@(min@(f1@t, g1@t), h1@t) &> i.11
0 3 5 7 8 9 10 11 12 13 14
   min@(f1@t, min@(g1@t, h1@t)) &> i.11
0 3 5 7 8 9 10 11 12 13 14
```

Alternatively, we could use the format:

```
   ((f1@t min g1@t) min h1@t) &> i.11
0 3 5 7 8 9 10 11 12 13 14
   (f1@t min (g1@t min h1@t)) &> i.11
0 3 5 7 8 9 10 11 12 13 14
```

The operations $\oplus$ and $\star$ are distributive, that is:

$$(f \oplus g) \star h = (f \star h) \oplus (g \star h) \tag{4.12}$$

The left-hand side of Equation (4.12) is given by:

```
   min@((min@(f1@s,:g1@s)) + h1@(t-s)) &> i.11
0 3 5 7 8 9 10 11 12 13 14
```

Here, we show the right-hand side of Equation (4.12) is equivalent, which confirms the distrbutive property:

```
   (min@(f1@s+h1@(t-s)) min min@(g1@s+h1@(t-s))) &> i.11
0 3 5 7 8 9 10 11 12 13 14
```

Consider the two sequences ϵ and $\mathbf{e}$, where $\epsilon(t) = \infty \; \forall t$ while $\mathbf{e}(0) = 0$ and $\mathbf{e}(t) = \infty \; \forall t > 0$. Then ϵ is the absorbing element, such that:

$$f \star \epsilon = \epsilon \star f = \epsilon \tag{4.13}$$

and $\mathbf{e}$ is the identity element:

$$f \star \mathbf{e} = \mathbf{e} \star f = f \tag{4.14}$$

We can define ϵ and $\mathbf{e}$ in J as:

```
   et_z_ =: _:      NB. absorbing element
   e_z_  =: _: * *  NB. identity element
```

The J expressions below demonstrate ϵ and $\mathbf{e}$:

```
   (e,:et) i.11
0 _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _
```

We demonstrate the absorption and identity properties of Equations (4.13) and (4.14) using the function f_1:

```
   min@(f1@s + et@(t-s)) &> i.11
_ _ _ _ _ _ _ _ _ _ _
   min@(f1@s + e@(t-s)) &> i.11
0 5 6 7 8 9 10 11 12 13 14
```

One of the differences of min-plus algebra and *regular* algebra is the idempotency of addition, We show that $f = f \oplus f$ with the J expression:

```
   min@(f1@t,f1@t) &> i.11
0 5 6 7 8 9 10 11 12 13 14
```

4.3 Mathematical Background

There are a number of function types that are important to min-plus algebra, namely:

- wide-sense increasing functions
- subadditive functions
- convex and concave functions

In this section we, examine the properties of these functions using J. We also introduce the concept of *subadditive closure*.

4.3.1 Wide-Sense Increasing

$\mathcal{F}$ denotes the set of wide-sense increasing functions and sequences. A function f belongs to the set wide-sense increasing functions $\mathcal{F}$ iff:

$$f(s) \leq f(t) \qquad \forall s \leq t \tag{4.15}$$

Furthermore, $\mathcal{F}_0$ denotes the set wide-sensing function for which $f(0) = 0$. Consider the two functions f_2 and g_2 below:

```
   f2 =: *^@-
   g2 =: %>:
```

We can see the f_2 is not wide-sense increasing:

```
   (f2@s <: f2@t) 10
1 0 0 0 0 0 0 0 0 0 1
```

However, g_2 is:

```
   (g2@s <: g2@t) 10
1 1 1 1 1 1 1 1 1 1 1
```

4.3.2 Types of Wide-Sense Increasing Functions

We have shown how peak rate and affine functions can be defined using the constant verbs (`1:`, `2:`, `3:` etc). This was done for convenience and served its purpose for illustrating the fundamental concepts of min-plus algebra. However, the practice is somewhat restrictive for any serious analysis. Defining wide-sense increasing functions that accept parameters greatly increases analytical flexibility. In this section we, define *parametric* peak rate and affine curve verbs. We also introduce other curves that are used in network calculus. We define a verb that, when applied, ensures a function belongs to $\mathcal{F}_0$, thus:

```
   load 'libs.ijs' NB. load pos. params. verbs
   F0_z_ =: [: 0&< rhs0
   F0 i.11
0 1 1 1 1 1 1 1 1 1 1
```

Peak Rate Function

For a rate $R \geq 0$, the peak rate $\lambda_R(t)$ function is given by:

$$\lambda_R(t) = \begin{cases} Rt & \text{if } t > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.16}$$

The J verb for peak rate function is:

```
   pr_z =: F0 * *
```

Thus, a peak rate function for $R = 5$ is given by the expression:

```
   5 pr i.11
0 5 10 15 20 25 30 35 40 45 50
```

For convenience, we can define the function $f_3 =: \lambda_5(t)$:

```
   f3 =: 5&pr
```

Verify that $f_3 =: \lambda_5(t)$ is wide-sense increasing for $t \geq 0$:

```
   (f3@s <: f3@t) 10
1 1 1 1 1 1 1 1 1 1 1
```

Affine Function

The affine curve $\gamma_{r,b}$ takes two parameters, the rate r and tolerance b, and is given by:

$$\gamma_{r,b}(t) = \begin{cases} rt + b & \text{if } t > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.17}$$

The verb for an affine curve is:

```
   af_z_ =: F0 * max0@(lhs2+lhs1*rhs0)
```

The expression below shows the affine function $g_3 = \gamma_{3,4}$:

```
   (g3 =: 3 4&af) i.11
0 7 10 13 16 19 22 25 28 31 34
```

Burst Delay

Burst delay is ∞ for $\forall t > T$ and zero otherwise:

$$\delta_T(t) = \begin{cases} +\infty & \text{if } t > T \\ 0 & \text{otherwise} \end{cases} \tag{4.18}$$

The J verb for the burst delay curve is:

```
   bd_z_ =: F0*_:
   3 bd i.11
0 0 0 0 _ _ _ _ _ _ _
```

Rate-Latency

The rate-latency function $\beta_{R,T}$ is given by the equation:

$$\beta_{R,T}(t) = R[t-T]^{+} = \begin{cases} R(t-T) & \text{if } t > T \\ 0 & \text{otherwise} \end{cases} \tag{4.19}$$

where parameters R, T are the rate and latency, respectively. The J verb definition is:

```
   rl_z_ =: lhs1 * max0 @ (rhs0-lhs2)
```

The rate-latency function $\beta_{3,2}$ yields the following result:

```
   3 2 rl i.11
0 0 0 3 6 9 12 15 18 21 24
```

Step Function

The step function v_T for some $T > 0$ is given by:

$$v_T(t) = 1_{t>T} = \begin{cases} 1 & \text{if } t > T \\ 0 & \text{otherwise} \end{cases} \tag{4.20}$$

The J verb definition is:

```
   step_z_ =: <
```

The step function for $T = 3$ is:

```
   3 step i.11
0 0 0 0 1 1 1 1 1 1 1
```

Stair Function

The stair function $u_{T,\tau}$ takes two parameters, T and τ, where $T > 0$ and tolerance $0 \leq \tau \leq T$. The expression for the stair function is given by:

$$u_{T,\tau}(t) = \begin{cases} \lceil \frac{t+\tau}{T} \rceil & \text{if } t > T \\ 0 & \text{otherwise} \end{cases} \tag{4.21}$$

The stair function is defined in J as:

```
   stair_z_ =: F0 * ceil@(lhs1 %~ lhs2 + rhs0)
```

For $T = 3$ and $\tau = 2$, the result of the stair function is:

```
   3 2 stair i.11
0 1 1 1 2 2 2 3 3 3 4
```

Fig 4.1 shows a graphical representation of the wide-sense increasing functions described above.

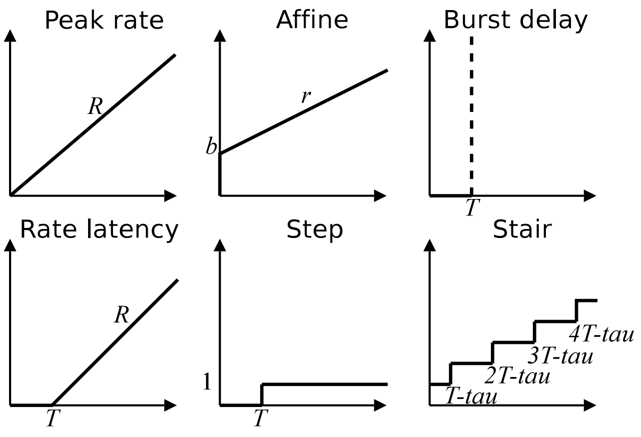

Fig. 4.1. Wide-sense increasing functions

4.3.3 Subadditive Functions

Subadditivity is an important property in network calculus. For a traffic flow constrained by a non-subadditive function $f' \in \mathcal{F}$, there exists a subadditive function $f \in \mathcal{F}$ which has a tighter bound than f'. A function or sequence f that belongs to $\mathcal{F}$ is subadditive if and only if $\forall s \leq t$ and $t \geq 0$:

$$f(t+s) \leq f(t) + f(s) \tag{4.22}$$

Define two function f_4 and g_4:

```
   f4 =: %:  NB. sqrt(t)
   g4 =: *:  NB. t^2
```

The expression below shows that f is subadditive:

```
   (f4@(t+s) <: f4@(t)+f4@(s)) 10
1 1 1 1 1 1 1 1 1 1 1
```

whereas g is not:

```
   (g4@(t+s) <: g4@(t)+g4@(s)) 10
1 0 0 0 0 0 0 0 0 0 0
```

This is equivalent to imposing the constraint $f \leq f \star f$. If $f \in \mathcal{F}_0$, $\mathcal{F}_0$ is a subset of $\mathcal{F}$ for which $f(0) = 0$, then $f = f \star f$.

```
   (f4@t = min@(f4@s + f4@(t-s))) &> i.10
1 1 1 1 1 1 1 1 1 1
```

We define a function, $h_4 \in \mathcal{F}$, where $h_4(t) = f_4(t)+1$. The function h is subadditive although $h(0) \neq 0$, so it satisifies $h_4 \leq h_4 \star h_4$, but not $h_4 = h_4 \star h_4$. This is demonstrated below:

```
   h4 =: >:@f4
   (h4@t = min@(h4@s + h4@(t-s))) &> i.10
0 0 0 0 0 0 0 0 0 0
   (h4@t <: min@(h4@s + h4@(t-s))) &> i.10
1 1 1 1 1 1 1 1 1 1
```

4.3.4 Subadditive Closure

A function constrained by wide-sense increasing function is also constrained by its *subadditive closure*. The subadditive closure f^* of a function f is the largest subadditive function less than or equal to f.

The function $f^{(n)}$ is the convolution of f with itself, that is, $f^{(n)} = f \star f^{(n-1)}$ and $f^{(1)} = f$. The subadditive closure is decreasing in n and converges to a limit. The function f^* represents the *subadditive closure* of $f \in \mathcal{F}$ and is given by the recursive equation:

$$\begin{aligned} f^*(0) &= 0 \\ f^*(t) &= \min[f(t), \min_{0<v<t}[f^*(v) + f^*(t-v)]] \end{aligned} \tag{4.23}$$

We can express this in J using the following J verbs:

```
   v_z_ =: }.@i.@t
   stop_z_ =: 2&<.
   conv_z_ =: min@(f, close"0@v + close"0@(t-v))
   close_z_ =: (0: ` f ` conv @. stop)"0
```

For the purpose of illustration, we use the peak rate function $f_3 = \lambda_3$ defined earlier:

```
   f =: f3
```

We find the subadditive closure:

```
   close &> i.11
0 3 6 9 12 15 18 21 24 27 30
```

In this case, we can see that f_3 is equal to its subadditive closure. Now consider the function $g_4 = t^2$ defined earlier:

```
   g4 i.11
0 1 4 9 16 25 36 49 64 81 100
```

The subadditive closure g_4^* is computed:

```
   f =: g4
   close &> i.11
0 1 2 3 4 5 6 7 8 9 10
```

In this case, the subadditive closure of g_4 is less then or equal to g_4 itself, that is $g_4^* = t$. We can verify that the subadditive closure of g_4 is indeed subadditive even though g_4 is not:

```
   (close@t <: min@(close@s + close@(t-s))) &> i.11
1 1 1 1 1 1 1 1 1 1 1
```

4.3.5 Concavity and Convexity

A function f is concave if, for $w \in [0, 1]$, the following holds:

$$f(ws + (1 - w)t) \geq wf(s) + (1 - w)f(t) \tag{4.24}$$

To illustrate, we examine the function $f_4(t) = \sqrt{t}$ defined earlier. Define w to evaluate to the left argument of the verb:

```
   w_z_ =: [
```

The function below generates a normalised sequence of values, we use to generate numbers in the interval $[0, 1]$:

```
  i01_z_ =: %(>./@:|)
  i01 i.6
0 0.2 0.4 0.6 0.8 1
```

Calculate $f_4(ws + (1 - w)t)$:

```
   ]x1 =: (i01 i.6) f4@((w * s) + >:@-@w * t) &> i.6
       0       0       0       0 0       0
0.894427       1       0       0 0       0
 1.09545 1.26491 1.41421       0 0       0
 1.09545 1.34164 1.54919 1.73205 0       0
0.894427 1.26491 1.54919 1.78885 2       0
```

Calculate $wf_4(s) + (1 - w)f_4(t)$:

```
   ]y1 =: (i01 i.6) ((w * f4@s) + >:@-@w * f4@t) &> i.6
       0       0       0       0 0       0
     0.8       1       0       0 0       0
0.848528 1.24853 1.41421       0 0       0
 0.69282 1.29282 1.54135 1.73205 0       0
     0.4     1.2 1.53137 1.78564 2       0
       0       1 1.41421 1.73205 2 2.23607
```

The expression below shows that Equation (4.25) holds, and therefore f_4 is concave:

```
   x1 >: y1
1 1 1 1 1 1
1 1 1 1 1 1
1 1 1 1 1 1
1 1 1 1 1 1
1 1 1 1 1 1
1 1 1 1 1 1
```

Conversely, f is convex if:

$$f(ws + (1 - w)t) \leq wf(s) + (1 - w)f(t) \tag{4.25}$$

Show that $g_4(t) = t^2$ is convex:

```
   x2 =: (i01 i.6) g4@((w * s) + >:@-@w * t) &> i.6
   y2 =: (i01 i.6) ((w * g4@s) + >:@-@w * g4@t) &> i.6
   x2 <: y2
1 1 1 1 1 1
1 1 1 1 1 1
1 1 1 1 1 1
1 1 1 1 1 1
1 1 1 1 1 1
1 1 1 1 1 1
```

4.3.6 Star Shaped Functions

Chang [10] and Le Boudec and Thiran [36] introduce the properties of *star-shaped* functions. Function f is star-shaped if $f(t)/t$ is wide-sense decreasing $\forall t > 0$. Concave functions are star-shaped. Thus:

```
   2 5 $ (f4%t) 1 to 10
       1 0.707107  0.57735      0.5 0.447214
0.408248 0.377964 0.353553 0.333333 0.316228
```

Note the shape term `2 5 $` is used merely for brevity, it arranges the output into a 2×5 matrix so that the values may be displayed on the page. The rate latency function $\beta_{3,2}$, which is not concave, is not star-shaped:

```
   3 2 (rl%t) 1 to 10
0 0 1 1.5 1.8 2 2.14286 2.25 2.33333 2.4
```

We can see that the two functions $f_3(t) = \lambda_3(t)$ and $g_3(t) = \gamma_{3,4}$ are star-shaped:

```
   ((f3%t) ,: (g3%t)) 1 to 10
5 5       5 5   5       5       5   5       5   5
7 5 4.33333 4 3.8 3.66667 3.57143 3.5 3.44444 3.4
```

One of the properties of star-shaped functions is that if f and g are star-shaped then so is $\min(f, g)$. We define h_3 as the pointwise minimum of f_3 and g_3:

```
   (h3 =: f3 min"0 g3) &> i.11
0 5 10 13 16 19 22 25 28 31 34
```

We show that h_3 is also star-shaped:

```
   (h3%t) 1 to 10
5 5 4.33333 4 3.8 3.66667 3.57143 3.5 3.44444 3.4
```

4.4 Arrival Curves

In Section 4.1, we showed how to characterise traffic flows. We say that, for $f \in \mathcal{F}$, a flow A is f-upper constrained if:

$$A(t) - A(s) \leq f(t - s) \qquad \forall s \leq t \tag{4.26}$$

This is equivalent to imposing the constraint $A = A \star f$, that is:

$$A = \min_{0 \leq s \leq t} (A(s) + f(t - s)) \tag{4.27}$$

Consider a link with a peak rate of $f_5(t) = \lambda_3(t)$:

```
   f5 =: 3&pr
```

We defined the flow sequence A_1 in Section 4.1, and we know that A_1 is *not* f_5-upper constrained. Thus, traffic will be buffered at the link causing a backlog, which is given by:

$$q(t) = \max_{0 \leq s \leq t} (A(t) - A(s) - f(t - s)) \tag{4.28}$$

In J, the backlog q_1 is computed thus:

```
   ]q1 =: max@((A1@t - A1@s) - f5@(t-s)) &> i.11
0 0 2 1 2 4 3 3 1 1 0
```

The expression in Equation (4.28) is a consequence of the Lindley equation [39]:

$$q(t+1) = (q(t) + A(t+1) - A(t) - c)^{(+)} \tag{4.29}$$

Chang [10] gives a formal proof by induction. Here, we merely demonstrate that they are equivalent by showing that Equations (4.28) and (4.29) give the same result for the example here. The verb for the Lindley equation is (re)defined in Listing 4.1.

Listing 4.1 *Lindley Equation*

```
cocurrent < 'LND'
qlist =: rhs0
c     =: >@lhs1
alist =: >@lhs2
ind   =: <:@#@qlist
anext =: ind { alist
qprev =: {:@qlist
qnext =: max0 @ anext + qprev + c
cocurrent <'base'
lindley_z_ =: qlist_LND_,qnext_LND_
```

The link capacity is passed to the *lindley* function as an argument (on the left-hand side) along with the sequence of arrivals a. Note that $a(t) = A(t) = A(t-1)$. We defined a_1 in Section 4.1, but we can derive it from A_1 with the expression:

```
   ]a1 =: (}. - }:) A1 i.11
3 5 2 4 5 2 3 1 3 2
```

However, a more verbose, but mathematically attractive expression could be:

```
   ]a1 =: (A1@(t+1:) - A1@t) &> i.10
3 5 2 4 5 2 3 1 3 2
```

For convenience, we pass the capacity and the backlog at $t = 0$ as literals (that is, $c = 3$ and $q(0) = 0$) to the *lindley* function. We show that Equations (4.28) and (4.29) are indeed equivalent:

```
   ]q2 =: (3;a1) lindley^:(10) 0
0 0 2 1 2 4 3 3 1 1 0
```

The output of a communications link B is given by the convolution of the arrival sequence A and the function for the *service curve* f; thus $B = A \star f$. Service curves are covered in Section 4.5 below. The output can therefore, be found:

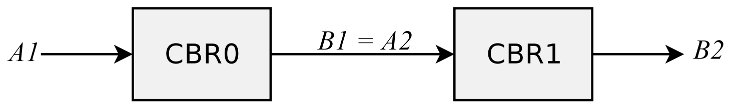

Fig. 4.2. Two CBR links in serial

$$B(t) = \min_{0 \leq s \leq t} (A(s) + f(t-s)) \tag{4.30}$$

Consider the following example. Given the arrival sequence A_1 and CBR link operating at a rate $f_5 = \lambda_3$, the output B_1 is:

```
   ]B1 =: min@(A1@s + f5@(t-s)) &> i.11
0 3 6 9 12 15 18 21 24 27 30
```

Assume that there are two CBR links, CBR0 and CBR1 (see Fig 4.2) connected in series (by a multiplexer which causes no delay) both operating at a rate $R = 3$ ($f_5 = \lambda_3$). We have seen from the example above that the output of CBR0 is B_1 for a flow with an arrival sequence A_1. The arrival sequence $A_2 = B_1$ is the input to the second link, CBR1. We define the sequence A_2 in J:

```
   A2 =: B1&seq
```

We can see that A_2 is f_1-upper constrained:

```
   */@ ((A2@t-A2@s) <: f5@(t-s)) &> i.11
1 1 1 1 1 1 1 1 1 1 1
```

There is no backlog at CBR1 because A_2 conforms to the link service curve:

```
   ]B2 =: max@((A2@t - A2@s) - f5@(t-s)) &> i.11
0 0 0 0 0 0 0 0 0 0 0
```

CBR0 has a *shaping* effect on the traffic flow A_1. In this particular case, shaping is merely a consequence of the link's capacity and the buffering mechanism. However, traffic shaping is a common technique for modifying the characteristics of a flow, such that it conforms to a specific set of traffic descriptors. Shaping can be used to reduce the backlog (and delays) at a communication link. Suppose we send the flow A_1 through a leaky bucket shaper, which shapes the incoming flow according to the rate and burst tolerance traffic descriptors $r = 3$ and $b = 1$, respectively. Figure 4.3 shows the shaper VBR0 connected to CBR1. Define $g_5 = \gamma_{3,1}$:

```
   g5 =: 3 1&af
```

The output from the shaper VBR0 B_3 and the arrival flow into CBR1 A_3 is:

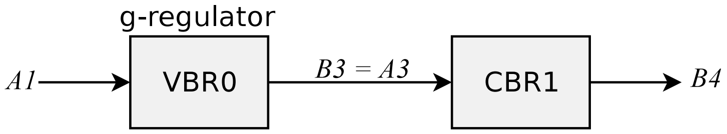

Fig. 4.3. Traffic flow shaped by a g-regulator

```
   ]B3 =: min@(A1@s + g5@(t-s)) &> i.11
0 3 7 10 13 16 19 22 25 28 30
   A3 =: B3&seq
```

The arrival flow to CBR1 is $A_3 = B_3$, where A_3 is g_1-upper constrained, though not f_1-upper constrained:

```
   */@ ((A3@t-A3@s) <: g5@(t-s)) &> i.11
1 1 1 1 1 1 1 1 1 1 1
   */@ ((A3@t-A3@s) <: f5@(t-s)) &> i.11
1 1 0 0 0 0 0 0 0 0 1
```

However, the backlog at CBR1 is reduced as a result of the shaping by VBR0:

```
   ]q4 =: max@((A3@t - A3@s) - f5@(t-s)) &> i.11
0 0 1 1 1 1 1 1 1 1 0
```

The shaper g_5 is called a g-regulator, as it modifies an incoming flow by buffering nonconforming traffic. It should be clearly understood that shaping traffic with a g-regulator does not cause the backlog to "disappear," merely that (some of) the backlog is incurred in the shaper itself rather than at the interface of the communications link. An alternative type of shaper is the g-clipper (see Fig 4.4), which drops nonconforming traffic instead of buffering it. The recursive function gives the departures B for arrivals A admitted to the g-clipper:

$$B(t) = \min(B(t-1) + A(t), \min_{0 \le s \le t}(B(s) + g(t-s))) \tag{4.31}$$

The J verb definition for the g-clipper is given in Listing 4.2.

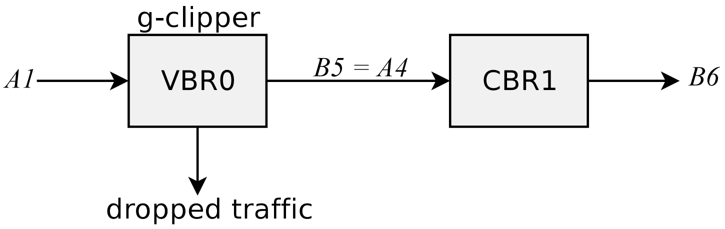

Fig. 4.4. Traffic flow shaped by a g-clipper

Listing 4.2 *g-clipper*

```
cocurrent < 'CLIPPER'
Blist =: rhs0
t =: #@Blist
s =: i.@t
Alist =: lhs0
ind   =: <:@#@Blist
Bprev =: {:@Blist
Aprev =: ind { Alist
Anext =: >:@ind { Alist
f1 =: min@(Blist + g@(t-s))
f2 =: Bprev + A@t - A@(<:@t)
Bnext =: min@(f1,f2)
cocurrent <'base'
gclipper_z_ =: Blist_CLIPPER_,Bnext_CLIPPER_
```

The verb *gclipper* operates explicitly on the arrival sequence A and shaping function g, thus:

```
   A_CLIPPER_ =: 0 3 7 10 13 16 19 22 25 28 30&seq
   g_CLIPPER_ =: 3 1&af
```

We can calculate the backlog q_5 at CBR1 as:

```
   ]B5 =: gclipper^:(10) 0
0 3 7 9 13 16 18 21 22 25 27
   A4 =: B5&seq
   ]q5 =: max@((A4@t - A4@s) - f1@(t-s)) &> i.11
0 0 1 0 1 1 0 0 0 0 0
```

It can be seen that, for the same traffic descriptors, the backlog at CBR1 is less when using the g-clipper than the g-regulator. This performance gain, however, has to be traded for some loss of traffic.

4.5 Service Curves

In order to provide guarantees to flows, the nodes must make resource *reservations* within the network. This function is carried out by *packet schedulers*. A packet scheduler is said to offer a service curve f for some $0 \leq t_0 \leq t$ if:

$$B(t) - B(t_0) \geq f(t - t_0) \tag{4.32}$$

where $B(t)$ is the backlog at time t and t_0 is the start of the busy period. The backlog is zero at t_0. It follows, then, that $B(t_0) - A(t_0) = 0$, and Equation (4.32) can be rewritten:

$$B(t) - A(t_0) \geq f(t - t_0) \tag{4.33}$$

This is equivalent to $B \geq A \star f$, or:

$$B(t) \geq \min_{t_0 \leq s \leq t} (A(s) + f(t - s)) \tag{4.34}$$

For the purpose of illustration, consider a CBR link, which guarantees that a flow will receive a service of rate $R = 3$, which we can represent using the function $f_5 = \lambda_3$, defined earlier. Suppose that the node for CBR link introduces a maximum delay of three. We use the Burst-delay function $f_6 = \delta_3$ to represent the latency in the node:

```
   f6 =: 3&bd
```

The overall service curve is given by the convolution of the two service curves: $h_1 = f_5 \star f_6$. For convenience, we assume $t_0 = 0$, thus:

```
   h1 =: min@(f5@s+f6@(t-s))
   h1 &> i.11
0 0 0 3 6 9 12 15 18 21 24
```

The resultant service curve h_1 is equivalent to the rate-latency function $f_7 = \beta_{3,3}$, which is defined in J below:

```
   (f7 =: 3 3&rl) i.11
0 0 0 0 3 6 9 12 15 18 21
```

4.5.1 Concatenation

Suppose we have two nodes, each offering rate-latency service curves β_{R_0,T_0} and β_{R_1,T_1}, respectively. The combined rate-latency curve $\beta_{R_0,T_0} \star \beta_{R_1,T_1}$ is the *concatenation* of the two systems; that is: $\beta_{\min(R_0,R_1),T_0+T_1}$. To illustrate this, we define the functions $f_8 = \beta_{3,2}$ and $f_9 = \beta_{4,1}$:

```
   f8 =: 3 2&rl
   f9 =: 4 1&rl
```

The convolution of the two curves $f_8 \star f_9$ yields the result:

```
   min@(f8@s + f9@(t-s)) &> i.11
0 0 0 0 3 6 9 12 15 18 21
```

The expression above is equivalent to the "concatenated" service curve $f_7 = \beta_{3,3}$.

4.5.2 Performance Bounds

Given that a flow arriving at a node is f-upper and the node offers a service curve g, we can compute the bounds for backlog, delay and output. For illustration, we will use an arrival curve $f_{10} = \gamma_{3,3}$ and service curve $g_6 = \beta_{6,3}$:

```
   (f10 =: 3 3&af) i.11
0 6 9 12 15 18 21 24 27 30 33
   (g6  =: 6 3&rl) i.11
0 0 0 0 6 12 18 24 30 36 42
```

The respective curves for f_{10} and g_6 are shown in Fig 4.5.

Backlog

The maximum backlog is given by the expression:

$$\max_{s \geq 0}(f(s) - g(s)) \tag{4.35}$$

We can calculate the backlog bound for f_{10} and g_6 by:

```
   max@(f10@s - g6@s) 10
12
```

Delay

The expression below gives the delay bound:

$$d(s) =: \min\{\tau \geq 0; f(s) \leq g(s + \tau)\} \tag{4.36}$$

The expression below generates a table showing $f_{10}(s)$ and $g_6(s + \tau)$ for different values of τ.

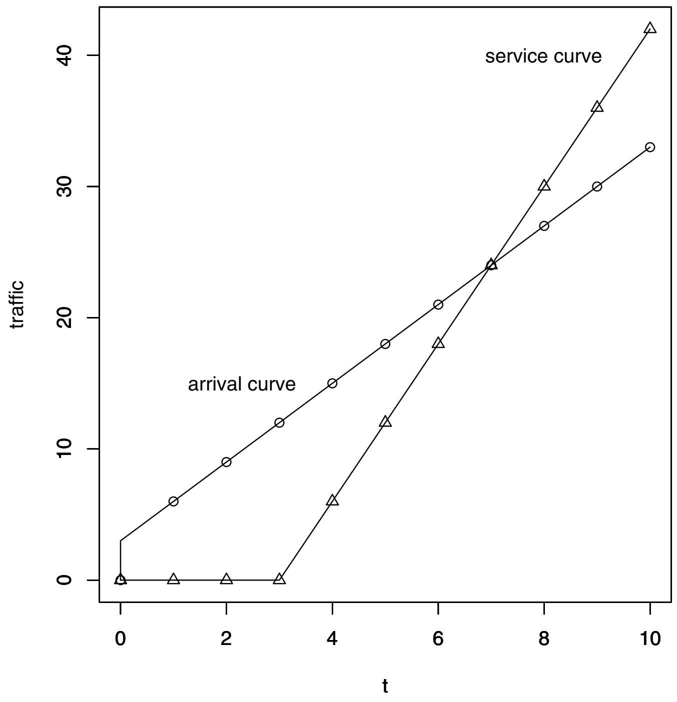

Fig. 4.5. Arrival curve $\gamma_{3,3}$ and service curve $\beta_{6,3}$

```
   tau_z_ =: lhs0
   (i.7) (tau; (f10@s ,: g6@(s+tau)))"0 (10)
+-+-------------------------------+
|0|0 6 9 12 15 18 21 24 27 30 33  |
| |0 0 0  0  6 12 18 24 30 36 42  |
+-+-------------------------------+
|1|0 6 9 12 15 18 21 24 27 30 33  |
| |0 0 0  6 12 18 24 30 36 42 48  |
+-+-------------------------------+
|2|0 6 9 12 15 18 24 24 27 30 33  |
| |0 0 6 12 18 24 30 36 42 48 54  |
+-+-------------------------------+
|3|0 6  9 12 15 18 21 24 27 30 33 |
| |0 6 12 18 24 30 36 42 48 54 60 |
+-+-------------------------------+
|4|0  6  9 12 15 18 21 24 27 30 33|
| |6 12 18 24 30 36 42 48 54 60 66|
+-+-------------------------------+
|5| 0  6  9 12 15 18 21 24 27 30 33|
| |12 18 24 30 36 42 48 54 60 66 72|
+-+--------------------------------+
|6| 0  6  9 12 15 18 21 24 27 30 33|
| |18 24 30 36 42 48 54 60 66 72 78|
+-+--------------------------------+
```

There are two stages required in calculating the delay bound:

```
   delay =: */@ (f10@s <: g6@(s + tau))"0
   (i.7) delay 10
0 0 0 1 1 1 1
```

The next stage counts the number of zeros returned by *delay*, which equals the number of time intervals for which $f_{10}(s) \leq g_6(s + \tau)$

```
   (i.7) +/@:-.@:delay 10
3
```

Output Bounds

The bound for the output is given by the min-plus *deconvolution* of the arrival and service curve where the deconvolution operation is:

$$(f \oslash g)(t) = \max_{u \geq 0}(f(t + u) - g(u)) \tag{4.37}$$

Note that the min-plus deconvolution operator is not necessarily zero for $t < 0$. In J, we can compute the bound for the output:

```
   u_z_ =: i.@>:@[   NB. define u
   10 max@(f10@(t+u) - g6@u) &> (i:4)
0 0 6 9 12 15 18 21 24
```

This result is shown in Fig 4.6.

4.6 Streaming Video Example

Consider a streaming video application. Every $T = 4$ time units, the sender transmits a frame of size $w = 6$. We can model this arrival curve with the stair function $g_7 = 6u_{4,4}$. We define the arrival g_7 in J as:

```
   g7 =: 6&*@(4 4&stair)
```

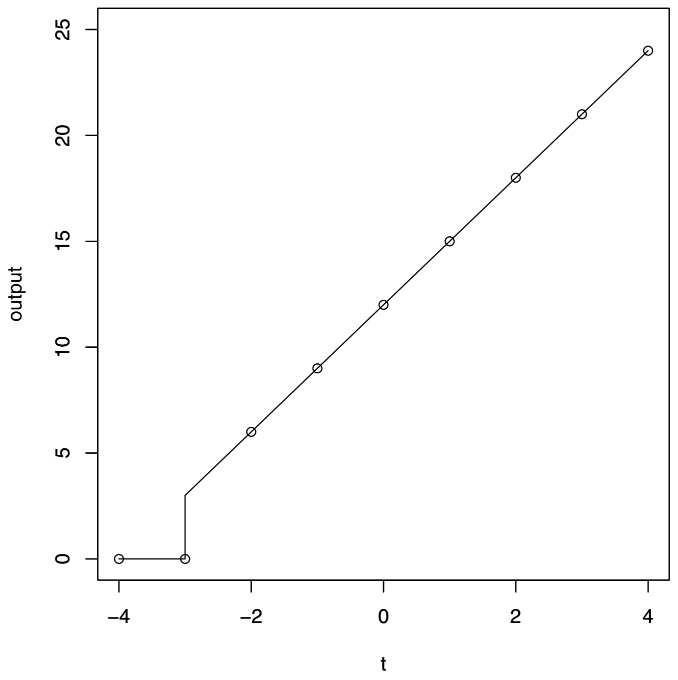

Fig. 4.6. Output bound $f_5 \oslash g_5$

The arrival sequence output by the sender is therefore:

```
   g7 i.11
0 6 6 6 12 12 12 12 18 18 18
```

The sender and receiver are connected via a CBR link with $R = 3$. We use the function $f_5 = \lambda_3$ defined above to represent the service curve of the link.

A function $f \in \mathcal{F}$ is a "good" function if it satisfies, according to Le Boudec and Thiran [36], any of the following:

- f is subadditive and $f(0) = 0$.

  Both $f_5(0) = 0$ and $g_7(0) = 0$:

  ```
     (f5;g7) 0
  0 0
  ```

  The J expressions below show that f_5 and g_7 are subadditive:

  ```
     (f5@(t+s) <: f5@t + f5@s)  10
  ```

```
1 1 1 1 1 1 1 1 1 1 1
   (g7@(t+s) <: g7@t + g7@s)  10
1 1 1 1 1 1 1 1 1 1 1
```

- $f^* = f$, where f^* is the subadditive closure.

  We show that $f_5 = f_5^*$:

```
   fc5 =: 0:`f5`(min@(f5, $:"0@v + $:"0@(t-v)))@.stop
   (f1,: fc5"0) i.11
0 3 6 9 12 15 18 21 24 27 30
0 3 6 9 12 15 18 21 24 27 30
```

  and $g_4 = g_4^*$:

```
   gc4 =: 0:`g4`(min@(g4, $:"0@v + $:"0@(t-v)))@.stop
   (g4,: gc4"0) i.11
0 6 6 6 12 12 12 12 18 18 18
0 6 6 6 12 12 12 12 18 18 18
```

- $f =: f \star f$.

  The J expression below shows that $f_1 = f_1 \star f_1$:

```
  (f1 ,: min@(f1@s + f1@(t-s))"0) i.11
0 3 6 9 12 15 18 21 24 27 30
0 3 6 9 12 15 18 21 24 27 30
```

  and $g_3 = g_3 \star g_3$:

```
   (g3,: min@(g3@s + g3@(t-s))"0) i.11
0 7 10 13 16 19 22 25 28 31 34
0 7 10 13 16 19 22 25 28 31 34
```

- $f = f \oslash f$, where $\oslash$ is the min-plus *deconvolution* operator, given by:

$$f = \min_{0 \leq s \leq t} (f(t+s) - f(s)) \tag{4.38}$$

  It can be seen that $f_1 = f_1 \oslash f_1$:

```
   (f1,: max@(f1@(t+s) - f1@s)"0) i.11
0 3 6 9 12 15 18 21 24 27 30
0 3 6 9 12 15 18 21 24 27 30
```

  However, $g_4 \neq g_4 \oslash g_4$:

```
    (g4,: max@(g4@(t+s) - g4@s)) i.11
   0  6  6  6 12 12 12 12 18 18 18
  18 12 18 18 12 12 18 18 12 18 18
```

When the sender outputs $w = 6$ units of data in a single batch (then it is idle until the next frame is due). However, the CBR link can only process three units of date per time interval. Thus, the sender's traffic flow is constrained by f_1 as well as g_4. The *combined* arrival curve is $f_4 = f_1 \oplus g_4$, which is given by the J expression:

```
   f4 =: f1 min"0 g4
   f4 i.11
0 3 6 6 6 12 12 12 12 18 18
```

However, f_4 does not respresent the tightest bound, as $f_4(5) \leq 6$ and $f_4(6) \leq 12$, whereas $f_4(6)$ should be bounded by nine.

We can check the subadditive properties of f_4. The result of the J expression below shows us that f_4 is not subadditive:

```
   */@ (f4@(t+s) <: (f4@t + f4@s))  &> i.11
1 1 1 1 0 1 1 1 0 1 1
```

The subadditve closure f_4^* of f_4 is given by the J expression:

```
   fc4 =: 0:`f4`(min@(f4 , $:"0@v+$:"0@(t-v))) @. stop
```

Note that `$:` is the self-reference construct. The resultant subadditive closure f_1^* yields a tighter bound than f_4 itself:

```
   fc4 &> i.11
0 3 6 6 6 9 12 12 12 15 18
```

We can verify that the subadditive closure f_4^* is subadditive:

```
   */@ (fc4"0@(t+s) <: (fc4"0@t + fc4"0@s)) &> i.11
```

In this particular case, this bound is also given by $f_1 \star g_4$:

```
   min@(f1@s + g4@(t-s)) &> i.11
0 3 6 6 6 9 12 12 12 15 18
```

The functions g_4, $f_1 \oplus g_4$ and $f_1 \star g_4$ are shown graphically in Fig 4.7.

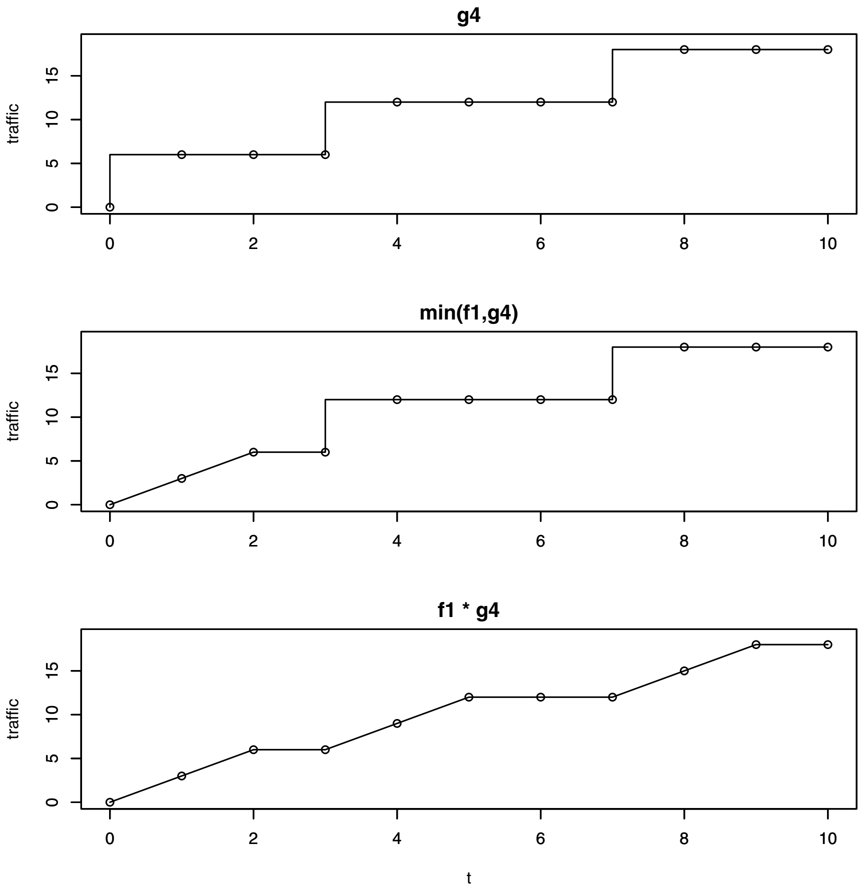

Fig. 4.7. Video frame transmission curve $u_{4,4}$ (top), the pointwise minimum of λ_3 and $u_{4,4}$ (middle) and the convolution of λ_3 and $u_{4,4}$ (bottom)

4.7 Effective Bandwidth and Equivalent Capacity

The effective bandwidth function provides a means of specifying the bit rate requirements of a traffic flow for a given delay bound. For a flow A, expressed as a cumulative function, and a delay bound D, the effective bandwidth is given by:

$$e_D(A) = \max_{0 \leq s \leq t} \frac{A(t) - A(s)}{t - s + D} \tag{4.39}$$

The effective bandwidth for flow A_1 can be calculated for various delay bounds:

```
D =: lhs0  NB. delay D passed as left argument
```

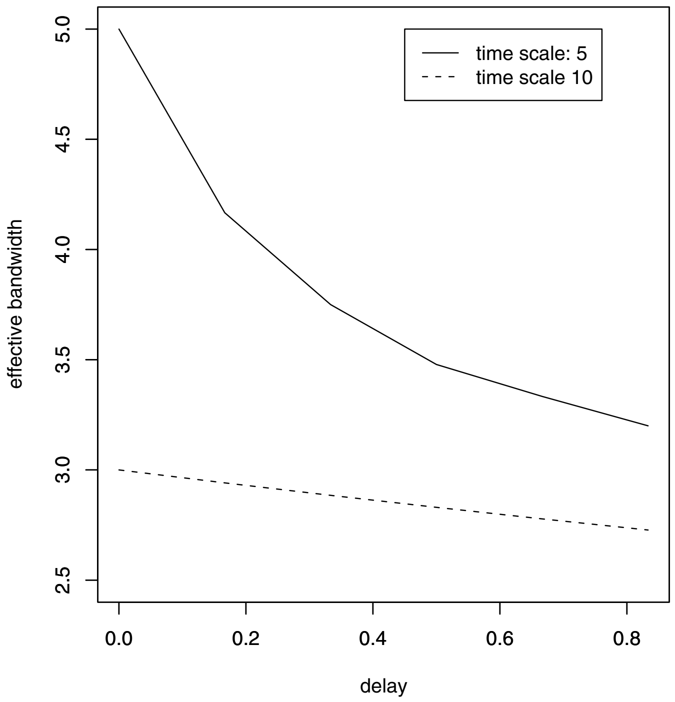

Fig. 4.8. Effective bandwidth of A_1

```
   (i01 i.6) max@((A1@t - A1@s) % (t - s) + D) &> 10
3 2.94118 2.88462 2.83019 2.77778 2.72727
```

We can see that, for zero delay, the effective bandwidth is three, which is the mean rate of A_1 over the interval $[0, 10]$. It diminishes as the tolerable delay bound increases. However, A_1 is not constant over t, if we examine the effective bandwidth over some other time scale (the interval $[0, 5]$ say) the result is different. The effective bandwidth at this time scale is higher than for the previous example. We can see that for zero delay tolerance, the effective bandwidth is five, which is equal to the peak rate of A_1:

```
   (i01 i.6) max@((A1@t - A1@s) % (t - s) + D) &> 5
5 4.16667 3.75 3.47826 3.33333 3.2
```

Figure 4.8 shows the effective bandwidth of the flow A_1.

We can compute the effective bandwidth for an arrival curve. For some "good" function f, the effective bandwidth for delay bound D is:

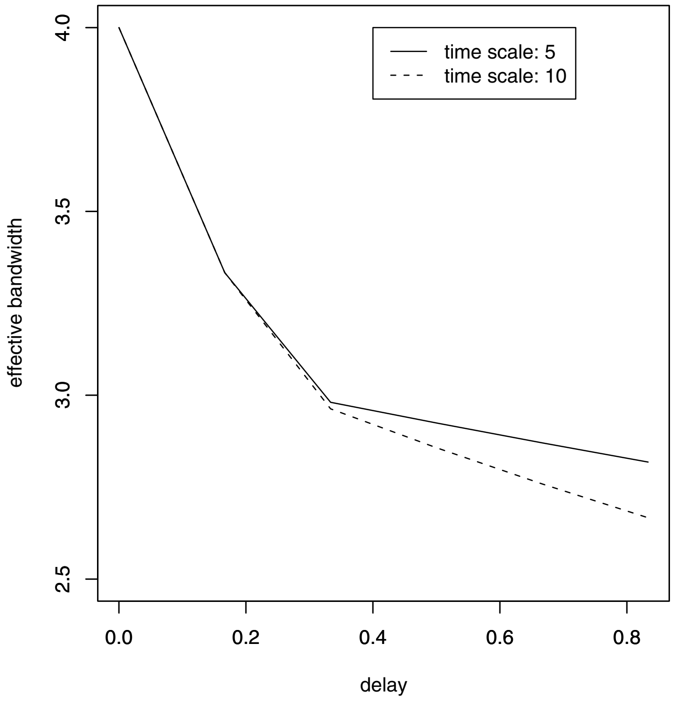

Fig. 4.9. Effective bandwidth of $f_2 = \gamma_{3,1}$

$$e_D(f) = \max_{s \geq 0} \frac{f(s)}{s + D} \tag{4.40}$$

For the affine curve $f_2 = \gamma_{3,1}$ we can compute the effective bandwidth thus:

```
   f2  NB. check the verb definition of f2
3 1&af
   (i01 i.6) max@(f2@s % s + D) &> 10
4 3.33333 2.98077 2.92453 2.87037 2.81818
```

The graph in Fig 4.9 shows the effective bandwidth for $f_2 = \gamma_{3,1}$ over the time scales $t = 5$ and $t = 10$.

The equivalent capacity specifies the bit rate for a flow, given some backlog bound. For a flow A and backlog bound Q, the equivalent capacity is given by the equation below:

$$e_Q(A) = \max_{0 \leq s \leq t} \frac{A(t) - A(s) - Q}{t - s} \tag{4.41}$$

We can compute the equivalent capacity for A_1 for a time scale of $t = 10$ with the J expression:

```
 Q =: lhs0 NB. buffer parameter Q given as left argument
 (i.6) max@(((A1@t - A1@s) - Q) % t - s) &> 10
3 2.9 2.8 2.7 2.6 2.5
```

Just as with the effective bandwidth, the equivalent capacity is higher for a time scale $t = 5$:

```
   (i.6) max@(((A1@t - A1@s) - B) % t - s) &> 5
5 4 3.5 3.25 3 2.8
```

The effective capacity for an arrival curve f is given by the equation:

$$e_Q(f) = \max_{s \geq 0} \frac{f(s) - Q}{s} \tag{4.42}$$

The equivalent capacity for the affine function $f_2 = \gamma_{3,1}$ is:

```
   (i.6) max@((f2@s - Q) % s) &> 100
4 3 2.99 2.98 2.97 2.96
```

Figure 4.9 shows effective bandwidth for f_2 for various time scales.

4.8 Summary

Insights into traffic flows can be gained from network calculus. Flows are characterised by envelope curves. Service curves can be expressed in a similar way. Then, using min-plus algebra, delay bounds can be derived for backlog, delay and output.

Effective bandwidth and equivalent capacity provide a way of determining network resource requirements for flows with determinsitic QoS bounds (in terms of delay and backlog, respectively). We revisit the topic of effective bandwidth in Chapter 6 for deriving bounds that are probabilistic.

5

Stochastic Processes and Statistical Methods

In this chapter we introduce stochastic processes and the statistical methods for analysing them. Specifically, we focus on processes with long-memory, which have been subject to much research in relation to data networks [6, 16] and are considered to be more representative of network traffic (particularly in packet networks) than traditional Markovian models. We introduce the concept of short-range and long-range dependence and how these can be simulated in J.

The long-range dependence property is often accompanied by *self-similarity*.[1] We will refer to short-range dependent and long-range dependent self-similar processes as "srd" and "lrd-ss," respectively. Whereas srd traffic processes tend to smooth rapidly as we "zoom in," lrd-ss processes retain their burstiness.

In this chapter, we develop a number of functions in J for generating and analysing both srd and lrd-ss stochastic processes. We also show how srd and lrd-ss properties can be identified.

5.1 Random Number Generators

Network analysis makes extensive use of simulation techniques and thus relies heavily on random number generators (RNG). In this section, we develop a number of functions for (pseudo) random number generation.

The monadic *roll* primitive `?` returns a random integer in the range 0 to $n-1$, where n is the value of the argument. So, for example, the line below returns six random numbers between 0 and 9:

```
   ? 10 10 10 10 10 10
4 1 3 3 0 6
```

[1] The correct term for the property found in network traffic is asymptotically, statistically self-similar.

Rolling a six-sided die ten times can be simulated by:

```
   >:@? 10 # 6
2 4 3 2 2 4 5 5 4 6
```

The reader should note that the results of examples involving RNG functions, will (most likely) differ from those in this book. The function *rand*, defined below, generates uniformly distributed random numbers in the range 0 to 2^{31}-1:

```
   rand_z_ =: (?@$&2147483646)
```

The verb is monadic, and its argument specifies how many random numbers to generate:

```
   rand 5
592937061 1229445466 1065583194 180909904 260315076
```

Arrays of random numbers can also be generated:

```
   rand 3 2
 463322583 1277891113
1922703664 1984820816
1393776341 1706683049
```

An RNG that generates values in the interval $[0, 1)$ is given by:

```
   runif_z_ =: 2147483647&(%~)@rand
   runif 5
0.120287 0.284969 0.481986 0.188577 0.104365
```

There are a number of methods for generating Gaussian random variates. Here we use the convolution method in [52]:

$$\epsilon = \sqrt{\frac{12}{n}} \sum_{i=1}^{i=n} \mathrm{U}_i(0, 1) - \sqrt{3n} \tag{5.1}$$

where $\mathrm{U}_i(0, 1)$ is a uniformly distributed random variate in the interval $[0, 1)$. This generator exploits the *central limit theorm*, whereby the sum of a sample of n iid (independent, identically distributed) random variates (not necessarily Gaussian) converges to a Normal distribution. The larger the number of variates n, the closer the approximation to normality.[2] The function in Listing 5.1 implements a random number generator for normal random variates of mean zero and variance one:

[2] Clearly there is a trade-off between convergence to normality and the processing cost incurred in calculating each random variate.

Listing 5.1 *Gaussian RNG*

```
cocurrent < 'RNORM'
c1 =: [: _4.24264&+ ]
c2 =: [: 1.41412&* ]
f1 =: [:6&*]
f2 =: runif@f1
g1 =: _6: +/\ ]
g2 =: c1@c2@g1@f2
cocurrent < 'base'
rnorm_z_ =: g2_RNORM_
```

Note, we *hardcode* $n = 6$. The RNG *rnorm* is invoked thus:

```
   rnorm 5
0.683747 0.300376 _0.825842 0.347796 1.09792
```

See [71] for other implementations of Gaussian random number generators. Exponentially distributed random variates [54] are derived:

$$\mathrm{V} = -\log_e \mathrm{U}(0,1) \tag{5.2}$$

The corresponding J verb is simply :

```
   rexp_z_ =: (-@^.)@runif
```

This generates a random variate of mean one, which is confirmed by:

```
   (+/%#) rexp 10000
0.99849
```

If we wish to generate exponential variates for some arbitrary mean, the RNG is:

```
   load 'libs.ijs' NB. load pos. params.
   exprand_z_ =: lhs1*[:rexp]
   (+/%#) 3 exprand 10000 NB. exp variate with mean 3
3.00421
```

A random variable from a geometric distribution is the length of the sequence until the first state change in a sequence of Bernoulli trials. The probability distribution is given by:

$$P(X = x) = p(1-p)^x \qquad x = 0, 1, \ldots, 0 < p < 1 \tag{5.3}$$

Geometric random variates are derived:

$$\mathrm{Geo} = \left[\frac{\mathrm{V}}{\log_e(1-p)}\right] \tag{5.4}$$

where $V = -\log_e U$ is the exponential variate, and Geo is rounded to an integer value. The J verb for generating a sequence of geometric variates is shown in Listing 5.2.

Listing 5.2 *Geometric RNG*

```
cocurrent <'RGEO'
p =: lhs1
n =: rhs1
f1 =: rexp@n
f2 =: ^.@(1:-p)
f3 =: -@f1%f2
f4 =: <.@>:@f3
cocurrent <'base'
rgeo_z_ =: f4_RGEO_
```

The expectation of the geometric RNG is the reciprocal of the probability; that is, $1/p$. Thus:

```
   hd1 =: 'theoretical';'simulation'
   hd1,: (% 0.4); (+/%#) 0.4 rgeo 10000
+-----------+----------+
|theoretical|simulation|
+-----------+----------+
|2.5        |2.498     |
+-----------+----------+
```

The Pareto distribution is a *heavy-tailed* distribution:

$$P(X = x) = \frac{\alpha\beta^{\alpha}}{x^{\alpha+1}} \tag{5.5}$$

where α and β are the shape and scale parameters, respectively. The mean is given by:

$$\mu = \frac{\alpha\beta}{\alpha - 1} \tag{5.6}$$

The J verb below takes β as the left-hand argument and α as the right-hand argument and returns the mean of the Pareto distribution for those parameters:

```
   mpar_z_ =: *%<:@]
   1 mpar 1.1 1.2 1.5 1.7 2
11 6 3 2.42857 2
```

The "heavy-tailed" phenomenon is somewhat ubiquitous within the field of packet data networks [1]. Pareto random variates can be generated from uniformly distributed variates:

$$\mathrm{Par}(\alpha, \beta) = \beta \left[\frac{1}{1 - \mathrm{U}(0,1)} \right]^{1/\alpha} \tag{5.7}$$

The random number generator for Pareto variates is shown in Listing 5.3.

Listing 5.3 *Pareto RNG*

```
cocurrent < 'PAR'
alpha =: lhs1
beta  =: lhs2
n =: rhs1
f1 =: runif@n
f2 =: %@>:@-@f1
f3 =: %@alpha
f4 =: beta*(f2^f3)
cocurrent < 'base'
rpar_z_ =: f4_PAR_
```

The example below generates a number of random Pareto variates for $\alpha = 1.2$ and $\beta = 1$, for which the expectation is six. It can be seen that the resultant sample mean is close:

```
   (+/%#) 1.2 1 rpar 1000000
5.8659
```

5.2 Statistical Functions

In Listing 5.4 we redefine some of the basic statistical functions introduced earlier in Section 3.3.

Listing 5.4 *Statistical Functions*

```
cocurrent < 'z'
mean  =: [: (+/%#) ]          NB. arithmetic mean
mdev  =: -mean                NB. deviations from the mean
sqdev =: [: (*:@mdev) ]       NB. square of deviations
sumsq =: [: (+/@sqdev) ]      NB. sum of squares
dof   =: [: <:@# ]            NB. degrees of freedom
var   =: [: (sumsq % dof) ]   NB. variance
std   =: [: (%: @ var) ]      NB. standard deviation
cocurrent < 'base'
```

We can confirm that *rnorm* returns Gaussian variates with (approximately) zero mean and a standard deviation of one:

```
   hd2 =: 'mean';'std'
   hd2,: (mean;std) rnorm 10000
+-----------+--------+
|mean       |std     |
+-----------+--------+
|_0.00818183|0.995637|
+-----------+--------+
```

Note that most of the functions are enclosed in `[:` and `]`. The *right* verb `]`, which was introduced in Section 2.4, returns the right-hand argument passed to it. The *cap* primitive `[:` is used to *force* a function to behave monadically, even if it is used dyadically. In order to illustrate the difference, we compare the execution of *sqdev* with its *uncapped* equivalent, *sqdev2*:

```
   sqdev2 =: *:@mdev
```

Used monadically, they return the same result:

```
   (sqdev;sqdev2) 1 2 3 4
+-------------------+-------------------+
|2.25 0.25 0.25 2.25|2.25 0.25 0.25 2.25|
+-------------------+-------------------+
```

However, if we execute them both dyadically, we get:

```
   2 (sqdev;sqdev2) 1 2 3 4
+-------------------+----+
|2.25 0.25 0.25 2.25|0.25|
+-------------------+----+
```

For the verb *sqdev2*, the presence of the left argument has caused `-` in *dev* to behave as a dyad; thus it performs a subtraction instead of a (monadic) negation. The mean of the right-hand argument is subtracted from the value of the left-hand argument (in this case, two). That is, the arithmetic operation $2 - \overline{x}$ is performed instead of $x_i - \overline{x}$. In the case of *sqdev*, `-` is unaffected by the left-hand argument and performs a monadic negation function due to the `[: ]` encapsulation. This may not seem important, as running a monadic verb as a dyad makes little sense. However, these verbs may be used to build other functions which are designed to operate as dyads. The presence of a left-hand argument will cause any monadic subfunctions to behave dyadically, resulting in an undesirable change in their intended functionality. In this example, the `-` primtive behaves as a subtraction instead of a negation.

5.2.1 Autocovariance and Autocorrelation

The covariance COV_{XY} and correlation COR_{XY} of the two random variables X and Y are given by:

$$\mathrm{COV}_{XY} = \frac{1}{n-1}\sum_{i=1}^{i=n}(X_i - \overline{X})(Y_i - \overline{Y}) \tag{5.8}$$

$$\mathrm{COR}_{XY} = \frac{\mathrm{COV}_{XY}}{s_X s_Y} \tag{5.9}$$

The implementation of the covariance and correlation functions *cov* and *cor*, are shown below:

```
   sp_z_ =: [: +/ *&mdev
   cov_z_ =: sp % dof
   cor_z_ =: cov % (*&std)
```

The command-lines below compute the covariance and correlation of the two sets of (uniformly distributed) random numbers:

```
   X =: rnorm 100
   Y =: rnorm 100
   hd3 =: 'covariance';'correlation'
   hd3,: Y (cov;cor) X
+----------+-----------+
|covariance|correlation|
+----------+-----------+
|_0.0180343|_0.0166897 |
+----------+-----------+
```

As expected, the covarianace and correlation are low. The autocovariance $c(k)$ and autocorrelation coefficient $r(k)$ are defined below:

$$c(k) = \frac{1}{n}\sum_{t=k+1}^{t=n}(X_t - \overline{X})(X_{t-k} - \overline{X}) \tag{5.10}$$

$$r(k) = \frac{\sum_{t=k+1}^{t=n}(X_t - \overline{X})(X_{t-k} - \overline{X})}{\sum_{t=k+1}^{t=n}(X_t - \overline{X})} \tag{5.11}$$

where k is the lag. The corresponding J verbs for $c(k)$ and $r(k)$ are *autocov* and *autocor*, respectively. These verbs are defined in Listing 5.5.

Listing 5.5 *Autocovariance and Autocorrelation*

```
cocurrent < 'ACF'
f1 =: }. ,: -@[}. ]  NB. separate X(k) and X(k-t)
f2 =: f1-mean        NB. sub mean from X(k) and X(k-t)
n =: #@]             NB. count no. data items
sp =: +/@(*/@f2)     NB. sum of products
cocurrent < 'base'
autocov_z_ =: sp_ACF_ % n_ACF_
autocor_z_ =: sp_ACF_ % sumsq
```

The autocovariance is the sum of the products (*sp*) of the deviations, divided by the length of the times series. The autocorrelation is the sum of the products of the deviations, divided by the sum of the squares. For example, the autocovariance and autocorrelation for 10,000 normal deviates at lag $k = 1$, can be computed:

```
   X =: rand 10000 NB. generate 10000 random numbers
   hd4 =: 'autocov';'autocor'
   hd4,: 1 (autocov; autocor) X NB. c(1) and r(1)
+----------+----------+
|autocov   |autocor   |
+----------+----------+
|5.77953e14|0.00150565|
+----------+----------+
```

We can apply these functions over a range of lags. Using *autocor* to illustrate, we define k as a *column* list:

```
   ]k =: col i. 4   NB. define lags k
0
1
2
3
```

Giving due consideration to the rank attribute, we execute *autocor*:

```
   k autocor"1 X     NB. compute r(k) for k={0,1,2,3}
1 0.0165008 0.00945115 _0.00588122
```

The graph in Fig 5.1 shows the autocorrelation coefficients for an iid random variable with $k = 0, 1, \ldots, 100$. It can be seen that, for lags $k > 0$, the autocorrelations are small, which means that serial correlations are negligible. This is what we would expect for an iid random variable.

5.2.2 Variance Time Plot

A variance time plot shows the rate at which the variance of a time series decays over different time scales. The aggregated process $X^{(m)}$ is the averaged values of X grouped into non-overlapping blocks of m; that is:

$$X^{(m)}(k) = 1/m \sum_{i=(k-1)m+1}^{km} X(i) \quad k = 1, 2, \ldots \tag{5.12}$$

The J function below calculates var$[X^{(m)}]$, taking the block size m as the left-hand argument and $X(t)$ as the right-hand argument:

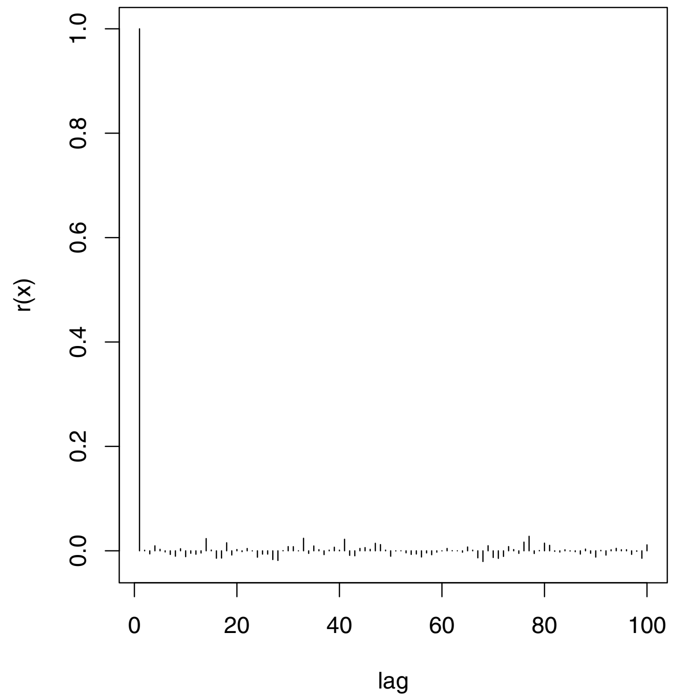

Fig. 5.1. Autocorrelation coefficient

```
cocurrent < 'VARM'
Xt =: rhs0
m =: lhs1
```

Dividing $X(t)$ into nonoverlapping blocks of m results in T/m blocks. If T mod $m = 0$, then each block will contain exactly m elements. If, however, T mod $m > 0$, then there are $\lfloor T/m \rfloor$ blocks of m and one block of T mod m. That is, if T is not exactly divisible by m, then we are left with a *runt* block at the end of the sequence. In the event that T is not exactly divisible by m we discard the runt block; thus obtaining, from $X(t)$, the sequence:

$$X^*(t) = \{X(1), X(2), \ldots, X(\tau)\} \tag{5.13}$$

where $\tau = m \times \lfloor T/m \rfloor$. $X^*(t) = X(t)$ if T mod $m = 0$. In practice, therefore:

$$X^{(m)}(k) = 1/m \sum_{i=(k-1)m+1}^{km} X^*(i) \qquad k = 1, 2, \ldots \tag{5.14}$$

This *truncation* function is performed by the J verbs:

```
   g1 =: (#@Xt)         NB. length of X(t)
   g2 =: <.@(g1%m)      NB. floor(T/m)
   g3 =: g2*m           NB. m * floor(T/m)
   g4 =: (i.@g3) { Xt   NB. truncate X(t)
```

The verb *varm* returns the variance of $X^{(m)}$:

```
   f1 =: (-@m) +/\ g4 NB. sum values in each block of m
   f2 =: f1%m         NB. divide by m (the average)
   f3 =: var"1@f2     NB. var[X^(m)(i)]
   cocurrent < 'base'
   varm_z_ =: f3_VARM_
```

The central limit theorm states that the variance of the sample mean decays as the reciprocal of the sample size m; that is:

$$\text{var}(\overline{X}) = \sigma^2 m^{-1} \tag{5.15}$$

For the purpose of illustration we generate 10,000 Gaussian random variates and find the variance-time plot:

```
   X =: rnorm 10000
   m =: col 10 to 100
   vx =: m varm"1 X
```

We take logs of both sides and calculate the least squares linear regression coefficients using the *matrix divide* verb `%.`, thus:

```
   hd5 =: 'intercept';'slope'
   hd5,: ;/ (log10 vx) %. 1,.log10 m
+----------+--------+
|intercept |slope   |
+----------+--------+
|0.00887776|_1.02227|
+----------+--------+
```

The slope is approximately -1, which is what we would expect for an iid random variable. Figure 5.2 shows the variance time plot on a log-log scale.

5.2.3 Fourier Transform and Power Spectrum

Analysis in the frequency domain also provides valuable insights into the nature of the time series. The discrete fourier transform is given by:

$$H(k) = \sum_{j=0}^{j=n-1} h(j)e^{-2\pi ijk/n} \qquad k = 0, \dots, n-1 \tag{5.16}$$

The discrete fourier transform is shown in Listing 5.6. Recall that *eu* is Euler's formula $e^{i\theta} = \cos\theta + i\sin\theta$:

Listing 5.6 *Discrete Fourier Transform*

```
cocurrent < 'DFT'
h =: rhs0
k =: lhs1
n =: #@h
j =: i.@n  NB. j=1,2,..,n-1
eu =: ^@j. NB. Euler's formula
f1 =: -@(+:@o.) @ (j*k%n) NB. -2pi jk/n
f2 =: h * (eu@f1)         NB. h*exp(-2pi jk/n)
f3 =: +/@f2
cocurrent < 'z'
dft   =: f3_DFT_      NB. in complex form
dftmp =: *.@dft       NB. amplitude and phase
dftm  =: {."1 @ dft  NB. amplitude only
dftp  =: {:"1 @ dft  NB. phase only
cocurrent < 'base'
```

Define a square wave and a range of frequencies:

```
   sqw =: 1 1 1 1 _1 _1 _1 _1
   freq =: col i.8
```

Calculate the transform of the square wave, returning the result in amplitude and phase form:

```
   freq dftmp"1 sqw
          0         0
    5.22625   _1.1781
          0         0
    2.16478 _0.392699
          0         0
    2.16478  0.392699
3.77476e_15         0
    5.22625    1.1781
```

A useful analysis tool for time series is the *power spectrum*. The power spectrum derived from the fourier transform of the autocorrelation coefficients:

$$S_h(k) = \sum_{j=0}^{j=n-1} r_h(j+1)e^{-2\pi ijk/n} \tag{5.17}$$

Thus, for an iid random variable (with a Gaussian distribtution), the power spectrum can be computed:

```
   X =: rnorm 10000
   ax =: (col 1 to 100) autocor"1 X
   sx =: (col 1 to 100) dft"1 ax
```

The graph in Fig 5.3 shows the power spectrum for the time series X. It can be seen that the power spectrum is (nearly) *flat* across the frequency range. This is what we would expect and is indicative of white noise. We can confirm this by taking logs and calculating the slope of the graph:

```
   hd5,: ;/ (log10 sx) %. 1,.(log10 1 to 100)
+---------+----------+
|intercept|slope     |
+---------+----------+
|_3.1693  |_0.0164034|
+---------+----------+
```

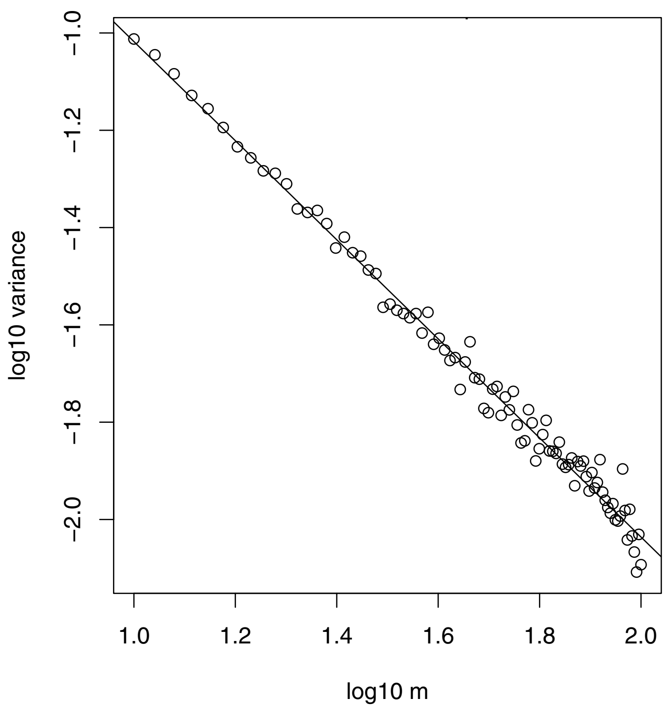

Fig. 5.2. Variance time plot

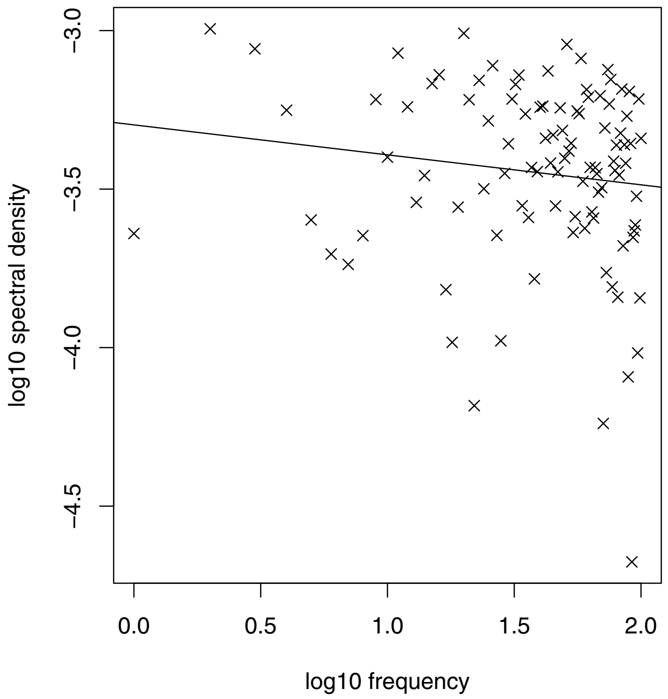

Fig. 5.3. Spectral density of an iid random variable (white noise)

5.3 Stochastic Processes

In Section 5.1, we developed functions for generating (pseudo) random number sequences. It is fairly straightforward to use these functions to generate stochastic processes with independent variates; for example a white noise process:

$$X(t) = \mu + \epsilon(t) \tag{5.18}$$

Thus, a white noise process, centered around the value of two, can be generated by:

```
   2 + rnorm 5
2.92504 1.9345 _0.5072 2.48281 3.52364
```

Such processes are uncorrelated in time. Consequently, the autocorrelation coefficient is $r(k) = 0$ for $k > 0$. In this section, we introduce stochastic processes that exhibit serial correlation, both short-range and long-range. Long-range dependent data is of particular interest to data communications because network traffic has been shown to exhibit these properties.

Autoregressive (AR) and moving average are two types of process that exhibit short-range dependence; that is, their autocorrelation coefficients decay exponentially quickly with increased lag:

$$r(k) \sim \alpha^{-k}, \quad k > 0 \text{ and } 0 < \alpha < 1 \tag{5.19}$$

Only the state of the system in the near past has any significant influence on the current state. An autogressive process of order p, denoted by AR(p), is given by the expression:

$$X(t) = \sum_{i=1}^{i=p} \phi_i X(t-i) + \epsilon(t) \tag{5.20}$$

Similarly, MA(q) is a moving average process of order q:

$$X(t) = \epsilon(t) + \sum_{i=1}^{i=q} \theta_i \epsilon(t-i) \tag{5.21}$$

Both processes can be combined to form an ARMA process:

$$X(t) = \epsilon(t) + \sum_{i=1}^{i=p} \phi_i X(t-i) + \sum_{i=1}^{i=q} \theta_i \epsilon(t-i) \tag{5.22}$$

which can be re-written as:

$$\left(1 - \sum_{i=1}^{i=p} \phi_i B^i\right) X(t) = \left(1 + \sum_{i=1}^{i=q} \theta_i B^i\right) \epsilon(t) \tag{5.23}$$

where B is the backshift operator, such that $B^i = X(t-i)$. ARMA processes are special cases of the more generalised autoregressive integrated moving mverage (ARIMA) processes. The ARIMA(p, d, q) process is an integrated AR(p) and MA(q) process with a differencing parameter d:

$$\left(1 - \sum_{i=1}^{i=p} \phi_i B^i\right) (1-B)^d X_t = \left(1 + \sum_{i=1}^{i=q} \theta_i B^i\right) \epsilon_t \tag{5.24}$$

ARIMA and ARMA are equivalent when $d = 0$. If $0 < d < 1$, then the process is a *fractional* ARIMA process and exhibit properties of *long-range dependence* and statistcial *self-similarity* (lrd-ss). The J functions *autocor*, *varm* and *dft*, developed in Section 5.2, enable us to analyse time series and determine the presence of lrd-ss.

5.3.1 Autoregressive Processes

Define a list x and a list of coefficients ϕ:

```
   x =: 8.4 5.5 _15.2 14.1 3.8 _10.4 _3.6 _7.4
   phi =: 1 0.4 0.2 0.1   NB. coefficients
```

We must include 1 as the first coefficient. We *stitch* the random numbers and the coefficients together to form a matrix to be passed to *ar*:

```
   ]Y =: x ,: (|.phi)
8.4 5.5 7.1 _15.2 14.1 3.8 _10.4 _21.7 _3.6 _7.4
0.1 0.2 0.4     1    0   0     0     0    0    0
```

A desirable effect of the *stitch* operation is that the coefficients are padded with zeros, up to the length of x. When we calculate $X(0)$, we not only need a corresponding $\epsilon(0)$, but also $\{X(-1),\ldots,X(-p)\}$. As we do not have these terms, we simply substitute $\{\epsilon(-1),\ldots,\epsilon(-p)\}$ instead. We, therefore, use the first p values of x. Also, x represents $\epsilon(t)$ for $t=\{0-p,1-p,\ldots,n-p\}$. The functions below extract the time series $\epsilon(t)$ and the coefficients:

```
   cocurrent <'AR'
   et    =: rhs1
   coefs =: rhs2
```

The autoregressive term $X(t)$ is just the *inner product* of the coefficients and (past) time series. Thus $X(0)$ can be calculated:

```
   g1 =: coefs ip et
   g1 Y_base_
9.96
```

The power conjunction `^:` is used to calculate subsequent terms. We need, therefore, to output another matrix of the modified time series and the coefficients. The coefficients need to be *shifted* forward (in time):

```
   cnext =: 0:,}:@coefs NB. shift
   cnext Y_base_        NB. test it
0 0.1 0.2 0.4 1 0 0 0 0 0
```

The term $\epsilon(t)$ is substituted for $X(t)$; that is, we replace the corresponding value in x for the newly computed autoregressive term. In order to do this, we need to calculate the index t which is the current time in the series. The functions below compute two *masks*, one that represents the past and present values (of x) and one that is for future values:

```
   f1 =: *@ coefs
   f2 =: |.@f1
   f3 =: +/\@f2
   mask1 =: |.@(*@f3)
   mask2 =: not@mask1
```

The example below demonstrates the two mask functions:

```
   (mask1,:mask2) Y_base_
1 1 1 1 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1
```

The first four elements of X represent the past and present values of the AR process; that is $X(t), X(t-1), \ldots$, while the last six elements represent $\epsilon(t+1)$, $\epsilon(t+2)$ etc. The J verb below implements the AR process:

```
   f5 =: <:@(+/@mask1)
   f6 =: +/@mask2
   f7 =: f5 {. et      NB. -p to (i-1) elements
   f8 =: (-@f6) {. et  NB.(i+1) to n elements
   xnext =: f7,g1,f8   NB. insert u(i)
   cocurrent < 'base'
   ar_z_ =: xnext_AR_,:cnext_AR_
```

This example demonstrates the evolution of the function for the first four iterations:

```
   ar^:(i.4) Y
8.4 5.5 _15.2 14.1   3.8   _10.4 _3.6 _7.4
0.1 0.2   0.4    1     0       0    0    0

8.4 5.5 _15.2 9.96   3.8   _10.4 _3.6 _7.4
  0 0.1   0.2  0.4     1       0    0    0

8.4 5.5 _15.2 9.96 5.294   _10.4 _3.6 _7.4
  0   0   0.1  0.2   0.4       1    0    0

8.4 5.5 _15.2 9.96 5.294 _7.8104 _3.6 _7.4
  0   0     0  0.1   0.2     0.4    1    0
```

The coefficients “move” forward after each iteration. Successive values of ϵ_t are replaced with their corresponding autoregressive terms $X(t)$. Running the function for eight iterations, removing the coefficients and stripping off the *history* terms gives:

```
   2 4 $ _8 {. rhs1(ar^:(8) X)
  8.4     5.5    _15.2      9.96
5.294 _7.8104 _4.66936 _0.163395
```

Note that the shape term `2 4 $` is used merely for brevity.

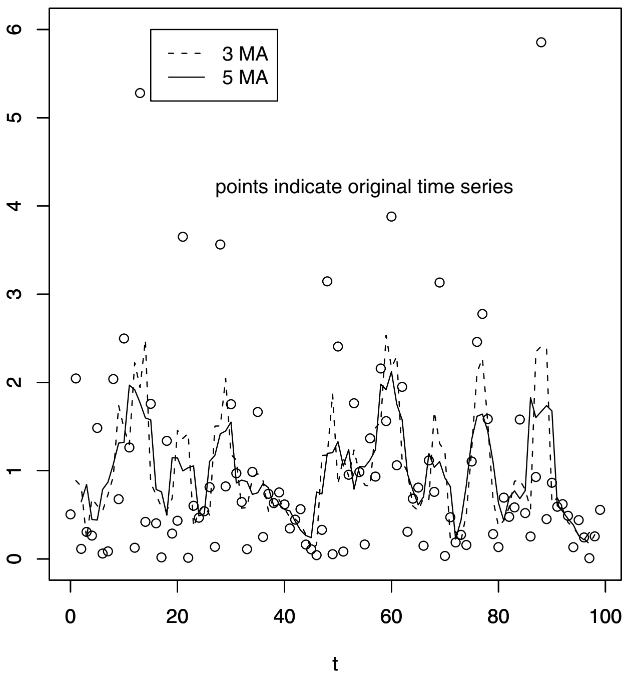

Fig. 5.4. Using a moving average for smoothing

5.3.2 Moving Average Processes

A *simple* moving average function for the basic smoothing of time series data is fairly straightforward to implement in J:

```
   sma_z_ =: +/\%[
```

Generate seven exponentially distributed random variates and *smooth* them with a 3 MA process:

```
   ]X =: rexp 6
2.80561 1.5806 0.973211 2.19895 0.870357 0.0316778
   3 sma X
1.78647 1.58425 1.34751 1.03366
```

With the simple MA, each value of X has equal *weight*, in this case $1/3$:

$$X^{(3\,\mathrm{MA})}(i) =: \frac{X_{i-1} + X_i + X_{i+1}}{3} \tag{5.25}$$

The graph in Fig 5.4 shows the 3 MA and 5 MA for time series of exponential distributed random variables. Weighted moving averages in Equation (5.21) is implemented in Listing 5.7.

Listing 5.7 *Weighted moving average*

```
cocurrent < 'MA'
f1 =: (#@lhs0) +\ rhs0
f2 =: f1 ip [
cocurrent < 'base'
wma_z_ =: f2_MA_
```

A moving average of order three, MA(3) with $\theta_1 = \theta_2 = \theta_3 = 1/3$ yields the same result as the simple 3 MA:

```
   1r3 1r3 1r3 wma X
1.78647 1.58425 1.34751 1.03366
```

Note the different use of terminology. For a simple smoothing MA, we use q MA, where q is order. However, for a weighted MA of order q, we use MA(q).

5.3.3 Processes with Long-Memory

Time series with long-memory can be generated using an ARIMA process with a *fractional* differencing parameter $0 < d < 1/2$. This special case of an ARIMA process is called a fractional ARIMA process or FARIMA(p, d, q). Furthermore, properties of long-memory do not require an AR or MA component; that is, a FARIMA$(0, d, 0)$ process is sufficient to produce long-memory. Thus, we can simplify Equation (5.23):

$$X(t)(1 - B)^d = \epsilon(t) \tag{5.26}$$

The binomial expansion of $(1 - B)^d$ [38] is:

$$(1 - B)^d = \sum_{k}^{\infty} \binom{d}{k} (-1)^k \cdot B^k \tag{5.27}$$

The compution of the coefficients of the binomial expansion $\binom{d}{k}$ is performed by the ! verb, the first five coefficients being:

```
   d =: 0.3 [ k =: i.5
   k ! d
1 0.3 _0.105 0.0595 _0.0401625
```

However, what we actually need to find are the coefficients for the expansion of $(1 - B)^{-d}$, as we are ultimately computing $X(t)$, where:

$$X(t) = \frac{\epsilon(t)}{(1 - B)^d}$$

The J expression below yields the sequence of terms $\binom{d}{k}(-1)^k$:

```
   (_1 ^ k) * k ! -d
1 0.3 0.195 0.1495 0.123337
```

The verb *fdcoefs* in Listing 5.8 computes the terms of the expansion for d and k. The fractional differencing function *fdiff* is shown in Listing 5.9.

Listing 5.8 *Coefficients*

```
cocurrent < 'FDCOEFS'
f1 =: (i.@]) ! (-@[)
f2 =: _1: ^ i.@]
cocurrent < 'base'
fdcoefs_z_ =: f1_FDCOEFS_ * f2_FDCOEFS_
```

Listing 5.9 *Fractional Differencing Function*

```
cocurrent < 'ARIMA'
c =: lhs0
nc =: #@c
en =: [: >@{. ]
xt =: [: >@{: ]
xn =: xt {~i.@ <:@nc
et =: {.@en
enext =: }.@en
g1 =: c*(et,xn)
g2 =: +/@g1
g3 =: enext;(g2,xt)
cocurrent < 'base'
fdiff_z_ =: g3_ARIMA_
```

We show how the fractional differencing function works with the power conjunction. First we compute (the first five) coefficients for $(1 - B)^{-d}$:

```
   cf =: 0.3 fdcoefs 5
```

Passing the coefficients as the left-hand argument we call *fdiff*:

```
   x =: (rnorm 10); rnorm 5
   4 4 $ >rhs2 (cf fdiff^:(8) x)
_0.786457  _1.72396  _1.32671 0.341925
 0.971792 _0.426737  _1.56226 _2.71365
_0.912689  _1.39846 _0.930819 _1.09579
  1.39835 _0.786457  _1.72396 _1.32671
```

As *fdiff* returns $e(t)$ and $X(t)$ in *boxed* format, the term `>rhs2` extracts $X(t)$ and unboxes it (again the term `4 4 $` is merely for brevity).

The wrapper function in Listing 5.10 simplifies the execution of *fdiff*. The explicit function *fd* takes the fractional differencing paramter d as the left-hand argument and the number of coefficients and the length of the time series as the right agruments.

Listing 5.10 *Fractional Differencing Wrapper Script*

```
cocurrent < 'z'
fd=: 4 : 0
'nc nx' =: y.
d =: x.
c =: d fdcoefs nc
e1 =: rnorm nx
e2 =: rnorm nc
|. (i.nx) { >{: (c fdiff^:(nx) e1;e2)
)
cocurrent < 'base'
```

We generate a FARIMA$(0,d,0)$ time series and analyse it with the statistical tools developed earlier in this chapter. Processes with long-range dependence and self-similarity are characterised by the *Hurst* parameter $0.5 \leq H \leq 1$, where $H \to 1$ indicates a high degree of long-range dependence and self-similarity. $H \to 0.5$ indicates short-range dependence (autocorrelations that decay exponentially) or even independence. We show how H can be estimated from both the time (variance time plot) and frequency (power spectrum) domain.

The relationship between the Hurst parameter and fractional differencing parameter is $H = 1/2+d$, so setting $d = 0.3$ should result in $H = 0.8$. We generate a sequence of 10,000 values of a FARIMA$(0,0.3,0)$ process with the command:

```
   y =: 0.3 fd 170 10000
```

There is little point in using more than 170 coefficients as the reciprocal of 171 factorial is *very* small; so small, for the level of precision in J, it is zero:

```
   %@! 170 171  NB. the reciprocol of 171! is zero
1.3779e_307 0
```

Nonsummable autocorrelations mean that the autocorrelation coefficients diminish as a power law:

$$r_k \sim k^{-\zeta}, \quad \text{where } k \geq 1 \text{ and } 0 < \zeta < 1 \tag{5.28}$$

As we have seen in Section 5.2.2, for independent times series, variances decay as the reciprocol of the sample size m (consistent with the central limit theorem). However, for time series that are self-similar, variances decay more slowly (with slope $-\beta$):

$$\text{var}[X^{(m)}] = m^{-\beta} \quad \text{where } \beta < 1 \tag{5.29}$$

Calculate the variance time plot (for $m = 10, 11, \ldots, 100$):

```
   m =: col 10 to 100
   vy =: m varm"1 y
```

Take logs of var$[X^{(m)}]$ and m and find the slope β using linear regression:

```
   hd5,: ;/ (log10 vx) %. 1,.log10 m
+---------+---------+
|intercept|slope    |
+---------+---------+
|0.133737 |_0.431246|
+---------+---------+
```

Thus $\beta = 0.431246$. The Hurst parameter H is related to the slope β:

$$H = 1 - \beta/2 \tag{5.30}$$

Writing the Hurst parameter calculation as a function:

```
   hurst_z_ =: >:@-@-:
   hurst 0.431246
0.773587
```

For a fractional differencing parameter $d = 0.3$, we would expect a Hurst parameter of 0.8. From this simulation, we get $H \approx 0.77$. Self-similar processes resemble *fractional* $1/f$ noise near the origin. That is, for low frequencies the power spectrum:

$$S(f) \sim |f|^{-\gamma} \tag{5.31}$$

where γ is fractional ($0 < \gamma < 1$). The power spectrum can be computed by:

```
   k =: col 1 to 200
   freq =: col 1 to 100
   ry =: k autocor"1 y  NB. autocorrelation
   gamma =: freq dft"1 ry  NB. power spectrum
```

Take logs and find the slope γ:

```
   hd5,: ;/ (log10 gamma) %. 1,.log10 freq
+---------+--------+
|intercept|slope   |
+---------+--------+
|_1.68467 |_0.65241|
+---------+--------+
```

The Hurst parameter is related to the slope γ by:

$$H = \frac{1+\gamma}{2} \tag{5.32}$$

Thus the corresponding J function for finding the Hurst parameter from the slope of the power spectrum is:

```
   hurst2 =: -:@>:
```

From the frequency domain, we can confirm that H is close to 0.8:

```
   hurst2 0.65241
0.826205
```

The graphs in Fig 5.5 show the time series plot for a FARIMA(0,0.3,0) process (top left), autocorrelation function (top right), power spectrum (bottom left) and variance time plot (bottom right).

5.4 Queue Analysis

We define network flow $Y_i = \{Y_i(t), t = 0, 1, \ldots\}$ that, in any time interval t, either transmits at a rate of one, or is silent. If $X(t)$ is an FARIMA$(0, d, 0)$ process, then:

$$Y_i(t) = \begin{cases} 0 & \text{if } X(t) < 0 \\ 1 & \text{if } X(t) \geq 0 \end{cases} \tag{5.33}$$

As $X(t)$ is centered around the origin (zero mean), the average transmission rate of $Y_i(t)$ is $1/2$. The J script in Listing 5.11 is a traffic simulation for long-range dependent flows. It generates a number of (FARIMA) flows (using the fractional differencing verb *fd*) and returns the aggregate.

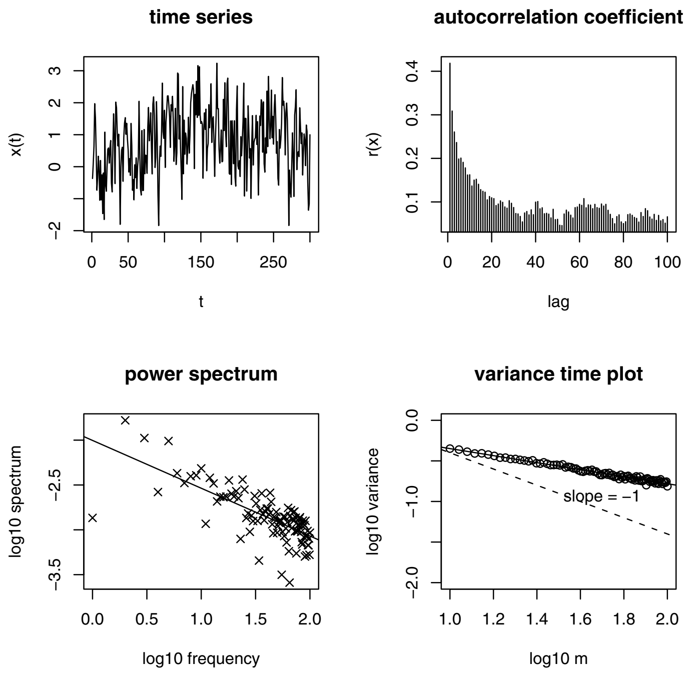

Fig. 5.5. Time series for an FARIMA(0,0.3,0) process (top left), autocorrelation function (top right), power spectrum (bottom left) and variance time plot (bottom right)

Listing 5.11 *Traffic Simulation Script*

```
cocurrent < 'z'
ts =: 4 : 0
'n l' =. y.
d =. x.
a =. 0
for_j. i. n
do. a =. a + 0> d  fd 170&,l
end.
a
)
cocurrent < 'base'
```

The function takes a number of arguments, d is the fractional differencing parameter (passed as a left-hand argument), and n and l are the number of flows and the flow length (number of time intervals over which we observe the flows), respectively.

Thus, we generate an aggregate traffic flow of 20 flows, over a period of 2000 time intervals with a fractional differencing parameter $d = 0.45$, using the command:

```
   y1=:0.45 ts 20 2000
```

We can use the variance time plot to determine the amount of lrd-ss in the aggregate process.

```
   vy1 =: m varm"1 y1
   hd5,: ;/ (log10 vy1) %. 1,.log10 m
+---------+---------+
|intercept|slope    |
+---------+---------+
|0.809115 |_0.476281|
+---------+---------+
   hurst 0.476281
0.76186
```

The Hurst parameter yields a somewhat lower value than expected. Using a fractional differencing parameter $d = 0.45$, we would expect a value of H closer to 0.95. However, multiplexing is known to reduce burstiness [10]. Moreover, this is a somewhat crude method of generating lrd-ss network traffic. More effective methods will be discussed later.

As this aggregate flow of traffic arrives at the buffer queue of a work conserving communications link, the size of the queue $q(t)$ at time t is given by the Lindley equation [39]. We can, therefore, analyse the queue with:

```
   load 'netcalc.ijs' NB. load lindley function
   q1 =: (12;y1) lindley^:(2000) 0
```

In order to gain an insight into the effects that lrd-ss traffic have on the backlog, we need to compare it against a traffic flow derived from a Markovian model (such as a Poisson process). However there is a simpler (and possibly more effective) method of comparing lrd-ss with srd traffic. If we take our original lrd-ss time series y_1, and randomly sort it, then the serial correlations in the sequence will be removed. The J function below randomly sorts a list of values:

```
   rsort_z_ =: {~ ?~@#
   y2 =: rsort y1
```

We can check that the variances of a new time series decays as the reciprocol of m to confirm that we have removed the long-range dependence and self-similarity:

```
   hd5,: ;/ (log10 vy2) %. 1,.log10 m
+---------+--------+
|intercept|slope   |
+---------+--------+
|0.63591  |_1.04039|
+---------+--------+
```

The slope of the decay on a log-log plot is close to -1, so the sequence is short-range dependent (srd). However, note that y_1 and y_2 contain the same volume of data:

```
   +/ y1,.y2
19786 19786
```

Consequently, they also have the same mean and peak rate:

```
   hd6 =: '';'y1';'y2'
   hd7 =: 'mean';'max'
   hd6, hd7,. 2 2 $ ;/ (mean, max) y1,.y2
+----+------+------+
|    |y1    |y2    |
+----+------+------+
|mean|9.8905|9.8905|
+----+------+------+
|max |18    |18    |
+----+------+------+
```

The sequences y_1 and y_2 differ by virtue of the reordering of y_2. Analyse the queue for the srd (randomly sorted) traffic arrival process:

```
   q2 =: (12;y2) lindley^:(2000) 0
```

It can be seen that the mean and peak queue size of the two arrival processes is significantly different:

```
   hd6, hd7,. 2 2 $ ;/ (mean, max) q1,.q2
+----+-------+--------+
|    |y1     |y2      |
+----+-------+--------+
|mean|1.28236|0.272364|
+----+-------+--------+
|max |31     |7       |
+----+-------+--------+
```

Clearly, if we had assumed that the arrival process was Markovian, when it was actually lrd-ss, we would have underestimated the required buffer queue (setting it to 7 rather than 31). This would have resulted in a significantly greater number of dropped packets than expected. The graph in Fig 5.6 clearly shows the difference in the backlog of y_1 and y_2. From a capacity planning point of view, the lrd-ss traffic arrival process needs a faster link to keep the backlog low:

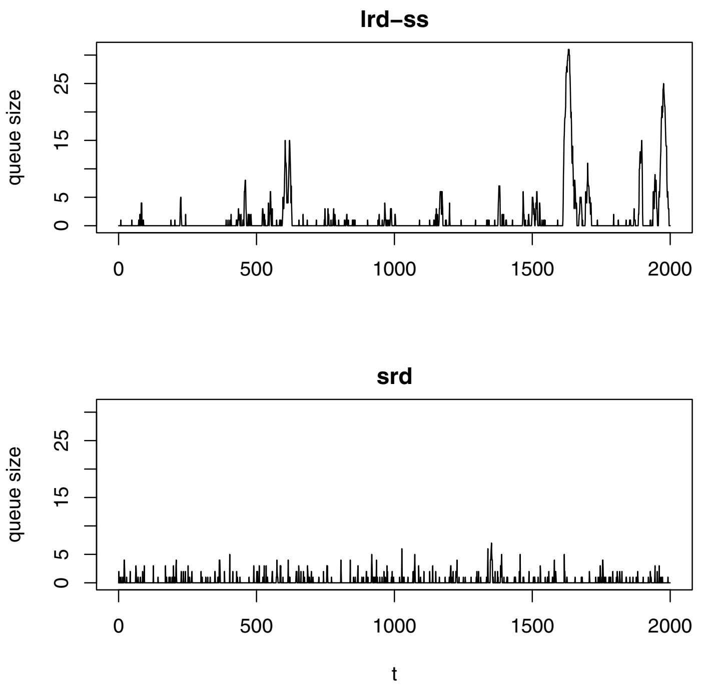

Fig. 5.6. Queue analysis for lrd-ss and srd traffic

```
   q3 =: (14;y1) lindley^:(2000) 0
   hd7,: (mean;max) q3
+--------+---+
|mean    |max|
+--------+---+
|0.045977|6  |
+--------+---+
```

5.5 Summary

The focus of this chapter was stochastic processes relevant to modeling network traffic and the statistical methods for analysing them. We introduced the concepts of short-range dependence and long-range dependence/self-similarity. We developed J functions for generating time series based upon AR and MA models. These resulted in srd time series. Time series with lrd-ss properies were generated using fractional differencing methods (FARIMA), whereby the degree of lrd-ss is determined by the fractional differencing parameter d.

6

Traffic Modeling and Simulation

Network traffic is often modelled as Markov processes. The problem with these models is that they do not necessarily capture the statistical properties of actual network traffic. It has been widely reported that network traffic exhibits *fractal* properties [6, 13, 37], that is, they have nonsummable autocorrelations, slowly decaying variances, spectral densities that obey a power law near the origin, and heavy-tailed distributions.

In this chapter, we introduce discrete on/off models for simulating traffic sources. Short-range dependent (srd), and thus Markovian, traffic models can be generated by sampling on and off periods from geometric distributions. Traffic with long-range dependence and self-similar (lrd-ss) properties can be generated by replacing the geometric distribution with a *heavy-tailed* distribution (such as the Pareto distribution). J functions are developed for simulating srd traffic with Geo[on]-Geo[off] times and lrd-ss traffic with Geo[on]-Par[off] times. We use the functions developed in Chapter 5 to analyse the simulated traffic in this chapter.

Chapter 4 introduced the concept of effective bandwidth, as a means of calculating resource requirements for flows that expect deterministic delay bounds. Effective bandwidth measures were derived, either from an arrival sequence or an arrival curve. In this chapter we revisit effective bandwidth and show how resource requirements can be derived for probabilistic QoS bounds.

6.1 On/Off Traffic Sources

A common network traffic model is the *on/off* model. During the on periods a source transmits at some (constant) peak rate r_{peak} and is idle during the off periods. Thus, the transmission rate of the on/off source $r(t)$ at any time t is:

$$r(t) = \begin{cases} r_{\text{peak}} & \text{on period} \\ 0 & \text{off period} \end{cases} \tag{6.1}$$

The probability p_{on} that the system is in an on period is given by:

$$p_{\text{on}} = \frac{\overline{r}}{r_{\text{peak}}} \tag{6.2}$$

where $\overline{r}$ is the mean rate. Consequently, the probability of an off period is $p_{\text{off}} = 1 - p_{\text{on}}$. A random sequence of binary outcomes $S = \{S_i, i = 1, 2\}$ is a sequence of Bernoulli trials. For each state S_i, there is an associated probability $p_i = P(X = S_i)$, and that the probabilities must sum to one. Thus $p_1 = 1 - p_2$. The J verb below generates a sequence of Bernoulli trials for $S_1 = 0$ and $S_2 = 1$: .

```
   load 'stats.ijs' NB. load statistics functions
   rber =: >runif
```

The right argument specifies the number of trials and the left argument is the probability p_1. An on/off flow with a peak rate $r_{\text{peak}} = 1$ and mean rate $\overline{r} = 0.25$ has a "on" probability of $p_{\text{on}} = p_1 = 0.25$. The J expression below generates a sequence of 20 Bernoulli trials, representing the on/off periods of a traffic flow:

```
   0.25 rber 20
0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 1 0 0 1 0
```

Multiple flows can be generated simultaneously. The expression below generates three flows:

```
   0.3 rber 3 20
1 0 0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 1 1
1 0 1 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1
0 1 0 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0
```

6.2 Binomial Distribution

If for an aggregate of n Bernoulli trials, k is the number of occurrences of S_1, then the probability of k out n possible outcomes is given by the Binomial distribution [26]:

$$P(Y = k) = \frac{k!}{k!(n-1)!} p^k (1-p)^{n-k} \tag{6.3}$$

where $p = P(X = S_1)$ and $P(X = S_2) = 1 - p$. The expectation of k is $\text{E}[k] = np$. Generate 20 flows of 10,000 trials for $p = 0.25$:

```
   a1 =: +/ 0.25 rber 20 10000
   mean a1
4.9856
```

As we can see, the mean aggregate rate is close to the expected rate $np = 5$. The J verb Listing 6.1 calculates the Binomial distribution function given in Equation (6.3). The function *bindist* requires the positional parameter functions, therefore we use the *load* command to run the *libs.ijs* script:

```
   load 'libs.ijs'  NB. load pos. param. functions
```

Listing 6.1 *Binomial Distribution*

```
cocurrent < 'BIN'
p =: lhs1
n =: rhs1
k =: i.@>:@n
f1 =: !@n
f2 =: (!@k) * (!@(n-k))
f3 =: f1%f2
f4 =: (p^k) * (1:-p)^(n-k)
f5 =: f3*f4
cocurrent < 'base'
bindist_z_ =: f5_BIN_
```

The J expression below computes the binomial distribution for $p = 0.25$ and $n = 20$:

```
    5 4 $ 0.25 bindist 20
 0.00317121   0.0211414  0.0669478    0.133896
   0.189685    0.202331   0.168609    0.112406
  0.0608867   0.0270608 0.00992228  0.00300675
0.000751688 0.000154192 2.56987e_5   3.4265e_6
 3.56927e_7  2.79942e_8 1.55524e_9 5.45697e_11
```

The shape term `5 4 $` is merely used for brevity. We can calculate the probability distribution of a flow. The verb *fdist* computes the frequency of occurrence of each member of a data set:

```
   fdist_z_ =: [: +/"1 =/
```

We calculate the frequencies of each element in a_1 and divide them by the *tally* `[:#]` to obtain the probabilities:

```
   5 4  $ (i.20) (fdist%[:#]) a1
0.0027  0.023 0.0674 0.1292
0.1842 0.2078 0.1691 0.1084
0.0645 0.0293 0.0093 0.0039
0.0008 0.0001 0.0003      0
     0      0      0      0
```

The Poisson distribution [26] is a limiting form of the Binomial distribution and is often used to model packet arrivals and packets being processed. The probability distribution is given by:

$$P(X = x) = \lambda^x \frac{e^{-\lambda}}{x!} \tag{6.4}$$

Listing 6.2 shows the J verb for computing the Poisson distribution.

Listing 6.2 *Poisson Distribution*

```
cocurrent < 'POIS'
x  =: rhs0
lm =: lhs1
f1 =: lm^x
f2 =: ^@(-@lm)
f3 =: !@x
f4 =: f1*f2%f3
cocurrent < 'base'
poisdist_z_ =: f4_POIS_
```

A rate parameter λ is passed to *poisdist* (instead of a probability p, as with *bindist*), where $\lambda = 4$ (equivalent to $p = 0.25$). Thus:

```
   5 4 $ 4 poisdist i.20
  0.0183156   0.0732626   0.146525   0.195367
   0.195367    0.156293   0.104196  0.0595404
  0.0297702   0.0132312 0.00529248 0.00192454
0.000641512 0.000197388 5.63967e_5 1.50391e_5
 3.75978e_6  8.84654e_7  1.9659e_7 4.13873e_8
```

The graph in Fig 6.1 shows the Binomial and Poisson distributions. It also shows the *normalised* frequency distribution of the arrival process a_1. It can be seen that the three distributions curves are reasonably alike.

6.3 Markov Models

An on/off source can be modeled as a discrete-time Markov chain (see Fig 6.2). The probability $p_{ij} = P(j|i)$ is the probability that the system transitions from state i to state j. If we assign state 1 to the on period and state 0 to the off period, then $\mathbf{T}$ represents the state probability matrix:

$$\mathbf{T} = \begin{bmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{bmatrix} = \begin{bmatrix} 1-a & a \\ b & 1-b \end{bmatrix} \tag{6.5}$$

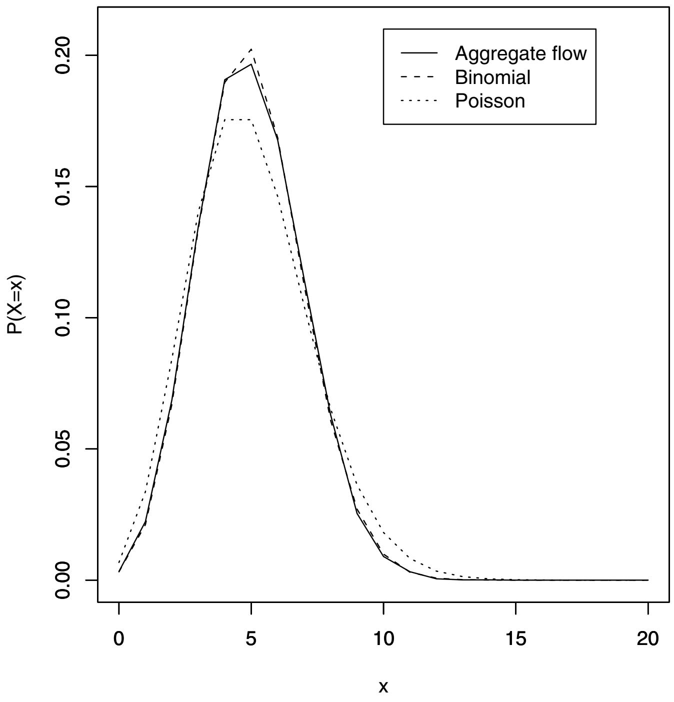

Fig. 6.1. Probability distribution of the aggregate of 20 on/off flows compared to Binomial and Poisson distribution

Note that the transition probabilities with the same originating state must sum to one; that is, $p_{00} + p_{01} = 1$. The probability of the system being in state i at time k is $\pi_i^{(k)}$. Thus:

$$\pi = \pi \mathbf{T} \tag{6.6}$$

where $\lim_{k \to \infty} \pi = \pi^{(k)}$. The terms $\pi_0^{(k)}$ and $\pi_1^{(k)}$ are given by the expressions:

$$\begin{aligned} \pi_0^{(k)} &= \pi_0^{(k-1)} p_{00} + \pi_1^{(k-1)} p_{10} \\ \pi_1^{(k)} &= \pi_0^{(k-1)} p_{01} + \pi_1^{(k-1)} p_{11} \end{aligned} \tag{6.7}$$

Solving for π_0 and π_1 gives:

$$\begin{aligned} \pi_0 &= \frac{p_{10}}{p_{01} + p_{10}} \\ \pi_1 &= \frac{p_{01}}{p_{10} + p_{01}} \end{aligned} \tag{6.8}$$

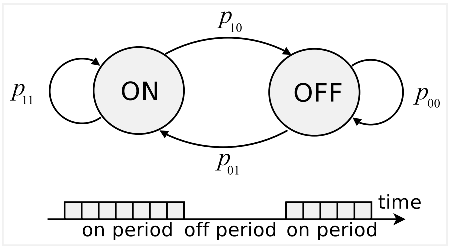

Fig. 6.2. Discrete time on/off source model

For the purpose of illustration, set the values of the transition matrix $\mathbf{T}$ to:

$$\mathbf{T} = \begin{bmatrix} 0.8 & 0.2 \\ 0.6 & 0.4 \end{bmatrix} \tag{6.9}$$

Substituting the values of $p_{ij}, i, j = 1, 2$ into the expressions in Equation (6.8) results in $\pi_0 = 0.75$ and $\pi_1 = 0.25$. Assign values to the probability matrix $\mathbf{T}$:

```
   ]T =: 0.8 0.2 ,: 0.6 0.4
0.8 0.2
0.6 0.4
```

Define the matrix multiplication verb:

```
   mm_z_ =: +/@:*
```

Choose (arbitrary) initial values for $\pi^{(0)}$, ensuring that $\pi_0^{(0)} + \pi_1^{(0)} = 1$, then use the matrix multiply verb and the power conjunction to find $\pi^{(n)}$:

```
  (col i.10) ; T mm^:(i.10) 0.5 0.5
+-+-----------------+
|0|     0.5      0.5|
|1|     0.7      0.3|
|2|    0.74     0.26|
|3|   0.748    0.252|
|4|  0.7496   0.2504|
|5| 0.74992  0.25008|
|6|0.749984 0.250016|
|7|0.749997 0.250003|
|8|0.749999 0.250001|
|9|    0.75     0.25|
+-+-----------------+
```

By the ninth iteration, the result converges to π_0 and π_1. The expectation is given by:

$$\mathrm{E}[X] = \sum_{i=0}^{1} S_i \pi_i \tag{6.10}$$

Thus, for $S_i = i$, the expectation can be calculated by the J expression:

```
   (T mm^:(10) 0.5 0.5) ip 0 1
0.25
```

6.4 Effective Bandwidth

For a link with capacity $c = 5$ and arrivals a_1, we can use the Lindley equation to compute the backlog for an infinite buffer:

```
   load 'netcalc.ijs' NB. load lindley function
   hd1 =: 'mean';'max'
   q1 =: (5;a1) lindley^:(10000) 0
   hd1,: (mean;max) q1
+-------+---+
|mean   |max|
+-------+---+
|61.54  |163|
+-------+---+
```

The graph in Fig 6.3 shows how the queue evolves over time. It shows that setting the capacity close to the mean rate of the aggregate flow results in high levels of backlog. The link is operating at the very limit of its capacity, and the autocorrelations in Fig 6.4 diminish linearly, indicating that the backlog is nonstationary. The autocorrelations are computed using:

```
   m =: col i.100
   rq =: m autocor"1 q1
```

Effective bandwidth [31] is a method of determining the amount of network resources that a flow needs in order to meet a particular QoS. Consider a link with a capacity c for which we can set the channel access control (CAC) limit n_{cac}; that is, the maximum number of flows that will be admitted to the link. Setting the CAC limit to $n_{\text{cac}} = c/\overline{r}$ ensures maximum statistical multiplexing gains. Clearly, $n_{\text{cac}} = c/\overline{r}$ should not exceed the capacity of the link, as the link will become unstable, resulting in buffer queues growing without bounds. Nevertheless, when the link operates at its CAC limit, there will be periods of heavy backlog in the system resulting in long delays. In short the link can only support a *best effort* quality of service.

The delay can be bounded by $1/c$, if we set $n_{\text{cac}} = c/r_{\text{peak}}$; however, this can lead to significant underutilisation of the link, particularly if the traffic is bursty Thus we have a trade-off between link efficiency and QoS. Clearly, we need to set $c/r_{\text{peak}} \leq$

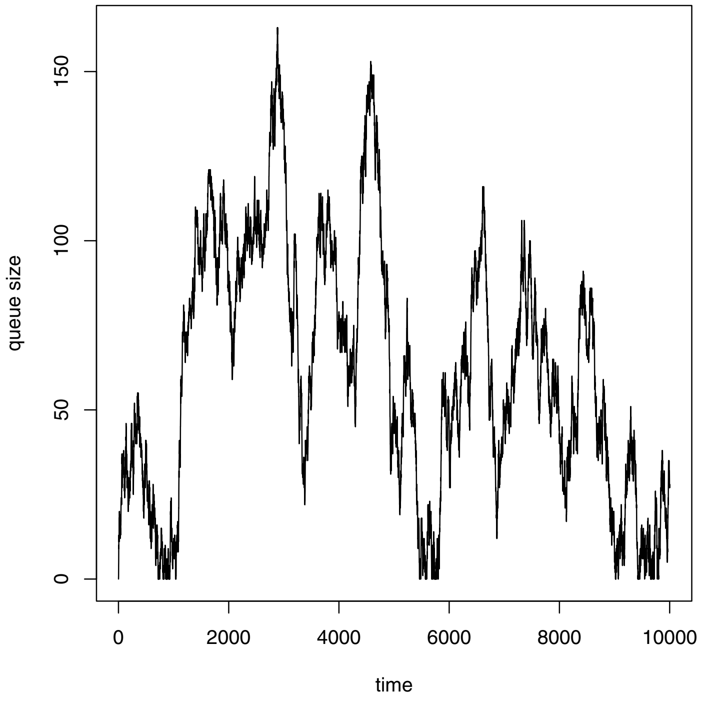

Fig. 6.3. Backlog for 20 on/off sources

$n_{cac} \leq c/\overline{r}$. The aim is to admit as many connections as possible without exceeding the QoS bounds. For a delay bound D, we define some QoS parameter γ, such that the delay exceeds D, with probability $e^{-\gamma}$. The shape parameter θ is given by:

$$\theta = \frac{\gamma}{D \cdot c} \tag{6.11}$$

The random variable X_j is the load admitted to the link by flow j in some unit time interval. The effective bandwidth $\Lambda(\theta)$ is given by:

$$\Lambda(\theta) = \theta^{-1} \log \mathrm{E} e^{\theta X_j} \tag{6.12}$$

The effective bandwidth function for deriving probabilistic delay bounds is given in Listing 6.3.

Listing 6.3 *Effective Bandwidth*

```
cocurrent <'EB'
s =: rhs0
```

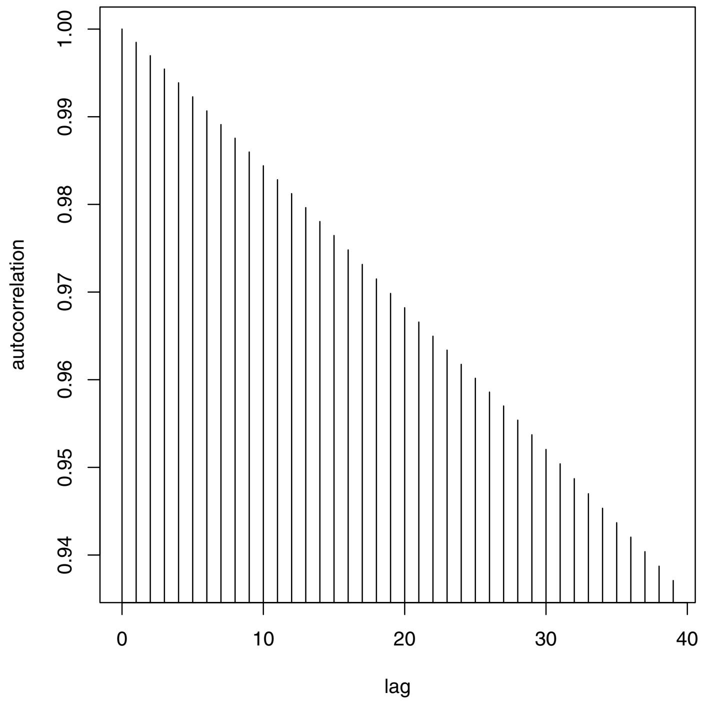

Fig. 6.4. Autocorrelation coefficients of queue length

```
x =: >@lhs1
p =: >@lhs2
f1 =: ^@(s */ x)
f2 =: f1 (*"1) p
f3 =: +/"1 @f2
f4 =: (^.@f3) % s
cocurrent <'base'
eb_z_ =: f4_EB_
```

Generate a Poisson random variable with a mean of two and peak of five:

```
   ]rv =: 2 (];poisdist) i.5
+---------+---------------------------------------+
|0 1 2 3 4|0.13533 0.27067 0.27067 0.18044 0.090223|
+---------+---------------------------------------+
```

For a link with capacity $c = 5$ and a delay bound $D = 0.25$, the effective bandwidth over a range of values of $\theta = \gamma/Dc$ is given by the J expression:

```
   rv eb (1 5 10 20)%0.25*5
```

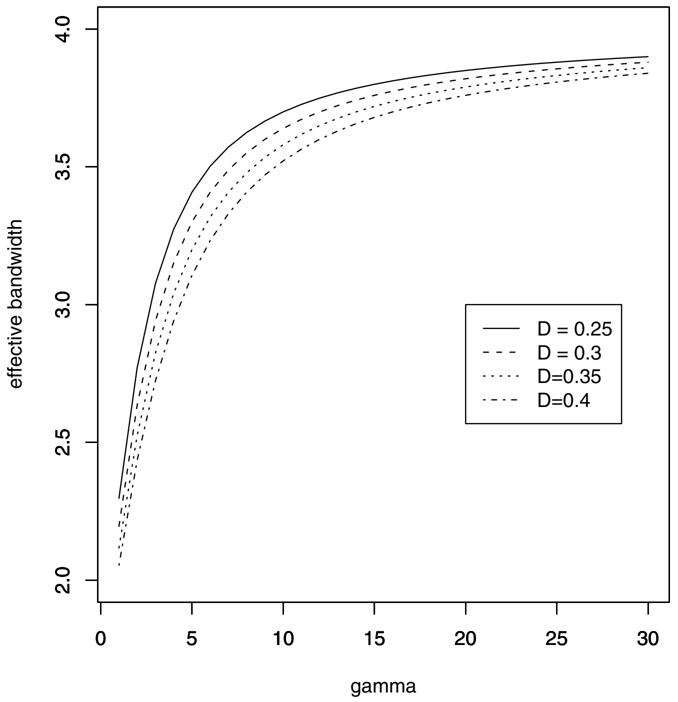

Fig. 6.5. Effective bandwidth of a Poisson process

```
2.29689 3.40787 3.6994 3.84966
```

The graph in Fig 6.5 shows the effective bandwidth for the Poisson process for various delay bounds.

Here, we demonstrate how to use effective bandwidth for capacity planning. Suppose that we have a link of capacity $c = 5$ and we wish to determine the maximum number of flows that can be admitted to the link for a given QoS. For the purpose of illustration, we specify a delay bound $D = 0.4$. For $\gamma = 2$, the probablity that the delay does not exceed 0.4 with a probability greater than $1 - e^{-2}$. The computation in J is:

```
   (>:@-@^) _2
0.864665
```

In other words, the link should satisfy the delay bound approximately 86 percent of time, provided the flows honour their traffic contract. For the purpose of illustration, we set the mean rate of the flow to $\overline{r} = 0.25$ and the peak rate $r_{\text{peak}} = 1$. In the example above (Section 6.2), we saw the effects of running 20 such flows (over a link of capacity $c = 5$). The delay is the service time $(1/c)$ plus the the time spent waiting in the queue. We can calculate the percentage of time for which the delay bound is exceeded:

```
   delay =: 0.2 * q1 + 1
   mean 0.4 <: delay
0.986302
```

The QoS is maintained less than 2 percent of the time if we provision the link for the mean rate of the aggregate flow. If we provision the link for the peak rate, then only five flows can be admitted and each will receive a delay of 0.2, which is well within our delay bound $D = 0.4$. If, however, we capacity plan for the effective bandwidth $\Lambda(\theta)$, then we should be able to admit more flows to the link and still maintain the required QoS. Given that $s = \gamma/Dc$, the effective bandwidth of a flow is:

```
   (0 1;0.75 0.25) eb (2%0.4*5)
0.357374
```

Dividing the link capacity c by the effective bandwidth $\Lambda(\theta)$ gives the number of flows:

```
   ceil 5 % 0.357374
13
```

The link should provide QoS for up to 13 flows. Generate 13 on/off flows and aggregate them:

```
   a2 =: +/ 0.25 rber 13 10000
```

Use the Lindley equation to calculate the queue size:

```
   q2 =: (5;a2) lindley^:(10000) 0
   hd1,: (mean;max) q2
+--------+---+
|mean    |max|
+--------+---+
|0.162784|6  |
+--------+---+
```

Then, we calculate the proportion of times the delay bound ($D = 0.4$) is exceeded:

```
   delay2 =: (1+q2)%5
   mean delay2 ge 0.4
0.10289
```

The delay is within the required bound approximately 90 percent of the time, thus QoS is maintained. Let us repeat this experiment for 14 flows:

```
   a3 =: +/ 0.25 rber 14 10000
   q3 =: (5;a3) lindley^:(10000) 0
   d3 =: (1+q3)%5
   mean (0.4 <: d3)
0.156284
```

Recall that $e^{-2} \approx 0.135$; thus the link cannot quite maintain the desired quality of service with 14 flows.

6.5 Discrete On/Off Source Models

In this section, we develop J verbs for generating simulations of traffic flows based upon on/off models. The superposition of flows with heavy-tailed on or off periods (or on *and* off periods) results in long-range dependence and self-similarity (lrd-ss) [55]. Two models are implemented, one with geometrically distributed on and off times, and one with Pareto distributed on times and geometrically distributed off times. The former model, Geo[on]-Geo[off], produces a Markovian traffic flow. The latter model Geo[on]-Par[off], when multiple flows are aggregated, produces a traffic process that is lrd-ss. The verb definition in Listing 6.4 generates Geo[on]/Geo[off] traffic flows.

Listing 6.4 *Geo[on]-Geo[off]*

```
cocurrent < 'OOGEOGEO'
p01 =: lhs1
p10 =: lhs2
n  =: rhs1
f1 =: %@p01 rgeo n
f2 =: %@p10 rgeo n
f3 =: f1,.f2
f4 =: f3 <@# 1:,0:
cocurrent < 'base'
oosrd_z_ =: ;@f4_OOGEOGEO_
```

The verb *oosrd* takes two left arguments, which are the respective on and off transition probabilities. The expression below generates 20 Geo[on]-Geo[off] flows:

```
   x4 =: 1r2 1r6 oosrd"1  (20 1 $ 2000)
```

We aggregate the flows and take a sample from the middle:

```
   a4 =:  (1001 to 3000) { +/ x4
```

We then find the mean rate, peak rate and standard deviation of the aggregate flow:

```
   hd2 =: hd1,<'std'
   (mean; max; std) a4
+------+---+-------+
|mean  |max|std    |
+------+---+-------+
|5.0645|12 |1.94582|
+------+---+-------+
```

We can calculate the frequency distribution of the aggregate flow:

```
   (i.11) ([,: fdist) a4
0  1   2   3   4   5   6   7   8  9 10
5 42 108 298 353 397 345 242 116 63 21
```

The resultant flow should be short-range dependent (srd). Use the variance-time plot to confirm this:

```
   m =:  col 10 to 100
   vx4 =: m varm"1 a4
```

We determine the slope at which the variance diminishes (with m) on the log-log scale:

```
   hd3 =: 'intercept';'slope'
   hd3,: ;/ (log10 vx4) %. (1,.log10 m)
+---------+--------+
|intercept|slope   |
+---------+--------+
|0.874321 |_1.00042|
+---------+--------+
```

The variance diminishes at a rate of approximately -1, yielding a Hurst parameter $H \approx 0.5$, confirming that the resultant aggregate traffic process is srd:

```
   hurst 1.00042
0.49979
```

In Listing 6.5, we define the verb for generating traffic flows with geometrically distributed on periods and Pareto distributed off periods.

Listing 6.5 *Geo[on]-Par[off]*

```
cocurrent < 'OOPARGEO'
p01 =: lhs1
alpha =: lhs2
n  =: rhs1
f1 =: %@p01 rgeo n
f2 =: (alpha,1:) (ceil@rpar) n
f3 =: f1,.f2
f4 =: f3 <@# 1:,0:
cocurrent < 'base'
oolrd_z_ =: ;@f4_OOPARGEO_
```

Generate 20 Geo[on]-Par[off] flows for $\alpha = 1.2$ and $r_1 = 8$ and then aggregate them:

```
   x5 =: 1.2 2 oolrd"1 (20 1 $ 2000)
   a5 =:  (1001 to 3000) { +/ x5
   hd2,: (mean; max; std) a5
+-----+---+-------+
|mean |max|std    |
+-----+---+-------+
|5.565|13 |2.29726|
+-----+---+-------+
```

From the variance-time plot, we can see that the traffic process with heavy-tailed off times yields a value of $H \approx 0.93$, which indicates a high degree of lrd-ss:

```
   vx5 =: m varm"1 a5
   hd3,: ;/(log10 vx5) %. 1,.log10 m
+---------+---------+
|intercept|slope    |
+---------+---------+
|0.55286  |_0.143089|
+---------+---------+
   hurst 0.143089
0.928455
```

We can compare the queue dynamics of the two traffic sources by calculating the backlog. We assume that the queue is sufficiently large so that there are no packet drops. We use the Lindley equation thus:

```
   q4 =: (8;a4) lindley^:(2000) 0
   q5 =: (8;a5) lindley^:(2000) 0
```

It can be seen that the backlog for the lrd-ss traffic source is far larger (and more varied) than for srd traffic:

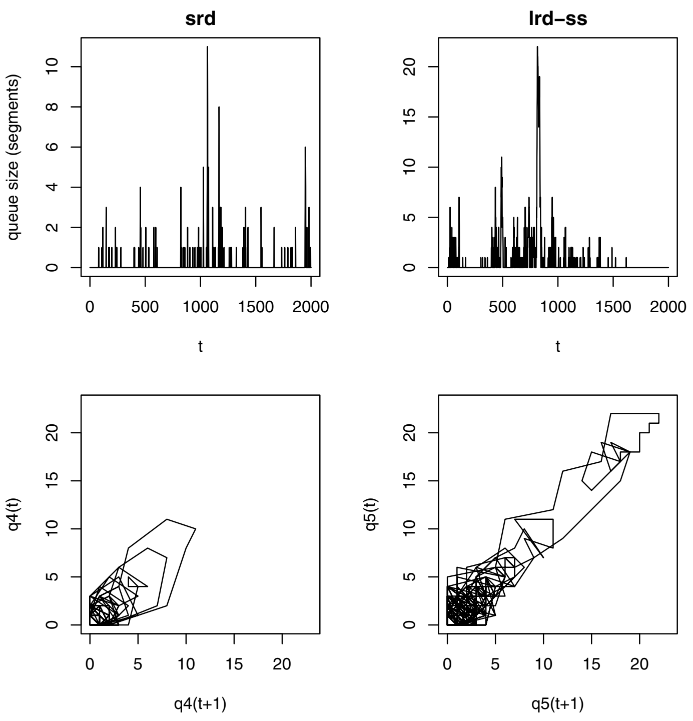

Fig. 6.6. The queue dynamics of a srd and lrd-ss traffic processes

```
   hd4 =: 'q4';'q5'
   hd2, (hd4,. (mean; max ; std)"1 q4,:q5)
+--+--------+---+--------+
|  |mean    |max|std     |
+--+--------+---+--------+
|q4|0.132434|11 |0.694948|
+--+--------+---+--------+
|q5|0.681659|22 |2.38015 |
+--+--------+---+--------+
```

The graphs in Fig 6.6 (top) show the backlog for the srd (q_4) and lrd-ss (q_5) traffic processes respectively. Figure 6.6 also shows the backlog as two dimensional phase diagrams by plotting $q_4(t)$ against $q_4(t+1)$ (bottom left) and $q_5(t)$ against $q_5(t+1)$ (bottom right). The J verb below, when supplied with a list of values y, returns two lists $y(t)$ and $y(t+1)$:

```
   (phasediag_z_ =: }: ,: }.) i.10
0 1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
```

The phase diagrams in Fig 6.6 were generated from the expressions:

```
   phase4 =: phasediag q4
   phase5 =: phasediag q5
```

6.6 Summary

On/off models can be used to simulate network traffic. They are based upon the abstraction that during an on period a source transmits at some constant rate and is idle during its off period. The superposition of multiple on/off flows with heavy-tailed on and/or off periods produces lrd-ss traffic.

7

Chaotic Maps

In this chapter, we introduce chaotic maps as a method of simulating on/off network traffic flows. Chaotic systems are systems that are sensitive to initial conditions (SIC). That is, even a small perturbation in the starting conditions can significantly affect the long-term behaviour of the system. Chaotic behaviour is modeled using low-order, nonlinear dynamical maps. These maps are *low-order* because they are described by only a few parameters and variables. The parameters of the maps are fixed, but variables vary over time, giving the maps their dynamical properties.

The evolution of the system over time is described as a *trajectory* in the (variable) state space. Short-term changes in state can be found deterministically by some transformation: $x(n+1) = f(x(n))$. Long-term behaviour is derived from successive iterations of the map. The term, f^N represents the N^{th} iteration of the map, thus $x(n+1) = f^n(x(0))$.

The *logistic map* is introduced in order to demonstrate chaotic behaviour. We then introduce two other maps, the Bernoulli shift map and the double intermittency map, to simulate on/off traffic flows. The Bernoulli shift map generates sequences that are short-range dependent (srd), while sequences from the double intermittency map are long-range dependent and self-similar (lrd-ss).

7.1 Analysing Chaotic Behaviour

Consider the logistic map in the expression:

$$f(x) = 4x(1-x) \tag{7.1}$$

For values of x in the interval $[0,1]$, the map is an inverted parabola with a maxima at $x = 0.5$. The expression in Equation (7.1) can be implemented by the J verb below:

```
   f =: 4&*@(* >:@-)
```

Which yields the expected inverted parabola:

```
   load 'libs.ijs'
   ]x =: int01 i.11
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
   f x
0 0.36 0.64 0.84 0.96 1 0.96 0.84 0.64 0.36 0
```

In its iterative form, the logistic map is:

$$f(x(n+1)) = 4x(n)(1-x(n)) \tag{7.2}$$

For the purpose of illustration, we select an initial starting value of $x(0) = 0.1$ and iterate the map by applying the power conjunction `^:` to the function f:

```
   10 5 $ f^:(i.50) 0.1
       0.1       0.36     0.9216    0.289014    0.821939
  0.585421   0.970813   0.113339    0.401974    0.961563
  0.147837   0.503924   0.999938 0.000246305 0.000984976
0.00393603  0.0156821  0.0617448     0.23173    0.712124
  0.820014   0.590364   0.967337    0.126384    0.441645
  0.986379   0.053742   0.203415     0.64815    0.912207
  0.320342   0.870893   0.449754    0.989902    0.039986
  0.153548   0.519885   0.998418  0.00631654   0.0251066
 0.0979049   0.353278   0.913891    0.314778    0.862771
  0.473588    0.99721  0.0111304   0.0440261    0.168351
```

The shape term `10 5 $` is used for brevity. Following the sequence $x(n)$ from the top left to bottom right reveals no obvious pattern. This is illustrated in graphic form in Fig 7.1, which shows the trajectory of the map. The graph in Fig 7.2 shows the plot of successive points; that is, $x(n+1)$ against $x(n)$. It shows how the points are *attracted* to the parabola. The expresssion below is a more generalised form of Equation (7.1):

$$g(x) = 4ax(1-x) \tag{7.3}$$

The expression in Equation (7.3) describes a family of parabolic curves, where the magnitude of the maxima is determined by $0 \leq a \leq 1$ and is implemented in J:

```
   g =: lhs1 * [: f ]
```

The parameter a is passed as the left argument, where, for the purpose of this example, $a = 0.8$:

```
   0.8 g x
0 0.288 0.512 0.672 0.768 0.8 0.768 0.672 0.512 0.288 0
```

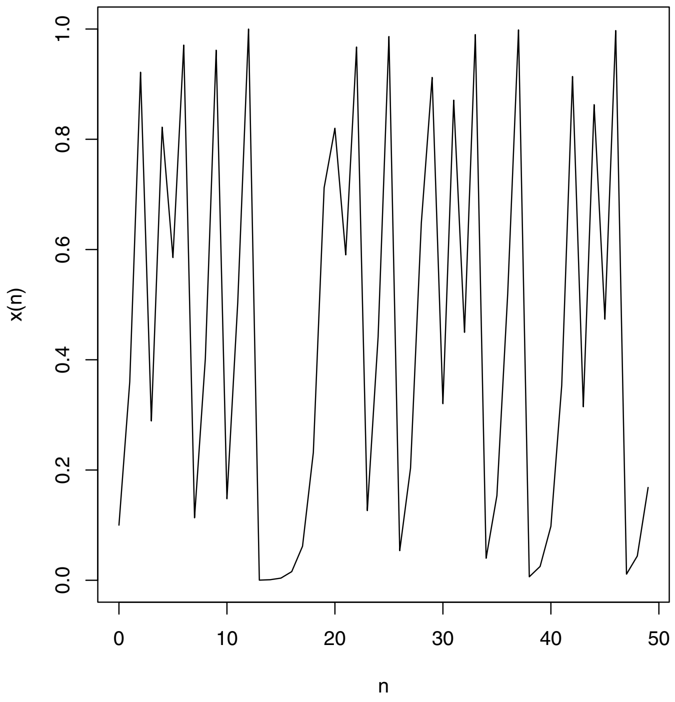

Fig. 7.1. The trajectory of the logistic map in Equation (7.2)

The graph in Fig 7.3 shows the parabolic curves for various values of a. Note that $g(x) = f(x)$, when $a = 1$:

```
   hd1 =: 'g',:'f'
   hd1; 1 (g ,: [: f ]) x
+-+-------------------------------------------+
|g|0 0.36 0.64 0.84 0.96 1 0.96 0.84 0.64 0.36 0|
|f|0 0.36 0.64 0.84 0.96 1 0.96 0.84 0.64 0.36 0|
+-+-------------------------------------------+
```

The value of parameter a does more that affect the height of the parabola; it also determines the behaviour of the trajectories when the map is applied iteratively. Sucessive iterations of the map for $a = 0.4$ shows that the trajectory increases from its initial starting value $x(0) = 0.1$ to a stable state:

```
   10 5 $ 0.4 g^:(i.50) 0.1
     0.1    0.144 0.197222 0.253321 0.302639
0.337678 0.357843 0.367666  0.37198 0.373778
0.374509 0.374803 0.374921 0.374968 0.374987
```

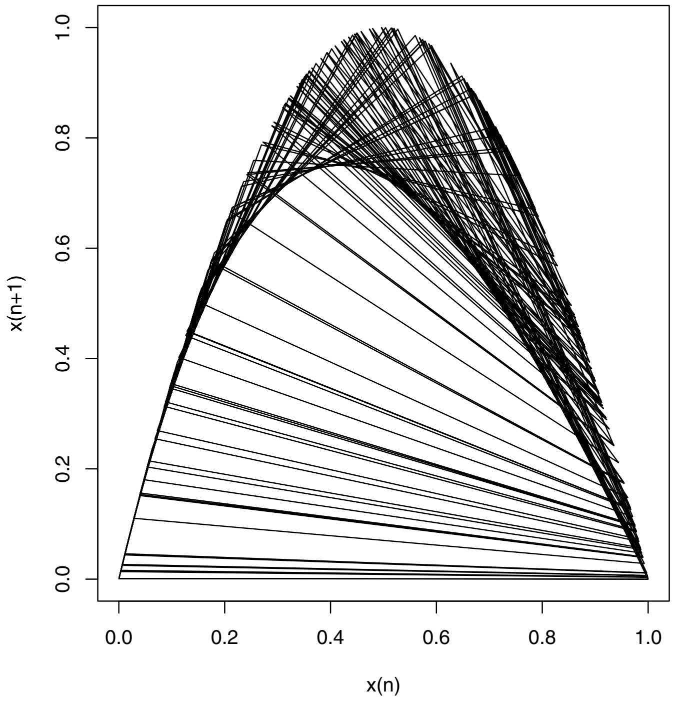

Fig. 7.2. Plot of $x(n+1)$ against $x(n)$ for the logistic map in Equation (7.2)

```
0.374995 0.374998 0.374999    0.375    0.375
   0.375    0.375    0.375    0.375    0.375
   0.375    0.375    0.375    0.375    0.375
   0.375    0.375    0.375    0.375    0.375
   0.375    0.375    0.375    0.375    0.375
   0.375    0.375    0.375    0.375    0.375
   0.375    0.375    0.375    0.375    0.375
   0.375    0.375    0.375    0.375    0.375
```

Similarly, a stable state is achieved for $a = 0.8$, although the (stable) trajectory cycles between two values:

```
   10 5 $ 0.8 g^:(i.50) 0.1
     0.1    0.288 0.656179 0.721946 0.642368
 0.73514 0.623069 0.751533  0.59754 0.769555
0.567488 0.785425 0.539304 0.795057 0.521413
0.798533  0.51481 0.799298 0.513346  0.79943
0.513093 0.799451 0.513052 0.799455 0.513046
0.799455 0.513045 0.799455 0.513045 0.799455
```

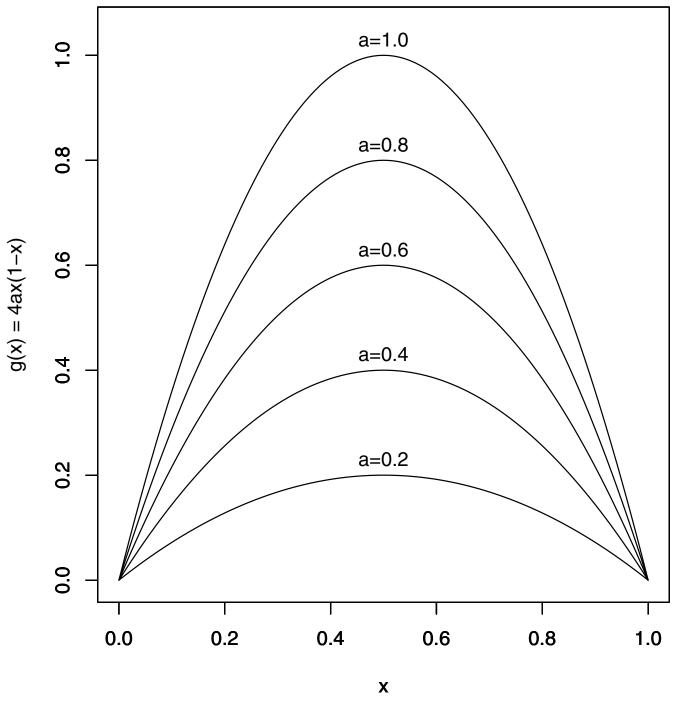

Fig. 7.3. Parabolic curves for the logistic map in Equation (7.3) for various values of a

```
0.513045 0.799455 0.513045 0.799455 0.513045
0.799455 0.513045 0.799455 0.513045 0.799455
0.513045 0.799455 0.513045 0.799455 0.513045
0.799455 0.513045 0.799455 0.513045 0.799455
```

The graph in Fig 7.3 shows that the trajectories for $a = 0.4$, 0.7, 0.8 and 0.88 converge to *periodic* stable states. For $a = 0.4$ and 0.7, the trajectories are period one, and for $a = 0.8$ and $a = 0.88$ the periods are 2 and 4, respectively. However, as we have already seen in Fig 7.1, when $a = 1$, the trajectory does not converge to any stable (periodic) state. This behaviour is common in dynamical systems, whereby the *stable* state undergoes a series of *bifurcations* in response to a change in conditions (in this case, the value of a). As the conditions continue to change, the system lapses into chaos.

Chaotic systems are sensitive to initial conditions (SIC). This is known as the butterfly effect [52], whereby small perturbations in the initial conditions can lead to widely varying results in the long term. In order to examine this, consider the trajectory for $a = 0.8$. The J expression below returns the results for the 49^{th} and 50^{th} iteration of the logistic map for $x(0) = 0.1, 0.4$ and 0.7:

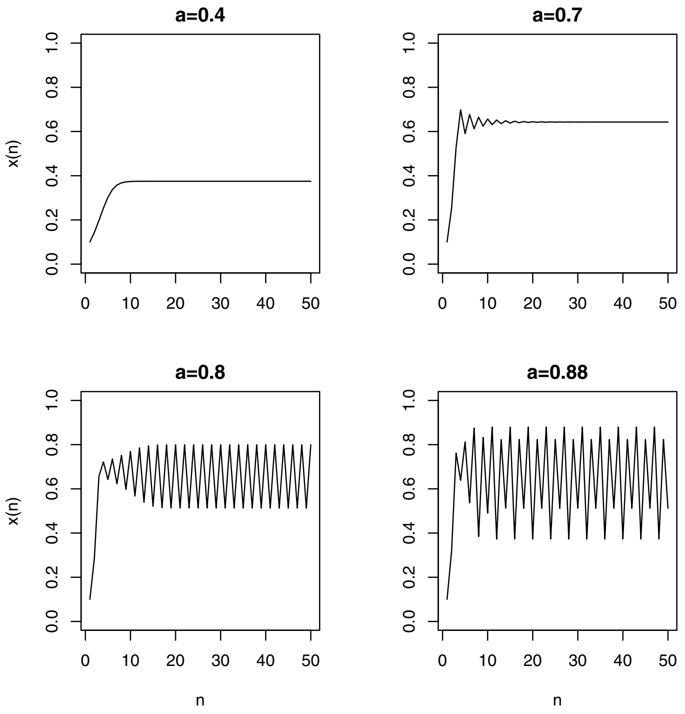

Fig. 7.4. The trajectories for $a = 0.4$, 0.7, 0.8, 0.88

```
   hd2 =: '49',:'50'
   hd2; 0.8 g^:(49,50) 0.1 0.4 0.7
+--+-------------------------+
|49|0.799455 0.799455 0.513045|
|50|0.513045 0.513045 0.799455|
+--+-------------------------+
```

We can see that, for each initial condition, the system converges to the same two-period state, albeit, for $x(0) = 0.7$, the trajectory is half a cycle out of phase with the other two. This is illustrated in the graph in Fig 7.5 which shows the trajectories up to $N = 50$ for each starting value $x(0) = 0.1$, 0.4 and 0.7.

Now, we compare two trajectories for $a = 1$ with starting conditions $x(0)$ and $x(0)+\varepsilon$, where $\varepsilon = 10^{-6}$ (in this case) is the perturbation. It is reasonable to expect the evolution of a system from a perturbed starting condition as relatively small as this to remain close to the original (nonperturbed) throughout each iteration. Indeed, this appears to be the case if we run the logistic map (for $a = 1$) up to $N = 10$ iterations:

```
   (col i.11); 1 g^:(i.11) 0.1 0.100001
```

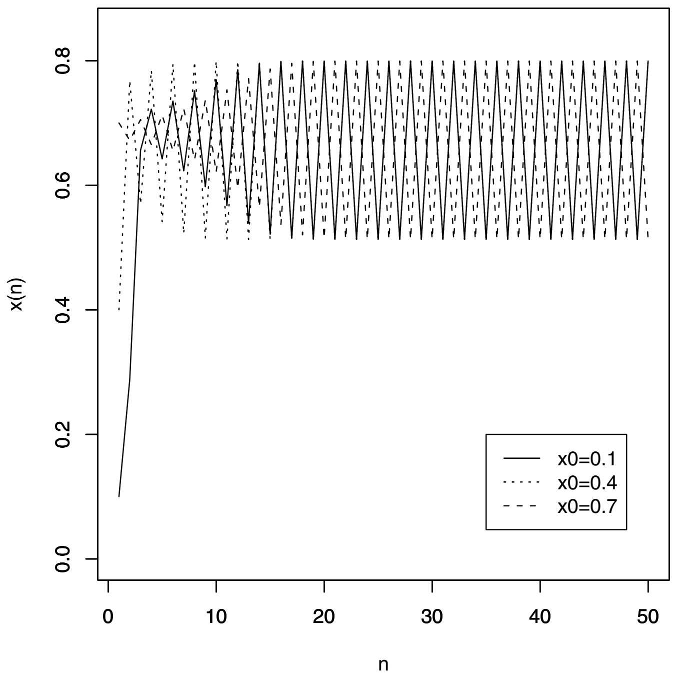

Fig. 7.5. Three trajectories for the initial starting conditions of $x(0) = 0.1$, 0.4 and 0.7

```
+--+-----------------+
| 0|     0.1 0.100001|
| 1|    0.36 0.360003|
| 2|  0.9216 0.921604|
| 3|0.289014 0.289002|
| 4|0.821939 0.821919|
| 5|0.585421 0.585473|
| 6|0.970813 0.970777|
| 7|0.113339 0.113475|
| 8|0.401974 0.402392|
| 9|0.961563 0.961891|
|10|0.147837 0.146627|
+--+-----------------+
```

It can be seen that, as the map evolves through the iterations of $n = 0, 1, \ldots 10$, the two trajectories stay close together. However, if we run the map for a further 10 iterations ($n = 11, \ldots 20$), the two trajectories begin to diverge:

```
    (col 11 to 20); 1 g^:(11 to 20) 0.1 0.100001
```

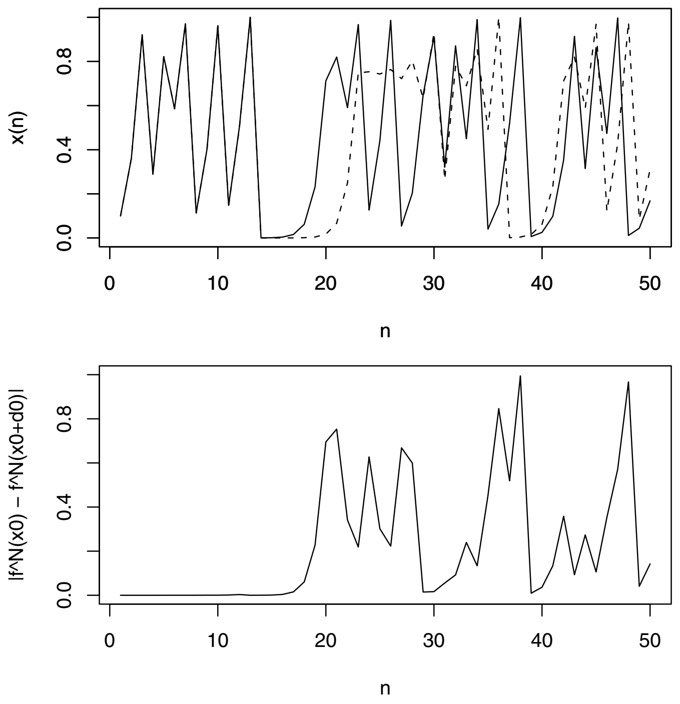

Fig. 7.6. Two trajectories from the logistic map (top). The difference between the trajectories (bottom)

```
+--+----------------------+
|11|   0.503924    0.50051|
|12|   0.999938   0.999999|
|13|0.000246305  4.16748e_6|
|14|0.000984976  1.66698e_5|
|15| 0.00393603  6.66783e_5|
|16|  0.0156821 0.000266695|
|17|  0.0617448   0.0010665|
|18|    0.23173  0.00426144|
|19|   0.712124   0.0169731|
|20|   0.820014   0.0667401|
+--+----------------------+
```

This is illustrated graphically in Fig 7.6 (top) for the first $N = 50$ iterations. The divergence of the two trajectories becomes apparent at about iteration thirteen. The bottom graph in Fig 7.6 shows the magnitude of the difference between the two trajectories according to the expression:

$$E_N = |g^N(x(0)) - g^N(x(0)+\varepsilon)| \tag{7.4}$$

This leads us to the Lyapunov exponent [17] which serves as a metric for assessing the rate at which a trajectory, with a perturbed starting value, diverges. The Lyapunov exponent is given by the expression:

$$\lambda(x(0)) = \frac{1}{N}\text{ln}E(N)/\varepsilon \tag{7.5}$$

The Lyapunov exponent $\lambda(x(0))$ for the logistic map ($x(0) = 0.1$, $\varepsilon = 10^{-6}$ and $N = 20$) can be calculated thus:

```
   E1 =: | -/ 1 g^:(20) 0.1 0.100001
   20 %~ ln E1%1e_6
0.676609
```

Systems like the logistic map are highly sensitive to small errors in the initial conditions. Such systems are chaotic and characterised by a Lyapunov exponent $\lambda(x(0)) > 0$. Nonchaotic systems have Lyapunov exponents $\lambda(x(0)) \leq 0$. Thus for $a = 0.8$:

```
   E2 =: | -/ 0.8 g^:(20) 0.1 0.100001
   20 %~ ln E2%1e_6
_0.18071
```

The J verb in Listing 7.1 is an explicit verb definition for calculating the difference between two trajectories for the logistic map in Equation (7.3).

Listing 7.1 *Difference Between Two Trajectories*

```
diff =: 4 : 0
'v0 N a'  =. x. [ 'x0' =. rhs1 y.
'xn1 xn2' =. a g^:(N) x0, (x0+v0)
xn1, | xn1-xn2
)
```

The verb *diff* takes three arguments which are passed on the left-hand side:

- ε (v_0) the value of the perturbation
- N the number of iterations
- a the parameter to the map

The initial value $x(0)$ is passed on the right-hand side. For example:

```
   1e_6 20 0.95 diff 0.1
0.575468 0.00610046
```

The function returns two values. The second value is the actual (absolute) difference between the two trajectories at iteration $N = 20$ (in this case). The first value is the value of the last iteration of the map. The reason for returning this value is so that *diff* can be run iteratively, using this value as the initial condition for sucessive iterations; thus:

```
   1e_6 50 1 diff^:(i.5) 0.1
      0.1        0
 0.560037 0.296104
 0.372447 0.129492
 0.787255  0.75693
0.0873697 0.801794
```

The *lyap* script for calculating the Lyapunov exponent is given in Listing 7.2.

Listing 7.2 *Script for Calculating the Lyapunov Exponent*

```
lyap =: 4 : 0
'v0 N a R'  =. x. [ 'x0' =. y.
df =: rhs2"1 ((v0,N,a) diff^:(1 to R) x0)
(ln df%v0)%N
)
```

The verb *lyap* calls *diff* (iterated R times). Running the *lyap* script, and taking the mean, gives the following results for various values of a:

```
   mean 1e_6 20 0.4 100 lyap 0.1  NB. a=0.4
_0.913705
   mean 1e_6 20 0.5 100 lyap 0.1  NB. a=0.5
__
   mean 1e_6 20 0.8 100 lyap 0.1  NB. a=0.8
_0.908953
   mean 1e_6 20 0.9 100 lyap 0.1  NB. a=0.9
0.183405
   mean 1e_6 20 1 100 lyap 0.1    NB. a=1
0.64713
```

The graph in Fig 7.7 shows the Lyaponov exponent over a range of values of a. Note that for $a = 0.5$, the result is negative infinity (denoted by a double underscore: __). This is a valid result and it is fortunate that J can deal with infinite values.

7.2 Chaotic Maps for Traffic Sources

In this section, we introduce chaotic maps for modeling on/off traffic sources. The state of the map determines whether or not the source is in a transmitting state (on) or

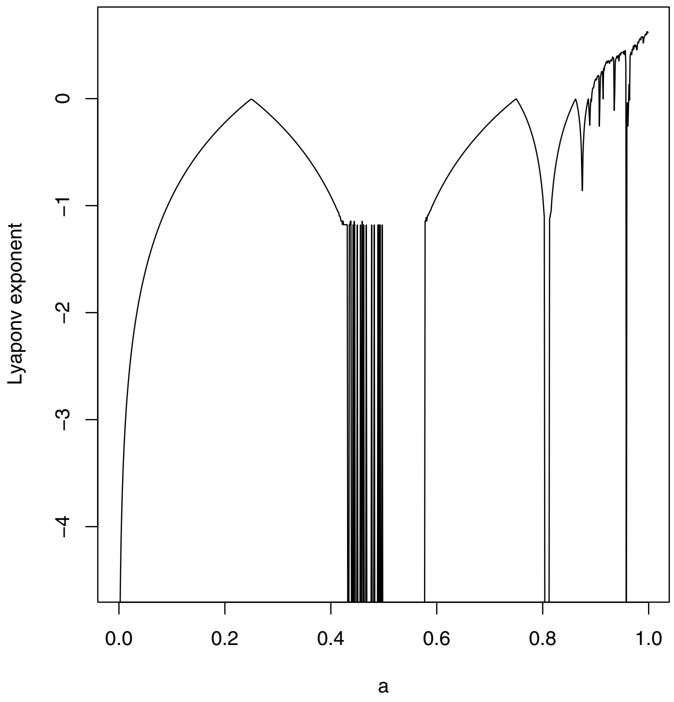

Fig. 7.7. Lyaponov exponent for the logistic map in Equation (7.3) for a range of a and $N = 20$

idle state (off). In this section we introduce two maps for this purpose: the Bernoulli shift and the double intermittency map [17]. Both maps (under the right conditions) exhibit chaotic behaviour, although the statistical properties of their respective trajectories are significantly different. The double intermittency map can (for certain parameters) produce lrd-ss sequences, while the Bernoulli shift is srd.

Both maps are one-dimensional, where $x(n)$ evolves in time according to the transformation:

$$x(n+1) = \begin{cases} f_1(x(n)) \text{ if } & 0 < x(n) \leq d \\ f_2(x(n)) \text{ if } & d < x(n) < 1 \end{cases} \tag{7.6}$$

In order to model an on/off traffic source model, an associated state variable $y(n)$ determines if the system is in an active state $y(n) = 1$ or an idle state $y(n) = 0$. The state of the system is governed by the magnitude of $x(n)$. If $x(n)$ exceeds threshold d, then the system transmits at a peak rate; otherwise it is idle:

$$y(n) = \begin{cases} 0 & \text{if} \quad 0 < x(n) \leq d \\ 1 & \text{if} \quad d < x(n) < 1 \end{cases} \tag{7.7}$$

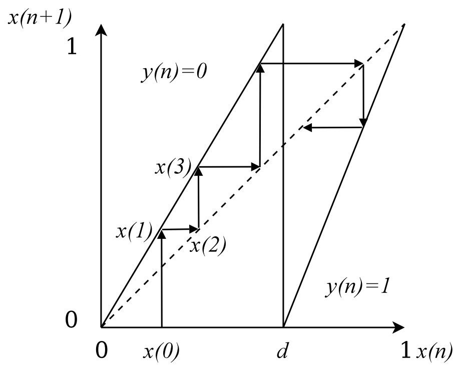

Fig. 7.8. Bernoulli shift map

7.2.1 Bernoulli Shift

Bernoulli Shift map (illustrated in Figure 7.8) consists of two linear segments and takes one parameter d. The evolution of the state variable $x(n)$ is given by the transformation:

$$x(n+1) = \begin{cases} \dfrac{x(n)}{d} & \text{if} \quad 0 < x(n) \leq d \\ \dfrac{x(n)-(d)}{1-d} & \text{if} \quad d < x(n) < 1 \end{cases} \tag{7.8}$$

For Bernoulli map verb, the parameter d is passed on the left of the verb and the initial starting condition $x(0)$ is passed on the right:

```
cocurrent < 'BMAP'
d =: lhs1
x =: rhs0
```

The verbs for the two linear segments are given by:

```
f1 =: x%d
f2 =: (x-d)%(1:-d)
```

The transformation in Equation (7.8) can be implemented thus:

```
xnext =: f1 ` f2 @. (d<x)
```

Finally, we define the verb *bshift* as a reference to *xnext*:

```
cocurrent < 'base'
bshift_z_ =: xnext_BMAP_
```

For $d = 0.75$ and $x(0) = 0.1$, we generate up to $N = 50$ iterations of the Bernoulli shift map with the J expresssion:

```
   10 5 $ ]x=: 0.75 bshift^:(i.50) 0.1
      0.1 0.133333 0.177778 0.237037 0.316049
 0.421399 0.561866 0.749154 0.998872 0.995488
 0.981954 0.927816 0.711264 0.948351 0.793405
 0.173622 0.231496 0.308661 0.411548 0.548731
 0.731641 0.975522 0.902087 0.608347  0.81113
  0.24452 0.326027 0.434702 0.579603 0.772804
0.0912144 0.121619 0.162159 0.216212 0.288283
 0.384377 0.512502 0.683336 0.911115  0.64446
  0.85928 0.437122 0.582829 0.777105  0.10842
  0.14456 0.192747 0.256995 0.342661 0.456881
```

Note that the sequence order reads from left to right, top to bottom. The resultant trajectory for the expression above is shown in Fig 7.9. The associated values of y are given by:

```
   10 5 $ ]y=: 0.75 (d_BMAP_ < bshift^:(i.50)) 0.1
0 0 0 0 0
0 0 0 1 1
1 1 0 1 1
0 0 0 0 0
0 1 1 0 1
0 0 0 0 1
0 0 0 0 0
0 0 0 1 0
1 0 0 1 0
0 0 0 0 0
```

Calculating the Lyaponov exponent for $N = 50$ yields a positive result, suggesting that the behaviour of the map is sensitive to initial conditions and is, therefore, chaotic:

```
   dx =: | -/ 0.75 bshift^:(20) &> 0.1 0.100001
   20 %~ ln dx%1e_6
0.617266
```

While the Bernoulli shift map is chaotic, it is Markovian in nature. That is, the sequence $x(n)$ is short-range dependent. To verify this, we iterate the map for $N = 10,000$:

```
   x1 =: 0.75 bshift^:(i.10000) 0.1
```

The autocorrelation coefficient for a range of lags k can be found by:

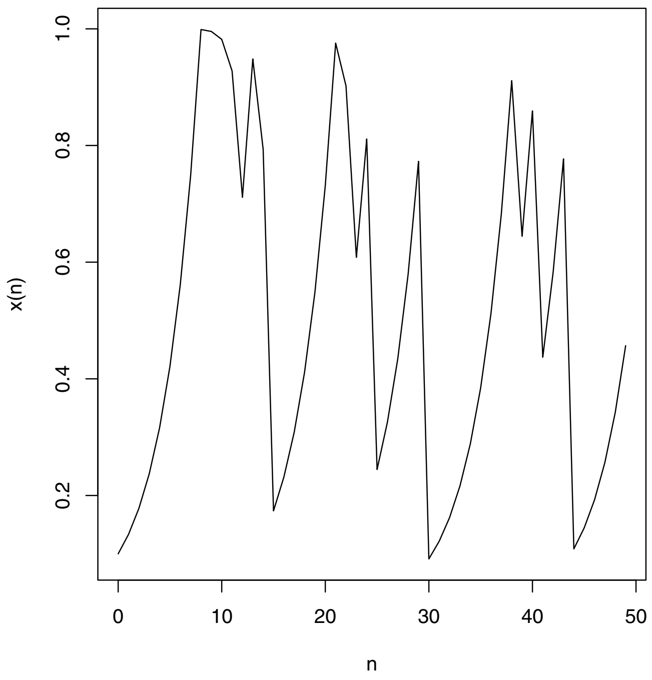

Fig. 7.9. First 50 iterations of $x(n)$ for the Bernoulli shift map, $d = 0.75$ and $x(0) = 0.1$

```
   k =: col i.100
   rx1 =: k autocor"1 x1
```

The graph in Fig 7.10 (top) shows that the autocorrelations decay exponentially with the lag, indicating that $x(n)$ is srd. For further confirmation, we calculate the variance time plot:

```
   m =: col 10 to 100
   vx1 =: m varm"1 x1
```

The slope is, as expected, approximately -1 (bottom graph in Fig 7.10). Computation of the Hurst parameter yields a value close to 0.5 indicating srd:

```
   hurst 0.952058
0.523971
```

Performing a similar analysis on the associated indicator variable y shows that it too, is srd:

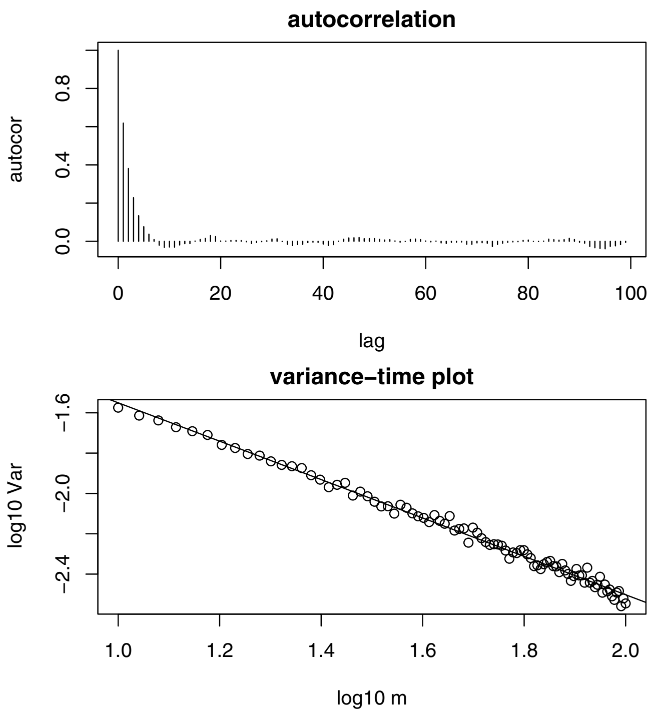

Fig. 7.10. Autocorrelation cofficient and variance-time plot for a Bernoulli shift map

```
   y1 =: 0.75 < x1
   vy1 =: m varm"1 y1
   hd3,: ;/ (log10 vy1) %. 1,.log10 m
+---------+--------+
|intercept|slope   |
+---------+--------+
|_0.742428|_1.00614|
+---------+--------+
   hurst 1.00614
0.49693
```

One problem with this map that we should mention is the case for $d = 0.5$. Below we run the map for $N = 60$ iterations:

```
   10 6 $ 0.5 bshift^:(i.60) 0.1
     0.1      0.2      0.4      0.8      0.6      0.2
     0.4      0.8      0.6      0.2      0.4      0.8
     0.6      0.2      0.4      0.8      0.6      0.2
     0.4      0.8      0.6      0.2      0.4      0.8
     0.6      0.2      0.4      0.8      0.6      0.2
     0.4      0.8      0.6      0.2      0.4      0.8
     0.6 0.200001 0.400002 0.800003 0.600006 0.200012
0.400024 0.800049 0.600098 0.200195 0.400391 0.800781
0.601562 0.203125  0.40625   0.8125    0.625     0.25
     0.5        1        1        1        1        1
```

We can see that the map *converges* to one. This is caused by the rounding of the digital computer and is not a property of the map itself. However, J does provide a solution to this problem. Instead of using floating point representation for d and $x(0)$, rationals can be used:

```
   10 6 $ 1r2 bshift^:(i.60) 1r10
1r10 1r5 2r5 4r5 3r5 1r5
 2r5 4r5 3r5 1r5 2r5 4r5
 3r5 1r5 2r5 4r5 3r5 1r5
 2r5 4r5 3r5 1r5 2r5 4r5
 3r5 1r5 2r5 4r5 3r5 1r5
 2r5 4r5 3r5 1r5 2r5 4r5
 3r5 1r5 2r5 4r5 3r5 1r5
 2r5 4r5 3r5 1r5 2r5 4r5
 3r5 1r5 2r5 4r5 3r5 1r5
 2r5 4r5 3r5 1r5 2r5 4r5
```

It is clear from the example above that the result does not suffer from the rounding problem. The graph in Fig 7.11 shows the trajectories for the floating point and fractional representations of the parameter d and initial condition $x(0)$.

7.2.2 Double Intermittency Map

Replacing the linear segments of the Bernoulli shift map with nonlinear segments results in a map that can (for certain parametric values) generate long-range dependent, self-similar trajectories. The double intermittency map, shown in Fig 7.12, is given by the transformation:

$$x(n+1) = \begin{cases} \epsilon_1 + x(n) + c_1 x(n)^m & \text{if} \quad 0 < x(n) \le d \\ 1 - \epsilon_2 - (1 - x(n)) - c_2(1 - x(n))^m & \text{if} \quad d < x(n) < 1 \end{cases} \tag{7.9}$$

where:

$$c_1 = \frac{1 - \epsilon_1 - d}{d^m} \quad \text{and} \quad c_2 = -\frac{\epsilon_2 - d}{(1-d)^m} \tag{7.10}$$

The parameters d, m, ϵ_1 and ϵ_2 are (as usual) passed on the left of the verb and $x(0)$ on the right:

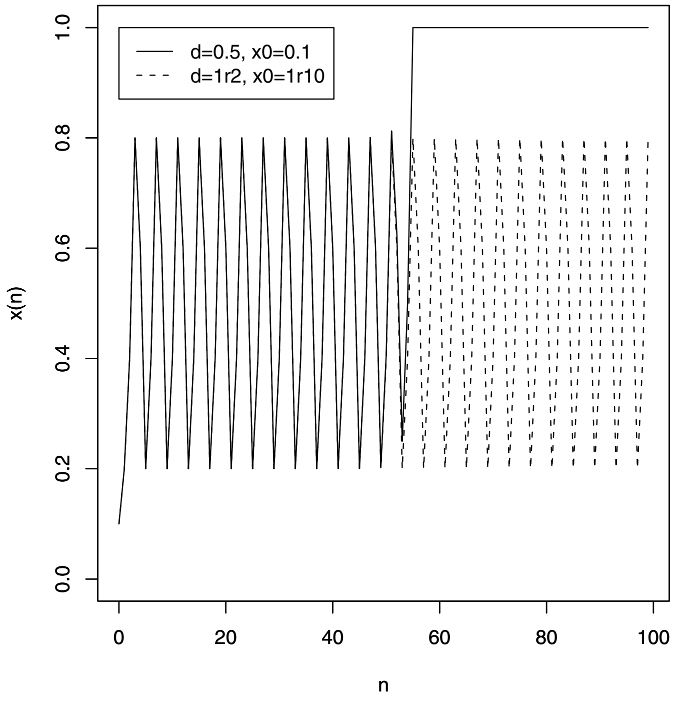

Fig. 7.11. Bernoulli shift map

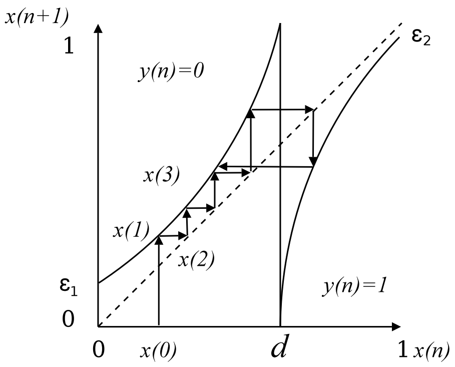

Fig. 7.12. Double intermittency chaotic map

```
   cocurrent < 'DIMAP'
   d  =: lhs1
   m  =: lhs2
   e1 =: lhs3
   e2 =: lhs4
   x  =: rhs1
```

The J expressions for the intermediate parameters c_1 and c_2 from Equation (7.10) are:

```
   c1 =: (>:@-@e1 - d) % d ^ m
   c2 =: -@(e2 - d) % >:@-@d ^ m
```

The expressions for the two (nonlinear) segments are given by:

```
   f1 =: e1 + x + c1 * x ^ m
   f2 =: -@e2 + x - c2 * >:@-@x ^ m
```

Thus, the transformation in Equation (7.9) is implemented in the same way as for the Bernoulli shift map:

```
   xnext =: f1 ` f2 @. (d<x)
```

Finally, we define the verb *dimap*:

```
   cocurrent < 'z'
   dimap =: xnext_DIMAP_
```

We generate up to $N = 50$ iterations of the double intermittency map for $d = 0.5$, $m = 2$, $\epsilon_1 = \epsilon_2 = 0.0001$ and $x(0) = 0.1$:

```
   10 5 $ 0.75 2 1e_4 1e_4 dimap^:(i.50) 0.1
     0.1 0.104543 0.109498 0.114925 0.120893
0.127486 0.134806  0.14298 0.152162 0.162548
0.174386 0.187997 0.203798  0.22235 0.244415
0.271054 0.303795 0.344897 0.397844 0.468263
0.565777 0.708089 0.930939 0.873615 0.681861
0.888516 0.739292 0.982207 0.978308 0.972563
 0.96343 0.947284 0.913842 0.824675 0.455756
0.548136 0.681718 0.888286 0.738445 0.980803
0.976282 0.969432 0.958121 0.936977 0.889221
0.741875  0.98649   0.9842 0.981105 0.976721
```

The graph in Fig 7.13 shows the trajectory for the double intermittency map. For analysis purposes, we run the map for $N = 10,000$ iterations:

```
   x2 =: 0.75 2 1e_4 1e_4 dimap^:(i.10000) 0.1
```

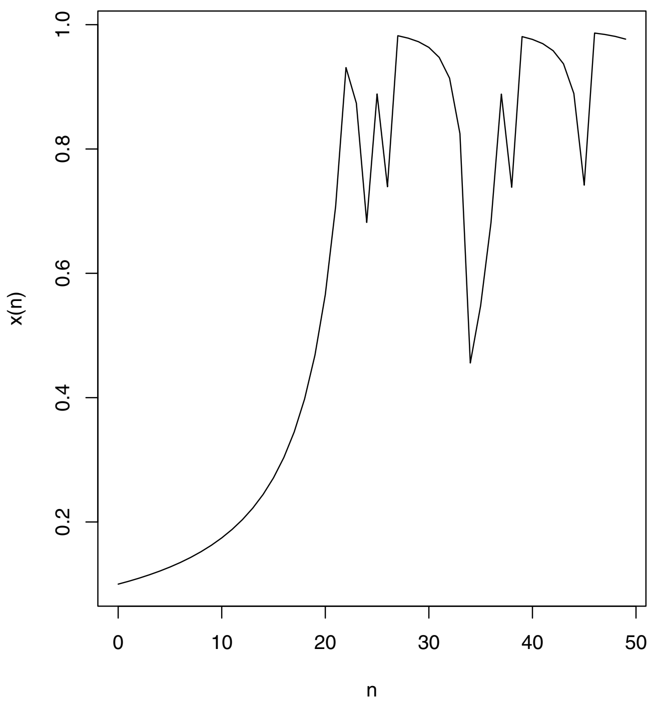

Fig. 7.13. First 50 iterations of x(n) for a Double intermittency chaotic map

Find the autocorrelation and variance time plot:

```
   rx2 =: k autocor"1 x2
   vx2 =: (col m) varm"1 x2
   hd3,: ;/ (log10 vx2) %. 1,.log10 m
+---------+---------+
|intercept|slope    |
+---------+---------+
|_0.625252|_0.352365|
+---------+---------+
```

The graphs in Fig 7.14 show the autocorrelation coefficients (top) and the variance time plot (bottom). The autocorrelations decay, as a power curve suggesting the trajectory, is long-range dependent. The slope of the variance time plot is greater than -1 which is an indication of self-similarity. This is confrmed by the computation of the Hurst parameter:

```
   hurst 0.352365
0.823817
```

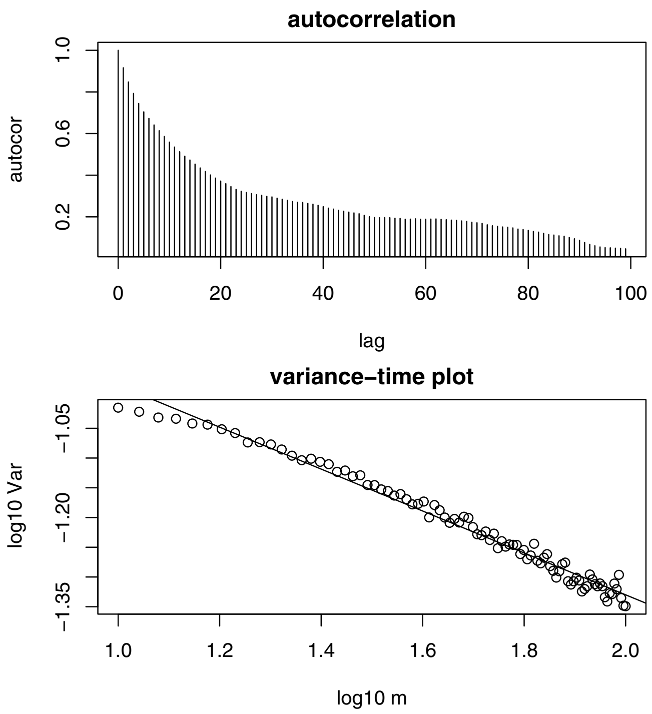

Fig. 7.14. Autocorrelation cofficient and variance-time plot for a Double intermittency chaotic map

7.2.3 Queue Dynamics

In this section, we compare the queue dynamics of a work conserving link for srd and lrd-ss processes, using the results of the Bernoulli shift and double intermittency maps described above. We can generate the on/off traffic sequence for the double intermittency map:

```
   y2 =: 0.75 < x2
```

The on/off sequence y_1 for the Bernoulli shift map was generated earlier (subsection 7.2.1). Comparing the average throughput of the respective traffic flows:

```
   hd4 =: 'y1';'y2'
   hd4,: ;/ mean y1,.y2
+------+------+
|y1    |y2    |
+------+------+
|0.2539|0.2342|
+------+------+
```

For each traffic source (y_1 and y_2), we can calculate the backlog for each time interval n, using the Lindley equation:

```
   q1 =: (0.6;y1) lindley^:(10000) 0
   q2 =: (0.6;y2) lindley^:(10000) 0
```

If we look at the mean backlog, we see that they are relatively similar:

```
   hd5 =: 'q1';'q2'
   hd5,: ;/ mean q1,.q2
+--------+-------+
|q1      |q2     |
+--------+-------+
|0.164164|1.48381|
+--------+-------+
```

There is a small difference between the backlog q_1 and q_2, although the mean throughput for the Bernoulli shift map was slightly higher than for the double intermittency. However, there is a more pronounced difference between the maximum backlogs of the two flows:

```
   hd5,: ;/ max q1,.q2
+---+----+
|q1 |q2  |
+---+----+
|2.4|23.6|
+---+----+
```

The maximum backlog q_2 is nearly 10 times higher than for q_1. The graph in Fig 7.15 shows the evolution of the queue sizes over time. It is clear that the queue dynamics for the two traffic flows are significantly different. This reinforces the assertion that the capacity planning of network resources should not be based solely upon first order statistics alone. The degree of lrd-ss has a significant impact on the performance seen at the buffer queues.

7.3 Summary

In this chapter we have used J to build and analyse chaotic maps. Chaotic behaviour is determined by a system's sensitivity to initial conditions. Long-term prediction of a dynamical system is dependent upon the assumption that two (or more) trajectories that have roughly (but not exactly) the same starting conditions will evolve in roughly the same way. This assumption is not valid for chaotic systems, where small perturbations in the starting conditions will result in widely varying outcomes

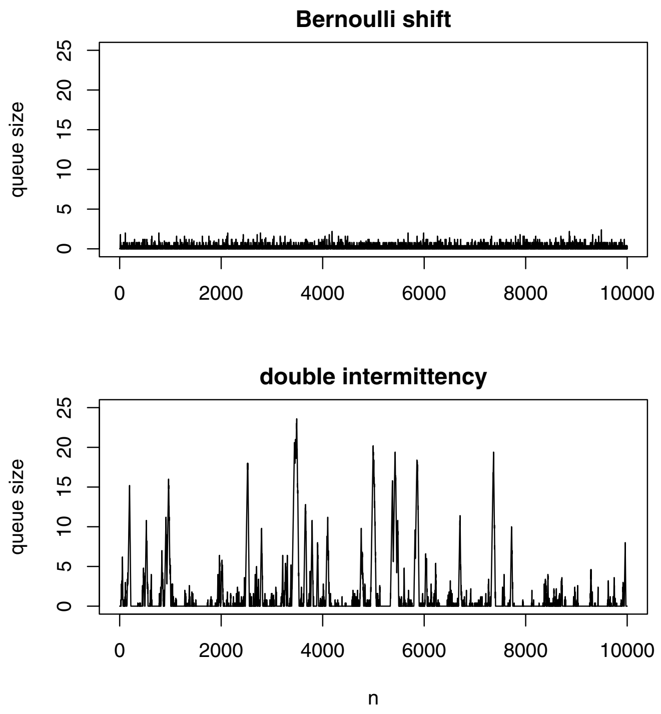

Fig. 7.15. Backlog for Bernoulli shift and double intermittency map traffic

in the system. The Lyapunov exponent $\lambda(x(0))$ provides a means of determining if a system is chaotic or not.

The Bernoulli shift and double intermittency map are two maps that can be used to model on/off traffic sources. The Bernoulli shift generates on/off periods that are srd, whereas the double intermittency map generates lrd-ss sequences.

8

ATM Quality of Service

Traffic flows through an ATM network are characterised by a number of traffic descriptors. The *peak cell rate* (PCR) is the maximum rate at which a source may submit cells to the network. The PCR is the reciprocal of the minimum cell spacing T; thus $\mathrm{PCR} = 1/T$. Cells that do not exceed the PCR are deemed to be conforming. While a source may submit conforming cells to the network, the effects of multiplexing could cause jitter in the cell spacing, resulting in nonconformance of cells as they propagate through the network. The network defines a *cell delay variation tolerance* (CDVT) that allows small bursts over the PCR.

Over a period of time greater than T, the *sustained cell rate* (SCR) defines the limit at which conforming cells may enter the network. A flow may be allowed to *burst* over the SCR (up-to the PCR). The burst "period" is defined by the *maximum burst size* parameter (MBS). The ATM Forum defines the following service categories:

- CBR: Constant bit rate
- VBR: Variable bit rate
- ABR: Available bit rate
- UBR: Unspecified bit rate

The CBR service category is for time-sensitive applications that require a fixed capacity, as specified by the PCR. Conforming traffic may *burst* over the PCR up to the CDVT. This service category is typically for circuit emulation, voice traffic or constant bit rate video.

For VBR traffic, conforming traffic may burst over the SCR up to the MBS. Like the CBR service category, it may burst over the PCR up to the CDVT.

The ABR service category is for traffic sources that can adapt their transmission rates to the conditions of the network through flow-control methods. As well as a PCR, the ABR service category also has a Minimum Cell Rate (MCR).

The UBR service category is defined by the PCR but there is no guaranteed throughput. Applications using this service category must have a high tolerance to delay. QoS is best-effort. In this chapter we focus on CBR and VBR service categories.

8.1 Generic Cell Rate Algorithm

The Generic Cell Rate Algorithm (GCRA) is a theoretical algorithm that determines whether a flow conforms to its contracted traffic descriptors. Cells that are deemed nonconforming are either discarded or *tagged*. Cells that are tagged are marked as low-priority. They may still pass through the network provided there is no congestion, but will be discarded if there is. In terms of the CBR service category, the GCRA is a function of two parameters $I = 1/\mathrm{PCR}$ and $L = \mathrm{CDVT}$. GCRA(I,L) can be implemented by either a *virtual scheduling algorithm* (VSA) or a *leaky bucket algorithm*.

Four parameters are defined for the VBR service category: $I_p = 1/\mathrm{PCR}$, $L_p = \mathrm{CDVT}$, $I_s = 1/\mathrm{SCR}$ and $L_s = \mathrm{BT}$. BT is the *burst tolerance*, and is given by:

$$\mathrm{BT} = (\mathrm{MBS} - 1) \times (1/\mathrm{SCR} - 1/\mathrm{PCR}) \qquad (8.1)$$

The J verb for calculating the burst tolerance from the MBS is shown in Listing 8.1.

Listing 8.1 *Burst Tolerance*

```
cocurrent < 'BT'
Is =: lhs1
Ip =: lhs2
mbs =: rhs0
f1 =: <:@mbs % Is-Ip
cocurrent < 'base'
burst_z_ =: f1_BT_
```

The dual virtual scheduling algorithm or the dual leaky bucket algorithm are used to implement VBR service categories.

8.2 Virtual Scheduling Algorithm and Leaky Bucket Algorithm

The theoretical arrival time (TAT) is the time that is compared to the arrival time of the next cell t_a in order to determine if it is conforming. If the cell arrives before $\mathrm{TAT} - L$, then it is nonconforming and conforming otherwise. The TAT is initialised to t_a when the first cell arrives. After checking the conformance of each cell, it is set

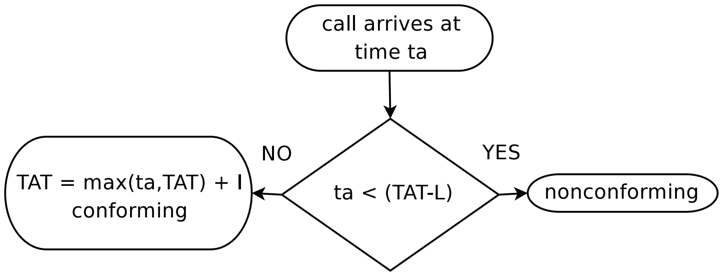

Fig. 8.1. Virtual scheduling algorithm

to t_a if the cell has arrived *after* the TAT, otherwise it is incremented by $1/\mathrm{PCR}$. The VSA is shown in Fig 8.1.

We implement the VSA in J. As the VSA and the leaky bucket algorithm (implemented below) share a number of common functions, we define these verbs in Listing 8.2. Listing 8.3 shows the virtual scheduling algorithm.

Listing 8.2 *Common Functions for VSA and Leaky Bucket*

```
cocurrent < 'GCRA'
I     =: lhs1
L     =: lhs2
ta    =: >@rhs1  NB. list of arrival times
clist =: >@rhs2  NB. conformance vector
ta1   =: {.@ta   NB. 1st element of ta list
tn    =: }.@ta   NB. tail of ta list
cocurrent < 'base'
```

Listing 8.3 *Virtual Scheduling Algorithm*

```
cocurrent < 'VSA'
TAT =: >@rhs3   NB. Theoretical Arrival Time
g1 =: TAT - L_GCRA_
g2 =: max @ (ta1_GCRA_, TAT) + I_GCRA_
conform =: ta1_GCRA_ < g1
f1 =: tn_GCRA_;(clist_GCRA_,0:);g2   NB. conforming
f2 =: tn_GCRA_;(clist_GCRA_,1:);TAT  NB. nonconforming
f3 =: f1 ` f2 @. conform
cocurrent < 'z'
vsa_z_ =: f3_VSA_
```

Generate some test data to demonstrate how the algorithm works. The sequence t_a is the cell arrival times:

```
   ta =: 1 6 7 18 20 21
```

The verb takes the traffic contract parameters, I and L, as left arguments. The right arguments are a boxed sequence. The first argument is the sequence of cell arrival times t_a. The next argument is a vector (list) of the conformance results. The function returns zero if a packet is conforming and one if it is nonconforming. It is initialised to a null list `''`. Conformance results are appended to the list as each cell is processed. The last argument is the TAT (which is initialised to the arrival time of the first cell). The example below shows the evolution of the virtual scheduling algorithm with traffic contract parameters $I = 4$ and $L = 2$.

```
   hd1 =: '';'ta';'conform';'TAT'
   hd1, (col i.7) ;"1 (4 2 vsa^:(i.7) ta;'';1)
+-+--------------+-----------+---+
| |ta            |conform    |TAT|
+-+--------------+-----------+---+
|0|1 6 7 18 20 21|           |1  |
+-+--------------+-----------+---+
|1|6 7 18 20 21  |0          |5  |
+-+--------------+-----------+---+
|2|7 18 20 21    |0 0        |10 |
+-+--------------+-----------+---+
|3|18 20 21      |0 0 1      |10 |
+-+--------------+-----------+---+
|4|20 21         |0 0 1 0    |22 |
+-+--------------+-----------+---+
|5|21            |0 0 1 0 0  |26 |
+-+--------------+-----------+---+
|6|              |0 0 1 0 0 1|26 |
+-+--------------+-----------+---+
```

An examination of the last iteration shows that cells three and six were nonconforming. The conformance results can be extracted thus:

```
   conform_z_ =: >@rhs2
   conform 4 2 vsa^:(6) ta;'';1
0 0 1 0 0 1
```

The leaky bucket algorithm defines a *bucket* with a maximum capacity L that leaks at a rate of $1/I$ (the PCR). When a cell arrives, the bucket fills by I. The algorithm records the occupancy of the bucket B and the last conformance time LCT. A cell is conforming provided the bucket is not full upon its arrival; that is, $\mathrm{B} - (t_a - \mathrm{LCT})$ does not exceed L.

The J verb for the implementation of the leaky bucket algorithm is shown in Listing 8.4 (see Fig 8.2):

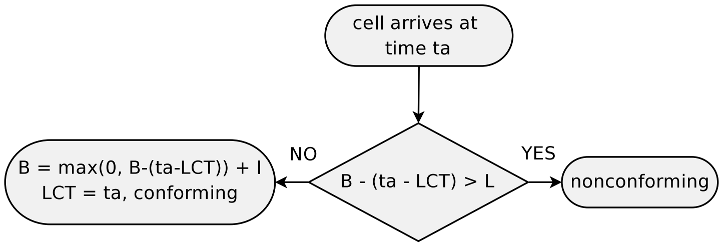

Fig. 8.2. Leaky-bucket algorithm

Listing 8.4 *Leaky Bucket Algorithm*

```
cocurrent < 'LB'
LCT =: >@rhs3
B =: >@rhs4
g1 =: B - (ta1_GCRA_ - LCT)
g2 =: max0@g1 + I_GCRA_
f1 =: tn_GCRA_;(clist_GCRA_,0:);ta1_GCRA_;g2
f2 =: tn_GCRA_;(clist_GCRA_,1:);LCT;B
conform =: g1 > L_GCRA_
f3 =: f1 ` f2 @. conform
cocurrent < 'base'
lb_z_   =: f3_LB_
```

As with the VSA, the traffic contract parameters are passed as left arguments. Furthermore, the first two (boxed) right arguments are the cell arrival times t_a and conformance results vector (initialised to null). The third argument is the LCT and initialised to the arrival time of the first cell. The last argument is the bucket occupancy B, which is initialised to zero. For $I = 4$ and $L = 2$; the evolution of the leaky bucket algorithm is:

```
   hd2 =: '';'ta';'conform';'LCT';'B'
   hd2, (col i.7) ;"1 (4 2 lb^:(i.7) ta;'';1;0)
+-+--------------+-----------+---+-+
| |ta            |conform    |LCT|B|
+-+--------------+-----------+---+-+
|0|1 6 7 18 20 21|           |1  |0|
+-+--------------+-----------+---+-+
|1|6 7 18 20 21  |0          |1  |4|
+-+--------------+-----------+---+-+
|2|7 18 20 21    |0 0        |6  |4|
+-+--------------+-----------+---+-+
|3|18 20 21      |0 0 1      |6  |4|
+-+--------------+-----------+---+-+
|4|20 21         |0 0 1 0    |18 |4|
+-+--------------+-----------+---+-+
|5|21            |0 0 1 0 0  |20 |6|
+-+--------------+-----------+---+-+
|6|              |0 0 1 0 0 1|20 |6|
+-+--------------+-----------+---+-+
```

It can be seen that the results for the conformance of cells are the same as the VSA.

8.2.1 Jitter

We demonstrate how the CDVT can be used to account for cell jitter. Randomly selecting cells to jitter (with a probability of 0.4) by generating a sequence of Bernoulli trials:

```
   load 'stats.ijs' NB. load RNGs
   ]a =: 1r8*(0.4 rber 10)
1r8 1r8 0 0 1r8 0 0 1r8 0 0
```

We then choose (with a probability of $1/2$) to jitter cells positively or negatively:

```
   ]b =: *rnorm 10
1 1 _1 _1 1 _1 1 1 1 _1
```

Note that, the use of RNGs above will yield different results for a and b each time the command-lines above are issued. In order to replicate the results here, set a and b explicitly:

```
   a =: 1r8 1r8 0 0 1r8 0 0 1r8 0 0
   b =: 1 1 _1 _1 1 _1 1 1 1 _1
```

The product of the sequence above results in a sequence of "jitter" values that are added to the sequence of packet times.

```
   ]jitter =: a*b
1r8 1r8 0 0 1r8 0 0 1r8 0 0
```

For a CBR service with PCR=4 cells per unit time, the flow t_a is conforming, as the minimum spacing between cells is $1/4$ (time units):

```
   ]ta =: +/\ 10 # 1r4
1r4 1r2 3r4 1 5r4 3r2 7r4 2 9r4 5r2
```

Even with a CDVT $= 0$, cells are conforming:

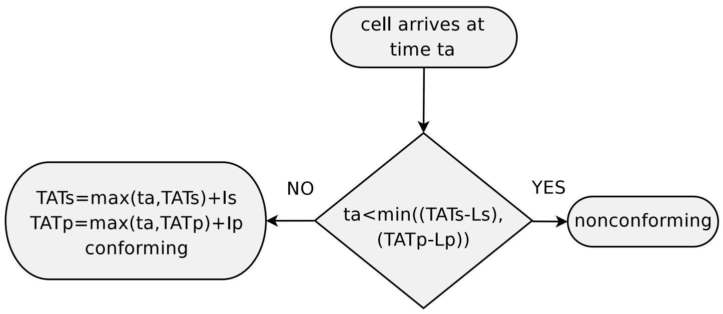

Fig. 8.3. Dual VSA

```
   conform 1r4 0 lb^:(10) ta;'';1r4;0
0 0 0 0 0 0 0 0 0 0
```

Applying jitter reduces the minimum cell spacing to $1/8$:

```
   ]taj =: ta+jitter
3r8 5r8 3r4 1 11r8 3r2 7r4 17r8 9r4 5r2
   min ipa taj
1r8
```

If we run the leaky bucket algorithm with $\text{CDVT} = 0$ on the jittered cell times, we get nonconforming cells:

```
   conform 1r4 0 lb^:(10) taj;'';1r8;0
0 0 1 0 0 1 0 0 1 0
```

However, if we set $\text{CDVT} = 1/8$, then all cells in the (jittered) flow are deemed to be conforming:

```
   conform 1r4 1r8 lb^:(10) taj;'';1r8;0
0 0 0 0 0 0 0 0 0 0
```

8.3 Dual Virtual Scheduling Algorithm and Dual Leaky Bucket

For VBR services, the traffic contract parameters are I_s, I_p, L_s and L_p. Listing 8.5 shows the common functions for the dual VSA and the dual leaky bucket algorithms. Listings 8.6 and 8.7 show the J verb definition for the dual VSA (Fig 8.3) and dual leak-bucket algorithm (Fig 8.4), respectively.

Listing 8.5 *Common Functions for Dual VSA and Dual Leaky Bucket*

```
cocurrent < 'GCRA'
Is =: lhs1
Ip =: lhs2
Ls =: lhs3
Lp =: lhs4
cocurrent < 'base'
```

Listing 8.6 *Dual Virtual Scheduling Algorithm*

```
cocurrent < 'DVSA'
TATs =: >@rhs3
TATp =: >@rhs4
g1 =: TATs - Ls_GCRA_
g2 =: TATp - Lp_GCRA_
g3 =: min @ (g1,g2)
conform =: ta1_GCRA_ < g3
TATsnext =: max @ (ta1_GCRA_, TATs) + Is_GCRA_
TATpnext =: max @ (ta1_GCRA_, TATp) + Ip_GCRA_
f1 =: tn_GCRA_;(clist_GCRA_,0:);TATsnext;TATpnext
f2 =: tn_GCRA_;(clist_GCRA_,1:);TATs;TATp
f3 =: f1 ` f2 @. conform
cocurrent < 'base'
dvsa_z_  =: f3_DVSA_
```

Arrival times are assigned to the flow t_{a_2} by the J expression:

```
   ta2 =: 1 3 5 6 7 8 10 14 17 20
```

We run the dual VSA for $I_s = 4$, $I_p = 2$, $L_s = 7$, and $L_p = 2$:

```
   hd3 =: 'ta';'conform';'TATs';'TATp'
   hd3, 4 2 7 2 dvsa^:(i.11) ta2;'';1;1
+------------------------+--------------------+----+----+
|ta                      |conform             |TATs|TATp|
+------------------------+--------------------+----+----+
|1 3 5 6 7 8 10 14 17 20|                    |1   |1   |
+------------------------+--------------------+----+----+
|3 5 6 7 8 10 14 17 20   |0                   |5   |3   |
+------------------------+--------------------+----+----+
|5 6 7 8 10 14 17 20     |0 0                 |9   |5   |
+------------------------+--------------------+----+----+
|6 7 8 10 14 17 20       |0 0 0               |13  |7   |
+------------------------+--------------------+----+----+
|7 8 10 14 17 20        |0 0 0 0            |17  |9   |
+-----------------------+-------------------+----+----+
|8 10 14 17 20          |0 0 0 0 1          |17  |9   |
+-----------------------+-------------------+----+----+
|10 14 17 20            |0 0 0 0 1 1        |17  |9   |
+-----------------------+-------------------+----+----+
|14 17 20               |0 0 0 0 1 1 0      |21  |12  |
+-----------------------+-------------------+----+----+
|17 20                  |0 0 0 0 1 1 0 0    |25  |16  |
+-----------------------+-------------------+----+----+
|20                     |0 0 0 0 1 1 0 0 1  |25  |16  |
+-----------------------+-------------------+----+----+
|                       |0 0 0 0 1 1 0 0 1 0|29  |22  |
+-----------------------+-------------------+----+----+
```

Listing 8.7 *Dual Leaky bucket*

```
cocurrent < 'DLB'
LCT =: >@rhs3
Bs =: >@rhs4
Bp =: >@rhs5
g1 =: -/@ (Bs,ta1_GCRA_,LCT)
g2 =: -/@ (Bp,ta1_GCRA_,LCT)
g3 =: max0@g1 + Is_GCRA_
g4 =: max0@g2 + Ip_GCRA_
conform =: (g1 > Ls_GCRA_) or g2 > Lp_GCRA_
f1 =: tn_GCRA_;(clist_GCRA_,0:);ta1_GCRA_;g3;g4
f2 =: tn_GCRA_;(clist_GCRA_,1:);LCT;Bs;Bp
f3 =: f1 ` f2 @. conform
cocurrent < 'base'
dlb_z_  =: f3_DLB_
```

Running the dual leaky bucket algorithm on t_{a_2} shows that the conformance results are the same as the virtual scheduling algorithm:

```
   conform 4 2 7 2 dlb^:(10) ta2;'';1;0;0
0 0 0 0 1 1 0 0 1 0
```

8.4 Analysing Burst Tolerance

In this section, we analyse the burst tolerance parameter of the GCRA with respect to the srd and lrd-ss *variable bit rate* sources. In Section 6.5 we presented a number of discrete on/off traffic models. These models produced traffic as a number of arrivals

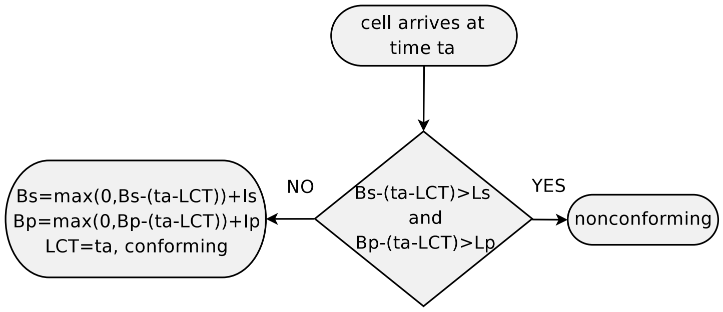

Fig. 8.4. Dual leaky bucket

per (discrete) time interval. However, the GCRA requires traffic in the form of a sequence of cell (or packet) arrival times. Fig 8.5 shows the Markov model for a *continuous* on/off source.

Like the discrete Markov model, the source transmits at a constant rate during the on period and is idle during the off period. However, the time periods are not represented as discrete time units, but are instead continuous. The lengths of the on and off periods are determined by the respective transition frequencies a and b. As we have seen from the discrete models, the superposition of on/off sources with heavy-tailed distributions leads to lrd-ss traffic. The same applies to continuous models. We present two J verbs, one for generating srd and the other lrd-ss.

The J verb *oosrdc* in Listing 8.8 generates cell arrivals times according to exponentially distributed on and off times (Exp[on]-Exp[off]).

Listing 8.8 *Exp[on]-Exp[off] Model*

```
cocurrent < 'OOEXPEXP'
recipa =: lhs1  NB. on rate
recipb =: lhs2  NB. off rate
T =: lhs3     NB. packet rate
n  =: rhs1    NB. no. on and off times
f1 =: recipa exprand n  NB. on duration
f2 =: recipb exprand n  NB. off duration
f3 =: (T) max f1
np =: ceil@(f3%T) NB. no. packets
f4 =: np # each T
f5 =: f2 (,~>) each f4
f6 =: ;@f5
cocurrent < 'base'
oosrdc =: +/\@f6_OOEXPEXP_
```

The J verb *oolrdc* in Listing 8.9 generates cell arrivals times according to exponentially distributed on and Pareto distributed off times (Exp[on]-Par[off]).

Listing 8.9 *Exp[on]-Par[off] Model*

```
cocurrent < 'OOEXPPAR'
recipa =: lhs1
alpha =: lhs2
T =: lhs3
n  =: rhs1
f1 =: recipa exprand n
f2 =: (alpha,T) (ceil@rpar_z_) n
f3 =: (T) max f1
np =: ceil@(f3%T) NB. no. packets
f4 =: np # each T
f5 =: f2 (,~>) each f4
f6 =: ;@f5
cocurrent < 'base'
oolrdc =: +/\@f6_OOEXPPAR_
```

The *oosrdc* verb takes three left parameters, the transition frequencies of the on and off periods, and the peak transmission rate (in cells of packets per unit of time). We generate srd traffic for $a = 8$, $b = 12$ and $r_{\text{peak}} = 1/2$ (thus the mean packet inter-arrival time is $T = 2$). We show that the average transmission rate is approximately 0.2, by dividing the number of packets by the difference between the first and last arrival time:

```
   (#%max) 8 12 2 oosrdc 10000
0.237995
```

Given the values of the parameters a, b and r_{peak} we can compute the average transmission rate:

$$r_{\text{peak}}\frac{a}{a+b} = \frac{1}{2} \times \frac{8}{8+12} = 0.2 \tag{8.2}$$

The verb *oolrdc* takes the same parameters as *oosrdc*, except for the transition α of the Pareto distribution.

```
   2 mpar 1.2
12
```

The mean transmission rate for $a = 8$, $\alpha = 1.2$ and $r_{\text{peak}} = 1/2$ is:

```
   (#%max) 8 1.2 2 oolrdc 10000
0.216951
```

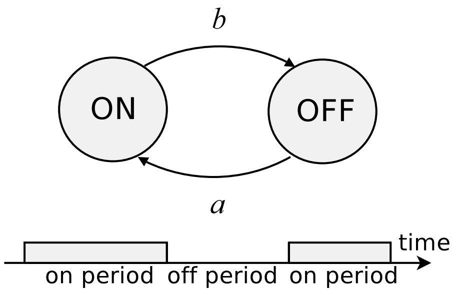

Fig. 8.5. Continuous On/Off source model

Now, we can use these source models to examine the effect of the burst tolerance (BT) parameter on conformance. We generate five lrd-ss continuous on/off sources and "aggregate" them by sorting them in ascending order of arrival time. We then take a sample from the middle of the sequence:

```
   hd4 =: 'min';'max'
   x1 =: sort ; (2 1.2 1) (oolrdc"1) (col 5 # 1000)
   hd4,: (min;max) (tal =: (2001 to 12000) { x1)
+-------+-------+
|min    |max    |
+-------+-------+
|1030.12|6943.56|
+-------+-------+
```

Similarly, an aggregate of five srd flows is generated:

```
   x2 =: sort ; (2 6 1) (oosrdc"1) (col 5 # 1000)
   hd4,: (min;max) (tas =: (2001 to 12000) { x2)
+-------+-------+
|min    |max    |
+-------+-------+
|606.355|5882.56|
+-------+-------+
```

Calculate the mean transmission rates of both the lrd-ss flow and srd flow by dividing the number of cells by the difference between the first and last arrival times:

```
   hd5 =: 'lrd-ss';'srd'
   hd5,: ;/ (#%max-min)"1 tal,:tas
+------+-------+
|lrd-ss|srd    |
+------+-------+
|1.8953|1.69114|
+------+-------+
```

In order to verify that traffic flows are lrd-ss and srd respectively, the flow is converted from its cell arrival time form to cells per (unit) time interval:

```
   al =: ipa tal ([: +/ </) (607 to 5881)
   as =: ipa tas ([: +/ </) (1031 to 6942)
```

We can confirm that the lrd-ss traffic process shows a reasonably high degree of self-similarity, with $H \approx 0.86$:

```
   hd6 =: 'intercept';'slope'
   m =: col 10 to 100
   vl =: m varm"1 al
   hd6,: ;/ (log10 vl) %. 1,.log10 m
+---------+---------+
|intercept|slope    |
+---------+---------+
|0.0709503|_0.283643|
+---------+---------+
   hurst 0.283643
0.858178
```

Furthermore, for the srd traffic flow, $H \approx 0.55$, indicating that it is indeed short-range dependent:

```
   vs =: m varm"1 as
   hd6,: ;/ (log10 vs) %. 1,.log10 m
+---------+---------+
|intercept|slope    |
+---------+---------+
|0.441152 |_0.898582|
+---------+---------+
   hurst 0.898582
0.550709
```

The traffic contract TC_1 consists of the set of parameters: I_s, I_p, L_s and L_p. These parameters are assigned values accordingly:

```
   hd7 =: 'Is';'Ip';'Ls';'Lp'
   hd7,: ;/TC1 =: 1r3 1r5 0 1r5
+---+---+--+---+
|Is |Ip |Ls|Lp |
+---+---+--+---+
|1r3|1r5|0 |1r5|
+---+---+--+---+
```

Note that, we set the burst tolerance (L_s) to zero. The J expression below computes the conformance of both flows using the dual leaky bucket algorithm. It is necessary to initialise $t_a(0)$ for each flow to the time of the first packet. The initialisation values are given by computation of the minimum of x_1 and x_2 above:

```
   cl1 =: conform TC1 dlb^:(10000) tal;'';606.355;0;0
   cs1 =: conform TC1 dlb^:(10000) tas;'';1030.12;0;0
```

The proportion of nonconforming cells is high for both flows:

```
   hd5,: ;/ mean"1 cl1,:cs1
   hd5,: ;/ mean"1 cl1,:cs1
+------+------+
|lrd-ss|srd   |
+------+------+
|0.3511|0.3303|
+------+------+
```

We now examine the effect of the burst tolerance (BT) on cell conformance. Define a new traffic contract TC_2. The parameters I_s, I_p, L_p remain the same but L_s is set to $1/3$.

```
   hd7 ,: ;/TC2 =: 1r3 1r5 1r3 1r5
+---+---+---+---+
|Is |Ip |Ls |Lp |
+---+---+---+---+
|1r3|1r5|1r3|1r5|
+---+---+---+---+
```

We run the dual leaky bucket algorithm for the new traffic contract:

```
   cl2 =:  conform TC2 dlb^:(10000) tal;'';606.355;0;0
   cs2 =:  conform TC2 dlb^:(10000) tas;'';1030.12;0;0
```

For a nonzero burst tolerance, the proportion of nonconforming cells is reduced for both flows; however, for the srd flow, the reduction is far more pronounced:

```
   hd5,: ;/ mean"1 cl2,:cs2
+------+-----+
|lrd-ss|srd  |
+------+-----+
|0.1805|0.074|
+------+-----+
```

8.5 Summary

ATM specifies a number of service categories, which in turn are defined by a number of traffic descriptors. In this chapter we have focused on two particular service categories: CBR and VBR. The conformance of traffic flows to either of these service categories is determined by the Generic Cell Rate Algorihtm (GCRA). Either the virtual scheduling algorithm (VSA) or leaky bucket algorithm can be used for the GCRA.

The single VSA or leaky bucket algorithm is used for testing conformance to the CBR service category, whereas the dual version of the algorithm is used for the VBR service category.

We generated srd and lrd-ss traffic using a continuous on/off traffic model implemented in J. We analysed the conformance of simulated traffic using the GCRA (dual leaky bucket). Traffic that is lrd-ss yields higher conformance failure rates over comparable srd traffic.

9

Congestion Control

Congestion control and avoidance, introduced by Van Jacobson [29] in 1988, has had a significant effect on the stability of the Internet. The TCP sender maintains a *congestion window* that determines the number of unacknowledged segments that it can deliver to a receiver. When the sender has transmitted an entire window of data, it must wait until it receives an acknowledgement before it can *slide* the window along and release more data into the network. Therefore, the sender's data rate is constrained by the reciprocal of the round-trip time delay.

The sender will, however, adjust the congestion window during the course of a session. It initialises the congestion window to a low value (e.g., one or two segments) and increases it each time an acknowledgement is received. In this way, the sender *probes* the network for available capacity. When the sender detects congestion in the network (typically inferred after packet loss), it responds by reducing its congestion window. Having dropped its congestion window (and therefore its transmission rate), the sender once again starts probing the network. How aggressively the sender probes the network and how conservatively it responds to congestion is governed by the algorithm and its parameters.

TCP ensures reliable data delivery by retransmitting unacknowledged segments. The delay incurred by retransmitting segements is a problem for certain applications, such as real-time audio and video, where dropped packets are preferred to late packets. The alternative is to use the UDP (User Datagram Protocol) transport protocol, but this has no congestion control mechanism at all. However, a number of *TCP-compatible* algorithms have been proposed and examined [75] for emerging Internet transport protocols, such as DCCP (Datagram Congestion Control Protocol) [24]. DCCP is designed for real-time applications, specifically audio, video and online gaming. There has been a growing interest in the analysis of these algorithms in order to understand, not only how they can provide adequate congestion control for multimedia applications, but also what effect they have on existing TCP congestion controlled flows.

9.1 A Simple Congestion Control Algorithm

In this section, we present a simple congestion control model based upon the one introduced in [63]. We consider traffic flows from two sources transmitting over a bottleneck link of capacity c. We assume that each source j is greedy and transmits up to the rate of its congestion window $w = \{w_j, j = 1, 2\}$ (per time interval). The offered load to the communications link at any time t is $w_1(t) + w_2(t)$.

Congestion occurs when the aggregate transmission rate of the two sources exceeds the capacity of the link. Since we assume that both sources are greedy, then the link experiences congestion when $w_1(t) + w_2(t) > c$. I_{con} and $I_{\overline{\mathrm{con}}}$ are indicator functions that signal a congestion condition on the link:

$$I_{\mathrm{con}} = \begin{cases} 1 & \text{if } w_1(t) + w_2(t) > c \\ 0 & \text{if } w_1(t) + w_2(t) \leq c \end{cases} \tag{9.1}$$

$$I_{\overline{\mathrm{con}}} = \begin{cases} 1 & \text{if } w_1(t) + w_2(t) \leq c \\ 0 & \text{if } w_1(t) + w_2(t) > c \end{cases} \tag{9.2}$$

The congestion window for each source j evolves according to the feedback equation:

$$w_j(t+1) = w_j(t) + \alpha I_{\overline{\mathrm{con}}} - \beta I_{\mathrm{con}} \tag{9.3}$$

Thus, the congestion window is decremented by β in the event of congestion; otherwise it is incremented by α. Here, we present a J verb for this simple congestion control algorithm. The verb *cwnd* takes three (static) parameters, c, α and β. The positional parameter verbs are required, so load in the *libs.ijs* script:

```
load 'libs.ijs'
```

Define c, α and β:

```
cocurrennt <'CWND'
c     =: lhs1  NB. get 1st left parameter
alpha =: lhs2  NB. get 2nd left parameter
beta  =: lhs3  NB. get 3rd left parameter
```

The window size is passed as a right argument:

```
w =: rhs0
```

The flows w_j are summed and then checked to see if the resulting sum exceeds the link capacity c. The two mutually exclusive indicator variables I_{con} and $I_{\overline{\mathrm{con}}}$ (*Incon*) set accordingly:

```
a      =: +/@w      NB. aggregate data flow
Icon   =: a > c     NB. 1 if congestion, 0 otherwise
Incon   =: -.@Icon NB. 1 if no congestion, 0 otherwise
```

The expression in Equation (9.3) is implemented by the verb:

```
   wnext =: 0.0001&max@(w + (alpha*Incon) - beta*Icon)
```

We ensure that the window size does not drop below a minimum threshold (set arbitrarily to 0.001 in this case). Finally, we define the verb *cwnd*:

```
   cocurrent <'base'  NB. return to base locale
   cwnd_z_ =: wnext_CWND_
```

We execute the function with parameters $c = 1$, $\alpha = 0.3$ and $\beta = 0.4$, and the initial conditions $w_1(0) = 0.3$ and $w_2(0) = 0.1$:

```
   (col i.5) ; 1 0.3 0.4 cwnd^:(i.5) 0.3 0.1
+-+-------+
|0|0.3 0.1|
|1|0.6 0.4|
|2|0.9 0.7|
|3|0.5 0.3|
|4|0.8 0.6|
+-+-------+
```

It shows that congestion occurred by iteration two, as both flows reduced their window size in the next iteration. In this example. while w_1 and w_2 have different starting points, their parameters (and thus their response to the network conditions) are the same. However, we can give each flow different parameters by constructing a parameter matrix:

```
   ]P1 =: 1 0.1 0.15 ,. 1 0.2 0.4
   1   1
 0.1 0.2
0.15 0.4
```

Here, we set $\alpha_1 = 0.1$, $\beta_1 = 0.15$, $\alpha_2 = 0.2$ and $\beta_2 = 0.4$. As the flows share a common link, then: $c_1 = c_2 = 1$. Now, each flow probes the network for capacity and responds to congestion differently:

```
   (col i.5) ; P1 cwnd^:(i.5) 0.3 0.1
+-+--------+
|0| 0.3 0.1|
|1| 0.4 0.3|
|2| 0.5 0.5|
|3| 0.6 0.7|
|4|0.45 0.3|
+-+--------+
```

While the feedback model we developed in Equation (9.3) is for two flows only, we can, in fact, process an arbitrary number of flows, provided we pass a parameter matrix of the appropriate order and with a corresponding number of initial conditions. For the purpose of illustration the parameters of the three flows are chosen to be:

```
   ]P2 =: 1 0.1 0.15 ,. 1 0.2 0.4 ,. 1 0.3 0.3
    1   1   1
  0.1 0.2 0.3
 0.15 0.4 0.3
```

No fundamental change to the implementation or execution of *cwnd* is required:

```
   (col i.5) ; P2 cwnd^:(i.5) 0.3 0.1 0.2
+-+---------------+
|0| 0.3    0.1 0.2|
|1| 0.4    0.3 0.5|
|2|0.25 0.0001 0.2|
|3|0.35 0.2001 0.5|
|4| 0.2 0.0001 0.2|
+-+---------------+
```

The graph in Fig 9.1 shows the congestion control window (and thus transmission rate) for all three flows. While this example is somewhat artificial, it does provide some insight into the effects that the parameters α (how aggressively the network is probed) and β (how conservative is the response to congestion) have on the allocation of resources amongst the flows.

This model illustrates the fundamental principles behind Internet congestion control, as well as introducing how to develop functions in the J programming language. Actual Internet congestion control algorithms are far more complex than the model presented here. Furthermore, any flow's reaction to network conditions is regulated by its round-trip. A flow with large round-trip times will not increase its window as rapidly as a flow with small round-trip times. Nor will it respond to congestion as quickly. With our current model, flows are *synchronised*; that is, they experience the same round-trip times and thus respond to network conditions at the same time (in the next time interval). In the next two sections, we develop models for TCP-compatible and TCP congestion control algorithms that incorporate a number of enhancements and improve on this simple example. These models reflect more closely the behaviour of the actual Internet congestion control algorithms. They also account for the variance in round-trip times across different flows.

9.2 Binomial Congestion Control Algorithms

The binomial congestion control algorithm below defines a set of congestion control algorithms [3, 30, 56]:

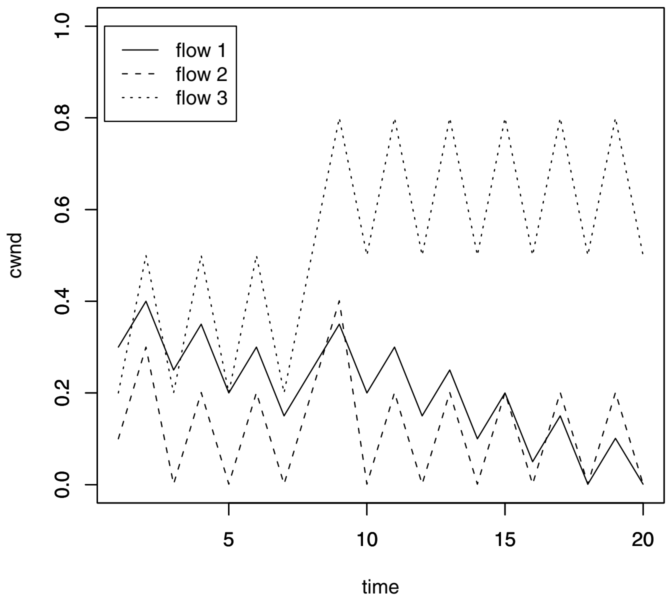

Fig. 9.1. Congestion window of three flows sharing a communications link

$$\begin{array}{ll}\text{Increase:} & w_j(t+d_{RTT}) \leftarrow w_j(t) + \alpha/w_j(t)^k \\ \text{Decrease:} & w_j(t+d_{RTT}) \leftarrow w_j(t) - \beta w_j(t)^l\end{array} \tag{9.4}$$

The parameters k and l define the nature of window increase and decrease, e.g, $k = 0$ and $l = 1$ defines an algorithm with an *additive* increase and a *multiplicative* decrease (AIMD). Other algorithms include multiplicative increase, multiplicative decrease (MIMD) and additive increase, additive decrease (AIAD). Table 9.1 shows corresponding values of k and l for various algorithms.

Algorithm	k	l	Increase	Decrease
AIMD	0	1	additive	multiplicative
MIMD	-1	1	multiplicative	multiplicative
MIAD	-1	0	multiplicative	additive
AIAD	0	0	additive	additive
IIAD	1	0	inverse	additive
SQRT	0.5	0.5	square root	square root

Table 9.1. Binomial congestion control algorithms

The parameters α and β allow for further control of the increase and decrease terms, but do not alter the nature of the algorithm. That is, AIMD still increases the window additively and decreases it multiplicatively, irrespective of the values of α and β. We use $\alpha = 1$ and $\beta = 0.5$ throughout this chapter. The function $w_j(t)$ gives the congestion window size for the j^{th} flow:

$$w_j(t + d_{RTT_j}) = w_j^{\triangle}(t) I_{\overline{\text{con}}} + w_j^{\bigtriangledown}(t) I_{\text{con}} \tag{9.5}$$

where d_{RTT_j} is the acknowledgement round-trip time for flow j, $w_j^{\triangle}(t)$ is the window size increase upon receiving an acknowledgement and $w_j^{\bigtriangledown}(t)$ is the window size decrease in response to congestion, where $w_j^{\triangle}(t)$ and $w_j^{\bigtriangledown}(t)$ are given by:

$$\begin{array}{ll} w_j^{\triangle}(t) = w_j(t) + \alpha / w_j(t)^k & \text{no congestion} \\ w_j^{\bigtriangledown}(t) = w_j(t) - \beta w_j(t)^l & \text{congestion} \end{array} \tag{9.6}$$

The aggregate offered load $a(t)$ to the network is the sum of the individual flow rates in the time interval t. We assume that all flows are greedy and transmit up to the maximum rate allowed by the congestion window, which we take to be $r_j(t) = w_j(t)/d_{RTT_j}$; thus $a(t)$ is:

$$a(t) = \sum_{j=1}^{j=N} r_j(t) \tag{9.7}$$

In our initial example presented in Section 9.1, the *contention* resource was a bufferless link. A congestion condition occurs when the aggregate flow exceeded the capacity $a(t) > c$. Here, we consider a communication link with a buffer queue. We use the Lindley equation [39] to determine the backlog at the communications link. The buffer queue is finite, so we drop any traffic that exceeds b:

$$q(t+1) = \max[b, (a(t+1) + q() - c)^+] \tag{9.8}$$

The congestion/no congestion indicators I_{con} and $I_{\overline{\text{con}}}$ are given by the two expressions below:

$$I_{\text{con}} = \begin{cases} 1 & \text{if } q(t) > b \\ 0 & \text{if } q(t) \le b \end{cases} \tag{9.9}$$

$$I_{\overline{\text{con}}} = \begin{cases} 1 & \text{if } q(t) \le b \\ 0 & \text{if } q(t) > b \end{cases} \tag{9.10}$$

The parameters to this function are c, b, α, β , k, l and d_{RTT} (refer to Table 9.2 for associated mathematical and J terms). These are passed as a left argument and are processed by the functions:

```
load 'libs'ijs' NB. load pos. param verbs
cocurrent <'TCPC'
```

Math term	J term	Comment
c	*c*	capacity of the link
b	*b*	buffer threshold
α	*alpha*	window increase factor
β	*beta*	window decrease factor
k	*k*	window increase param.
l	*l*	window decrease param.
$w_j(t)$	*w*	current window of flow j
$w_j(t+1)$	*wnext*	next window of flow j
$w_j^{\triangle}(t)$	*wi*	window increase
$w_j^{\triangledown}(t)$	*wd*	window decrease
$q(t)$	*q*	current backlog
$q(t+1)$	*qnext*	next backlog
$n_{\mathrm{ack}}(t)$	*ack*	number of acks
$n_{\mathrm{ack}}(t+1)$	*acknext*	next number of acks
d_{RTT_j}	*RTT*	round-trip time for flow j
$r_j(t)$	*tx*	current transmission rate
$r_j(t+1)$	*txnext*	next transmission rate
$a(t)$	*a*	aggregate offered load
I_{con}	*Icon*	congestion indicator
$I_{\overline{\mathrm{con}}}$	*Incon*	no congestion indicator
$\delta_{ssthresh}(t)$	*ssthresh*	current slow-start threshold
$\delta_{ssthresh}(t+1)$	*ssthnext*	next slow-start threshold
ω_{flow}	*flow*	flow control window

Table 9.2. J terms and associated mathematical terms

```
RTT   =: lhs1
c     =: lhs2
b     =: lhs3
alpha =: lhs4
beta  =: lhs5
k     =: lhs6
l     =: lhs7
```

The window size w, backlog q, time t and the cumulative number of acknowledgements ack are passed as boxed right arguments:

```
t        =: >@rhs1
w        =: >@rhs2
q        =: >@rhs3
ack      =: >@rhs4
tx       =: >@rhs5
```

The aggregate flow (offered load) (*a*) and backlog (*qnext*) are calculated, then the congestion/no congestion indicators (I_{con} and $I_{\overline{\text{con}}}$ respectively) are set accordingly (depending upon whether the backlog exceeds the buffer threshold):

```
txnext =: w%RTT
a =: +/@txnext                NB. aggregate flow
backlog =: max0@(a + q - c) NB. calculate backlog
Incon =: backlog <: b         NB. no congestion
Icon =: -.@Incon              NB. congestion
qnext =: b <. backlog
```

For each iteration of the congestion control algorithm, we compute a *stair* function $n_{\text{ack}}(t) = \lceil t/d_{RTT} \rceil$. When $n_{\text{ack}}(t+1) > n_{\text{ack}}(t)$, a round-trip time has expired and an acknowledgement has been received (or at least should have been). The J function *acknext* computes $n_{\text{ack}}(t+1)$. The indicator function *Iack* signals that an acknowledgement has arrived *or* should have arrived. We use this signal to defer the change in window size, whether it is an increase due to the receipt of an acknowledgement or a decrease as a result of congestion. The indicator function *Inack* signals that an acknowledgement is pending:

```
acknext =: ceil@(t%RTT)    NB. next acknowledgement
tnext =: >:@t              NB. increment t
Inak =: ack=acknext        NB. waiting for ack
Iack =: -.@Inak            NB. ack received
```

The verbs *wi* and *wd* implement the window increase and decrease functions $w_j^{\triangle}(t)$ and $w_j^{\triangledown}(t)$, respectively. We use the ceiling function to ensure that the window sizes are an integer number of segments:

```
wi =: ceil@(w + alpha % w ^ k) NB. window increase
wd =: ceil@(w - beta * w ^ l)  NB. window decrease
```

The window is increased if there has been no congestion ($I_{\overline{\text{con}}} = 1$), though we defer the change until $I_{\text{ack}} = 1$ (implying an acknowledgement). The window is decreased if there has been congestion ($I_{\text{con}} = 1$), and again we use the $I_{\text{ack}} = 1$ condition to defer the change:

```
h1 =: (w * Inak) + wi * Iack
h2 =: (w * Inak) + wd * Iack
h3 =: (h1 * Incon) + h2 * Icon
```

In the event of congestion, $I_{\text{ack}} = 1$ signifies an acknowledgement *anniversary*. Furthermore, a flow only decreases its window when $I_{\text{con}} = 1$ *and* $I_{\text{ack}} = 1$. This means that a flow will not always *detect* a congestion condition. This is not entirely unrealistic as flows receive congestion feedback from the network when their packets are dropped from the buffer queues of the router. In the event of congestion, not all flows will necessarily have their packets dropped. Also, flows with large round-trip times will detect congestion events less frequently than flows with shorter round trip times, and will thus respond less rapidly.

We ensure that the window size is at least one segment. The verb h_4 then returns the boxed results of the next state of the system:

```
   wnext =: max1@h3  NB. window is at least 1 segment
   h4 =: tnext ; wnext ; qnext ; acknext ; txnext
   cocurrent <'base'
```

Finally, we define *tcpf* which simply equates to h_4:

```
   tcpf_z_ =: h4_TCPC_
```

9.2.1 Analysis

We define the matrix P_1 and the vector of boxed elements X_0 to represent the static parameters and initial conditions, respectively. We define the vector $d_{RTT} = \{d_{RTT_j}, j = 0, 1, \ldots\}$, where $d_{RTT_j} = j + 1$:

```
   RTT =: >:@i.10
```

The other system parameters are the same for all flows: $c = 10$, $b = 5$, $\alpha = 1$, $\beta = 0.5$ $k = 0$ and $l = 1$; thus P_1 defines the parameters for ten AIMD flows:

```
   hd1=:'RTT';'c';'b';'alpha';'beta';'k';'l'
   P1 =: RTT,(10 $ &> 10 5 1 0.5 0 1)
   hd1,.  ;/ P1
+-----+---------------------------------------+
|RTT  |1 2 3 4 5 6 7 8 9 10                   |
+-----+---------------------------------------+
|c    |10 10 10 10 10 10 10 10 10 10          |
+-----+---------------------------------------+
|b    |5 5 5 5 5 5 5 5 5 5                    |
+-----+---------------------------------------+
|alpha|1 1 1 1 1 1 1 1 1 1                    |
+-----+---------------------------------------+
|beta |0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5|
+-----+---------------------------------------+
|k    |0 0 0 0 0 0 0 0 0 0                    |
+-----+---------------------------------------+
|l    |1 1 1 1 1 1 1 1 1 1                    |
+-----+---------------------------------------+
```

For time $t = 0$, the state conditions are initialised $w_j(0) = 1$, $q(0) = 0$, $n_{\text{ack}} = 1$ and $r_j(t) = 0$ for all j:

```
   X0 =: 0;(;/ 10 $ &> 1 0 1 0)
   hd2=:'t';'w';'q';'ack';'r'
   hd2,. X0
+---+-------------------+
|t  |0                  |
+---+-------------------+
|w  |1 1 1 1 1 1 1 1 1 1|
+---+-------------------+
|q  |0 0 0 0 0 0 0 0 0 0|
+---+-------------------+
|ack|1 1 1 1 1 1 1 1 1 1|
+---+-------------------+
|r  |0 0 0 0 0 0 0 0 0 0|
+---+-------------------+
```

The J expression below returns the state of the system for the first 200 iterations of the AIMD algorithm:

```
    S1 =: P1 tcpf^:(i.200) X0
```

We can extract individual state variables; for instance the transmission rate, with:

```
    tput1 =: tx_TCPC_"1 S1
```

The graph in Fig 9.2 shows the evolution of the transmission rate state variables for each flow (*tput1*). We can calculate the mean and peak transmission rate for each flow. The top plane is the mean transmission rate and the bottom plane is the peak transmission rate:

```
   load 'stats.ijs' NB. load stats functions
   hd3 =: 'mean';'max'
   hd3,. ;/ 2 2 5 $ (mean, max) tput1
+----+--------------------------------------+
|mean|    3.38 1.4075  1.85167  0.89125  0.556|
|    |0.795833  0.305 0.344375 0.490556 0.2445|
+----+--------------------------------------+
|max |8      3.5    5 1.75 1.4              |
|    |2 0.714286 0.75    1 0.6              |
+----+--------------------------------------+
```

We can compute the aggregate throughput on the link by summing across the flows. We can then derive the mean and peak aggregate throughput:

```
   agg1 =: +/"1 tput1
   hd3,: (mean;max) agg1
+-------+-------+
|mean   |max    |
+-------+-------+
|10.2667|14.6413|
+-------+-------+
```

The aggregate throughput represents the *offered* load to the communications link, which at times exceeds the capacity of the link. Traffic in excess of the capacity $c = 10$ is buffered, while buffered traffic in excess of the maximum queue size $b = 5$, is dropped. The graph in Fig 9.3 shows the aggregate throughput (offered load) for the AIMD algorithm. The backlog state variable is also of interest as it gives us some insight into the queue dynamics. Not surprisingly, the maximum backlog is five:

```
   q1 =: {."1 q_TCPC_"1 S1
   hd3,: (mean; max) q1
+-------+---+
|mean   |max|
+-------+---+
|3.24953|5  |
+-------+---+
```

For the MIMD algorithm, we set $k = -1$ and $l = 1$. The parameter matrix P_2 is defined as:

```
   P2 =: RTT,(10 $ &> 10 5 1 0.5 _1 1)
```

The evolution of the state variables for 200 iterations of the MIMD algorithm is given below:

```
   S2 =: P2 tcpf^:(i.200) X0
```

The per-flow and aggregate flow throughput are, as with the AIMD algorithm, derived by:

```
   tput2 =: tx_TCPC_"1 S2 NB. per flow throughput
   agg2 =: +/"1 tput2     NB. aggregate throughput
```

The mean and peak of the aggregate throughput are given by:

```
   hd3,: (mean;max) agg2
+-------+-------+
|mean   |max    |
+-------+-------+
|11.5196|21.5246|
+-------+-------+
```

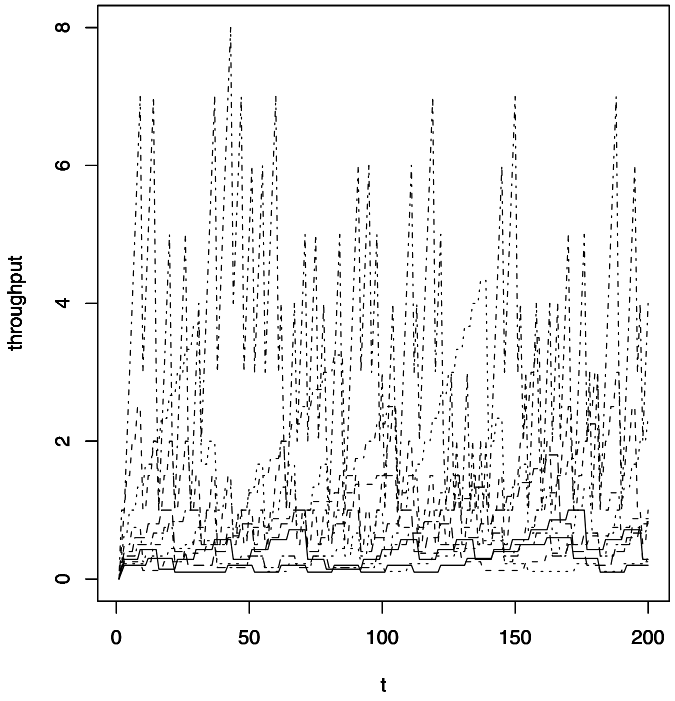

Fig. 9.2. Ten AIMD flows

The graph in Fig 9.4 shows the aggregate throughput for the MIMD algorithm. It can be seen that the aggregate throughput peaks are very high. Offered loads like this will fill the buffer queues quickly, resulting in high levels of traffic loss. It can be seen that the average queue occupancy for the MIMD algorithm is far higher than for AIMD and quite near to the peak:

```
   q2 =: {."1 q_TCPC_"1 S2
   hd3,: (mean;max) q2
+-------+---+
|mean   |max|
+-------+---+
|4.01483|5  |
+-------+---+
```

For the AIAD algorithm, we set $k = 1$ and $l = 0$. The parameter matrix P_3 is, therefore:

```
   P3 =: RTT,(10 $ &> 10 5 1 0.5 0 0)
```

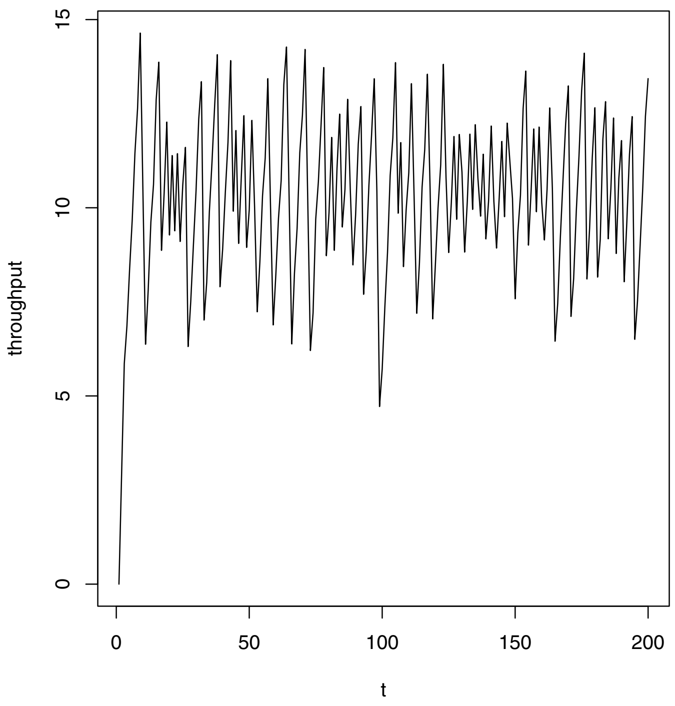

Fig. 9.3. Aggregate AIMD flow

We merely have to repeat the procedure above, using P_3 to derive the AIAD per-flow (*tput3*) and aggregate throughput (*agg3*):

```
   S3 =: P3 tcpf^:(i.200) X0
   tput3 =: tx_TCPC_"1 S3
   agg3 =: +/"1 tput3
```

We summarise the AIMD, MIMD and AIAD results below:

```
   hd4 =: '';'AIMD';'MIMD';'AIAD'
   (mean;max) agg1,.agg2,.agg3
+----+-------+-------+-------+
|    |AIMD   |MIMD   |AIAD   |
+----+-------+-------+-------+
|mean|10.2667|11.5196|10.4042|
+----+-------+-------+-------+
|max |14.6413|21.5246|14.6413|
+----+-------+-------+-------+
```

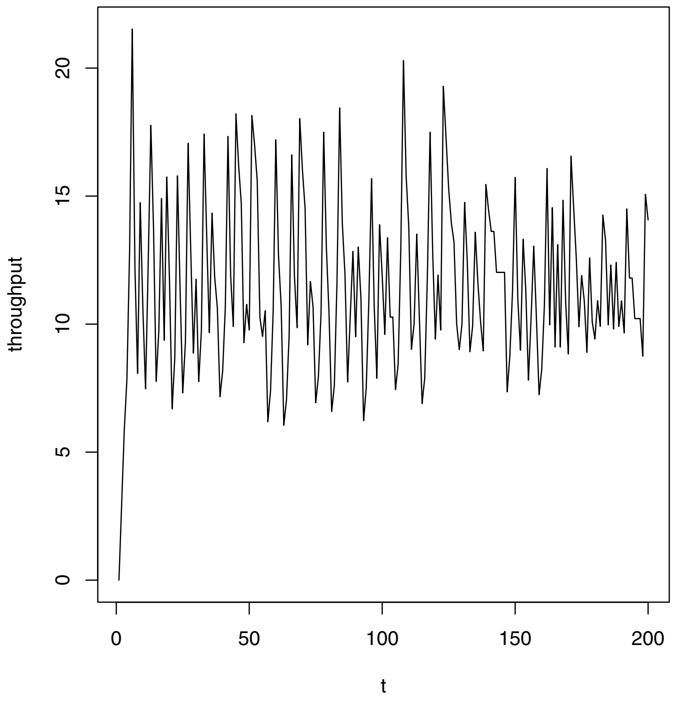

Fig. 9.4. Aggregate MIMD flow

9.3 Model of TCP Congestion Control

There is no single standardised congestion control algorithm for TCP. For one, there are a number of TCP implementations (for example: Tahoe, Reno NewReno, Vegas, Hybla, BIC, SACK and Westwood), each with a different congestion control algorithm. Furthermore, there are variations within each implementation (which can depend on thdepend on the operating system). Tahoe maintains a congestion window that is increased multiplicatively or additively, depending upon whether the session is in the *slow-start* or *congestion avoidance* phase. In the event of congestion, Tahoe reduces its congestion window to one segment. For congestion events triggered by a timeout, both reduce their congestion window to one. In this section, we build a model of TCP congestion control algorithm (loosely) based on Tahoe, as described by [63].

TCP has two congestion window increase phases: *slow-start* and *congestion avoidance*. A TCP session starts in the slow-start phase, setting the congestion window to a small value (one segment). The session sends one segment of data. Upon receiving an acknowledgement, it increases the window size by one and transmits data up to the new window size (now two segments). It continues to do this until the

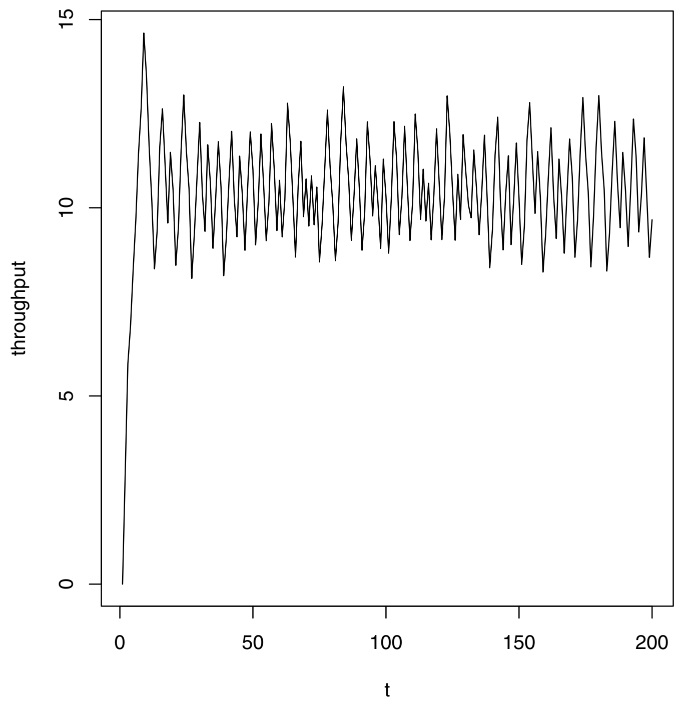

Fig. 9.5. Aggregate AIAD flow

window exceeds the slow-start threshold, whereupon it enters congestion avoidance. In congestion avoidance the window increases every d_{RTT_j}. Congestion avoidance represents an additive increase, whereas slow-start is multiplicative increase. When congestion occurs, TCP reduces its window back to one segment and initiates slow-start. However, it also sets the slow-start threshold to half the current window size. The function $\omega_j(t)$ gives the TCP congestion window size:

$$\omega_j(t + d_{RTT}) = \max[\omega_{\text{flow}}, \omega_j^{\triangle}(t) I_{\overline{\text{con}}} + \omega_j^{\triangledown}(t) I_{con}] \tag{9.11}$$

where $\omega_j^{\triangle}(t)$ is the window size increase upon receiving an acknowledgement, and $\omega_j^{\triangledown}(t)$ is the window size decrease in response to congestion. With the TCP model (unlike the TCP-compatible model), the transmission rate is *capped* by the receiver's flow control window ω_{flow}. The respective expressions for $\omega_j^{\triangle}(t)$ and $\omega_j^{\triangledown}(t)$ are:

$$\begin{array}{ll} \omega_j^{\triangle}(t) = \omega_j(t) + \alpha/\omega_j(t)^k & \text{no congestion} \\ \omega_j^{\triangledown}(t) = 1 & \text{congestion} \end{array} \tag{9.12}$$

If the current window size exceeds the slow-start threshold, the TCP congestion window undergoes a multiplicative increase; otherwise it undergoes an additive increase.

The slow-start and congestion avoidance stages are defined by the value of k:

$$k = \begin{cases} 0 & \omega_j(t) \leq \delta_{ssthresh_j}(t) \quad \text{(slow-start)} \\ -1 & \omega_j(t) > \delta_{ssthresh_j}(t) \quad \text{(congestion avoidance)} \end{cases} \tag{9.13}$$

where $\delta_{ssthresh_j}$ is the value of the slow-start threshold for flow j. The slow-start threshold is set to half of the current window size, if congestion occurs:

$$\delta_{ssthresh_j}(t + d_{RTT_j}) = \delta_{ssthresh_j}(t) I_{\overline{con}} + \omega_j(t) I_{con}/2 \tag{9.14}$$

The parameters to this function are the link capacity c, the buffer size b, α, the round-trip time d_{RTT} (*RTT*) and the receiver flow control window is w_{flow} (*flow*).

We do not need the window decrease parameters (β and l) as $\omega_j^{\bigtriangledown}(t) = 1$. However, if we wished to implement a Reno version of the TCP congestion control algorithm, we would need to reinstate them. The window increase parameter k is "hardcoded," and is either 1 or 0 depending upon whether it is in slow-start or congestion avoidance phase. We can (re)use some of the functions from the binary congestion control algorithms verbs; thus we only have to define one new parameter (*flow*) and one new state variable (*ssthresh*):

```
cocurrent <'TCP'
flow =: lhs6
ssthresh =: >@rhs6
```

We compute k based upon the current window size relative to the slow-start threshold, and calculate the window increase and decrease:

```
k  =: -@(w_TCPC_ <: ssthresh) NB. slow-start?
wi =: ceil@(w + alpha % w ^ k)
wd =: 1:
```

If the current window size is less than or equal to the slow-start threshold, then $k = 1$ and the window is increased multiplicatively as per slow-start. Otherwise $k = 0$ and the window increase is additive as per congestion avoidance. Note that we will still pass the β, even though *wd* does not use it. We continue to pass β for two reasons. The first reason is for convenience; the TCP congestion control verb would need some significant redevelopment, if it was removed. The other reason is that, for some TCP implementations, such as Reno, when three duplicate acks are received, instead of setting the congestion control, window to one segment, it is multiplicatively decreased (typically it is halved) and *fast recovery* is initiated. With fast recovery, the algorithm goes straight into congestion avoidance rather than slow-start. If we were to model this behaviour of TCP congestion control we would need the parameter β.

If an acknowledgement has been received, we recalculate the window according to congestion conditions of the network:

```
h1 =: (w_TCPC_ * Inak_TCPC_) + wi * Iack_TCPC_
h2 =: (w_TCPC_ * Inak_TCPC_) + wd * Iack_TCPC_
wnext =: (h1 * Incon_TCPC_) + h2 * Icon_TCPC_
```

If there has been congestion, the slow-start threshold is set to half the current window size (using the `-:` primitive). To ensure that it is an integer value and at least one, *ceil* and *max1* are applied, respectively:

```
g1 =: ceil@-:@ wnext
g2 =: max1@g1
g3 =: (ssthresh * Inak_TCPC_) + g2 * Iack_TCPC_
ssthnext =: (ssthresh*Incon_TCPC_) + g3 * Icon_TCPC_
```

In addition to the congestion window advertised by the sender, the TCP receiver advertises a flow-control window. We, therefore, cap the transmission rate, if the congestion window exceeds the flow-control window:

```
h4 =: (flow%RTT_TCPC_) ,: txnext_TCPC_
txnext =: min"2@h4   NB. apply minimum cwnd
```

Next, we prepare to output the results of the current iteration to be fed back into the next iteration of the system. We could do this in one step, although we separate it into multiple lines for brevity:

```
h5 =: tnext_TCPC_ ; wnext ; qnext_TCPC_
h6 =: acknext_TCPC_ ; txnext_TCPC_ ; ssthnext
h7 =: h5,h6
cocurrent <'base'
```

Finally, we define *tcp*:

```
tcp_z_ =: h7_TCP_
```

9.3.1 Analysis

If we consider only one flow, we can examine the slow-start and congestion avoidance behaviour of TCP:

```
   (tx_TCPC_"1) 1 10 5 1 0.5 12 tcp^:(i.10) 0;1;0;1;0;6
0 1 2 4 8 9 10 11 12 13
```

We can run the TCP congestion control algorithm for ten flows by setting the parameter matrix P_4 and initial starting conditions Y_0:

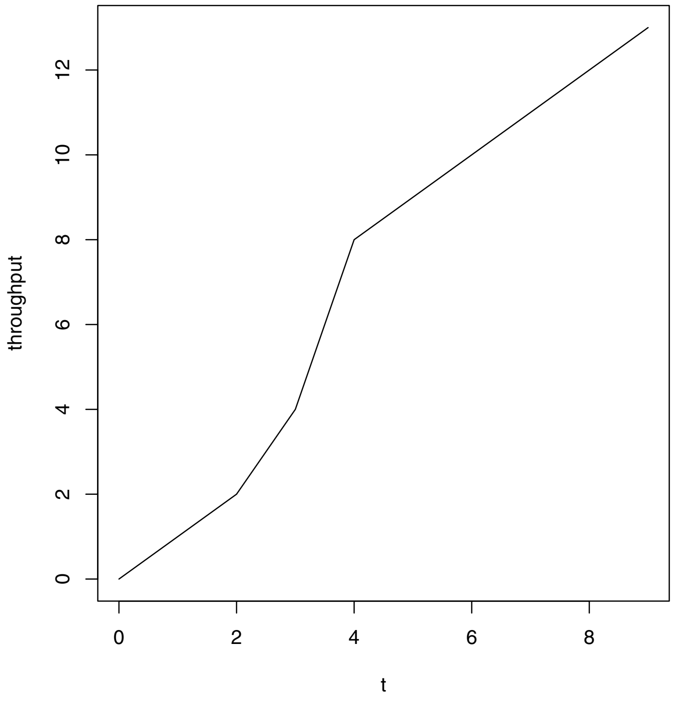

Fig. 9.6. The transition of a TCP flow from the slow-start phase (multiplicative increase) to the congestion avoidance phase (additive increase)

```
P4 =: RTT , (10 $ &> 10 5 1 0.5 12)
Y0 =:  0 ; (;/ 10 $ &> 1 0 1 0 7)
```

Then, we compute the aggregate throughput (offered load):

```
S4 =: P4 tcp^:(i.200) Y0
tput4 =: tx_TCPC_"1 S4
agg4 =: +/"1 tput4
```

The graph in Fig 9.7 shows the aggregate throughput (*agg4*) for TCP.

Here we analyse the second flow (the flow for which $d_{RTT_2} = 2$) using the network calculus methods from Chapter 4. The effective bandwidth algorithm was given in Equation (4.39). We can, therefore, compute the effective bandwidth of the an individual TCP flow by deriving the cumulative arrivals for the flow and expressing them as a function A:

```
X =: +/\ rhs1"1 tput4
A =: X&seq
```

The effective bandwidth is then given by:

```
   load 'netcalc.ijs'
   delay =: (i.20)%5
   D =: [ NB. pass delay bound as left argument
   4 5 $ delay max@((A@t - A@s) % ((t-s) + D)) &> 199
      7 5.90909 5.41667       5 4.73684
    4.5 4.28571 4.09091 3.92857  3.7931
3.66667 3.54839 3.50123 3.49265 3.48411
3.47561 3.46715 3.45874 3.45036 3.44203
```

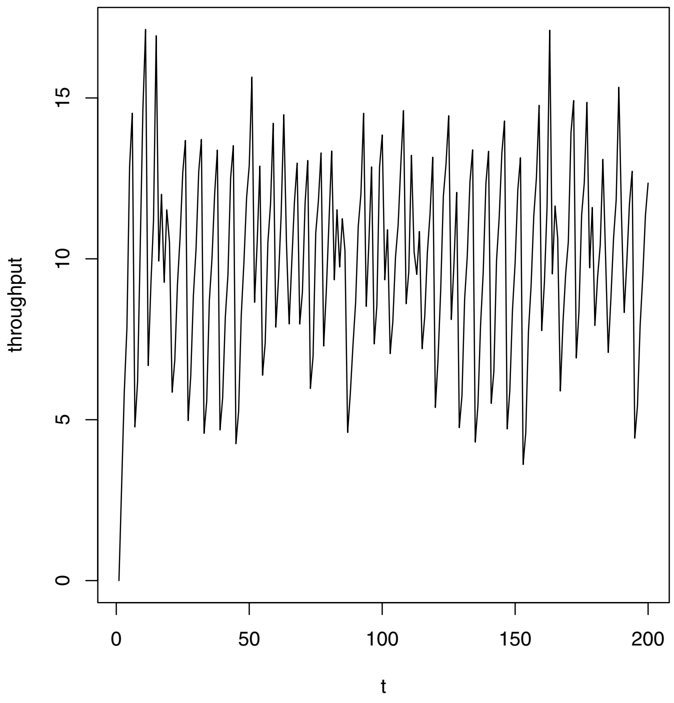

Fig. 9.7. Aggregate TCP flows

9.4 Summary

In this chapter we used J to develop a number of dynamical feedback systems for implementing TCP and TCP-compatible congestion control algorithms. This constitutes a *closed-loop* model of network traffic. We assume all flows are greedy and transmit up to their maximum window size. Binomial congestion control algorithms represent a family of algorithms. The nature of each algorithm is defined by the parameters k and l, where k determines how aggressively the network is probed for capacity and l determines how conservative the response to congestion. We can also use parameters k and l to control when TCP is in slow-start and when it is in congestion avoidance.

What is interesting about the results from these dynamical feedback systems is that, while they are deterministic, they yield traffic patterns that appear stochastic.

A

Scripts

Create a startup script `user/config/startup.ijs` under the directory where you installed J, (remember to reverse `/` instead `\` if you are using Windows). Any commands you place in this file will be executed when J starts up. You can create and edit this file from the J GUI. Select “Edit” and then “Configure” from the drop down menu. In the categories section select “startup.” You create and edit `startup.ijs`.

A.1 Scripts from Chapter 3

Listing A.1 *Common Functions: libs.ijs*

```
NB. Network Performance Analysis
NB.    by Alan Holt
NB.
NB. Chapter 3: Advanced Programming
NB. Common functions
NB.
NB. Execute script with the command line:
NB.    load 'libs.ijs'
NB.

cocurrent < 'z'
max =: >./     NB. max(x)
max0 =: 0&max  NB. max(0,x)
max1 =: 1&max  NB. max(1,x)
min =: <./     NB. min(x)
ceil  =: <.
floor =: >.
```

```
ip =: +/ .*     NB. inner product

ge =: >+.=  NB. greater than or equal to
le =: <+.=  NB. less than or equal to

log10 =: 10&^. NB. log10
log2 =: 2&^.   NB. log2
ln =: [:^.]    NB. natural log

col =: ,."2 NB. "columnize" list

for =: @i.
ei =: (}:@(-@i.@-)),i. NB. -(x-1)..0..(x-1)

rsort =: {~ ?~@#     NB. random sort
cocurrent < 'base'

cocurrent < 'range'
f1 =: -.&i.~ , ]        NB. +lhs to +rhs
NB. f2 =: -@(}:@i.@<:@[) , >:@i.@]
f2 =: -@(i.@<:@[) ,>:@i.@]
select =: 0&>@[         NB. is left arg -ve?
cocurrent < 'base'

to_z_ =: f1_range_ ` f2_range_ @. select_range_

not_z_ =: -.

NB. Positional parameters
cocurrent < 'z'
lhs0 =:  [        NB. all left arguments
lhs1 =:  0&{@lhs0 NB.  1st left argument
lhs2 =:  1&{@lhs0 NB.  2nd left argument
lhs3 =:  2&{@lhs0 NB.  3rd left argument
lhs4 =:  3&{@lhs0 NB.  4rd left argument
lhs5 =:  4&{@lhs0 NB.  5th left argument
lhs6 =:  5&{@lhs0 NB.  6th left argument
lhs7 =:  6&{@lhs0 NB.  7th left argument
lhs8 =:  7&{@lhs0 NB.  8th left argument
lhs9 =:  8&{@lhs0 NB.  9th left argument
lhs10 =: 9&{@lhs0 NB. 10th left argument
rhs0  =: ]        NB. all right arguments
rhs1  =: 0&{@rhs0 NB.  1st right argument
rhs2  =: 1&{@rhs0 NB.  2nd right argument
rhs3  =: 2&{@rhs0 NB.  3rd right argument
```

```
rhs4  =: 3&{@rhs0 NB.  4th right argument
rhs5  =: 4&{@rhs0 NB.  5th right argument
rhs6  =: 5&{@rhs0 NB.  6th right argument
rhs7  =: 6&{@rhs0 NB.  7th right argument
rhs8  =: 7&{@rhs0 NB.  8th right argument
rhs9  =: 8&{@rhs0 NB.  9th right argument
rhs10 =: 9&{@rhs0 NB. 10th right argument
cocurrent < 'base'
```

A.2 Scripts from Chapter 4

Listing A.2 *Network Calculus Functions, Script: netcalc.ijs*

```
NB. Network Performance Analysis
NB.    by Alan Holt
NB.
NB. Chapter 4: Network Calculus
NB.
NB. Execute script with the command line:
NB.    load 'netcalc.ijs'
NB.

NB. Prerequisites
load 'libs.ijs'

seq_z_ =: {~ NB. sequence

NB. range of values of the interval [0,1]
i01 =: %(>./@:|)


NB. 'time' index verbs
t_z_ =: ]
s_z_ =: i.@>:@t
v_z_ =: }.@i.@t
u_z_ =: i.@>:@[

plus =: min"0 NB. point-wise minimum

NB. Wide-sense Increasing Functions
F0_z_ =: ([: 0&<rhs0)
```

```
NB. peak rate function
pr_z_ =: F0 * *

NB. affine function
af_z_ =: F0 * lhs2 + lhs1 * rhs0

NB. burst delay function
bd_z_ =: F0*_:*<

NB. rate latency function
rl_z_ =: lhs1 * max0 @ (rhs0-lhs2)

NB. step function
step_z_ =: 0:`1:@.<

NB. stair function
stair_z_ =: F0 * ceil @ (lhs1 %~ lhs2 + rhs0)

NB. subadditive closure
cocurrent < 'z'
stop =: 2&<.
conv =: min@(f, close"0@v + close"0@(t-v))
close =: (0: ` f ` conv @. stop)"0
cocurrent < 'base'

NB. Lindley equation
cocurrent < 'LND'
qlist =: rhs0
c     =: >@lhs1
alist =: >@lhs2
ind   =: <:@#@qlist

anext =: ind { alist
qprev =: {:@qlist
qnext =: max0 @ (anext + qprev - c)
cocurrent <'base'
lindley_z_ =: qlist_LND_,qnext_LND_

NB. Lindley equation (limited buffer)
cocurrent < 'LND2'
qlist =: rhs0
c     =: >@lhs1
q     =: >@lhs2
alist =: >@lhs3
ind   =: <:@#@qlist
```

```
anext =: ind { alist
qprev =: {:@qlist
qnext =: min@(max0 @ anext + qprev - c,q)
cocurrent <'base'
lindley2_z_ =: qlist_LND2_,qnext_LND2_

NB. g-clipper
cocurrent < 'CLIPPER'
Blist =: rhs0
t =: #@Blist
s =: i.@t
Alist =: lhs0
ind   =: <:@#@Blist
Bprev =: {:@Blist
Aprev =: ind { Alist
Anext =: >:@ind { Alist
f1 =: min@(Blist + g@(t-s))
f2 =: Bprev + A@t - A@(<:@t)
Bnext =: min@(f1,f2)
cocurrent <'base'
gclipper_z_ =: Blist_CLIPPER_,Bnext_CLIPPER_
```

A.3 Scripts from Chapter 5

Listing A.3 *Statistics functions, script: stats.ijs*

```
NB. Network Performance Analysis
NB.    by Alan Holt
NB.
NB. Chapter 5: Statistical Methods
NB.            and Stochastic Processes

NB. Execute script with the command line:
NB.    load 'stats.ijs'
NB.

NB. Prerequisites
load 'libs.ijs'

cocurrent < 'z'
mean  =: [: (+/%#) ]        NB. arithmetic mean
mdev  =: -mean              NB. deviations from the mean
```

```
sqdev =: [: (*:@mdev) ]      NB. square of deviations
sumsq =: [: (+/@sqdev) ]     NB. sum of squares
dof   =: [: <:@# ]           NB. degrees of freedom
var   =: [: (sumsq % dof) ] NB. variance
std   =: [: (%: @ var) ]     NB. standard deviation

sp  =: [: +/ *&mdev
ssp =: [: +/ @ (*&mdev)~
cov =: sp % dof
cor =: cov % (*&std)
cocurrent < 'base'

NB. Autocovariance and Autocorrelation
cocurrent < 'ACF'
COV =: 0
COR =: 1
f1 =: }. ,: -@[}. ]   NB. compose data
f2 =: (f1-mean)       NB. deviation from mean
n =: #@]
sp =: +/@(*/@f2)
cocurrent < 'base'
autocov_z_ =: sp_ACF_% n_ACF_
autocor_z_ =: sp_ACF_ % sumsq

NB. Long-range Dependence

NB. Calculate Hurst parameters
hurst_z_ =: >:@-@-: NB. time domain
hurst2_z_ =: -:@>:  NB. frequency domain

NB. Variance time plot

cocurrent < 'VARM'
Xt =: rhs0
m =: lhs1

g1 =: (#@Xt)
g2 =: <.@(g1%m)
g3 =: g2*m
g4 =: (i.@g3) { Xt

f1 =: (-@m) +/\ g4
f2 =: f1%m
f3 =: var"1@f2
```

```
cocurrent < 'base'
varm_z_ =: f3_VARM_

cocurrent < 'DFT'
h =: rhs0
k =: lhs1
n =: #@h
j =: i.@n  NB. j=1,2,..,n-1
eu =: ^@j. NB. Euler's formula
f1 =: -@(+:@o.) @ (j*k%n) NB. -2pi jk/n
f2 =: h * (eu@f1)          NB. h*exp(-2pi jk/n)
f3 =: +/@f2
cocurrent < 'z'
dft   =: f3_DFT_     NB. in complex form
dftmp =: *.@dft      NB. phase and magnitude
dftm  =: {."1 @ dft  NB. magnitude only
dftp  =: {:"1 @ dft  NB. phase only
cocurrent < 'base'


NB. RANDOM NUMBERS

cocurrent < 'z'
NB. uniform deviates
rand =: [: (?@$&2147483646) ]
NB. runif =: 2147483647& (%~) @ (?@$&2147483646)
runif =: 2147483647& (%~) @ rand
cocurrent < 'base'

cocurrent < 'RNORM'
c1 =: [: _4.24264&+ ]
c2 =: [: 1.41412&* ]
f1 =: [:6&*]
f2 =: runif@f1
g1 =: _6: +/\ ]
g2 =: c1@c2@g1@f2
cocurrent < 'base'
rnorm_z_ =: g2_RNORM_

NB. Arbitrary mean and std
rnorm2 =: +`(*rn1)/ NB. rnorm2 10 5 5

NB. negative exponential
rexp_z_ =: (-@^.)@runif    NB. mean 1
exprand_z_ =: lhs1*[:rexp] NB. arbitrary mean
NB. Geometric
cocurrent <'RGEO'
p =: %@lhs1
n =: rhs1
rate =: %@p
NB. f1 =: rate (exprand) n
f1 =: rexp@n
f2 =: ^.@(1:-p)
f3 =: -@f1%f2
f4 =: ceil@>:@f3
cocurrent <'base'
rgeo_z_ =: f4_RGEO_

NB. roll dice
dice_z_ =: >:@?@]#&6

NB. Bernoulli (0/1)
rber_z_ =: >runif

NB. Pareto

cocurrent < 'PAR'
alpha =: lhs1
beta  =: lhs2
n =: rhs1

f1 =: runif@n
f2 =: %@>:@-@f1
f3 =: %@alpha
f4 =: beta*(f2^f3)
cocurrent < 'base'
rpar_z_ =: f4_PAR_

mpar_z_ =: *%<:@]
```

Listing A.4 *Functions for Stochastic Processes, Script: stochastic.ijs*

```
NB. Network Performance Analysis
NB.    by Alan Holt
NB.
NB. Chapter 5: Statistical Methods
NB.            and Stochastic Processes

NB. Execute script with the command line:
```

```
NB.    load 'stochastic.ijs'
NB.

NB. Prerequisites
load 'libs.ijs'
load 'stats.ijs'

NB. Hurst parameter
hurst_z_  =: >:@-@-: NB. time domain
hurst2_z_ =: -:@>:   NB. frequency domain

NB. Variance-time plot
cocurrent < 'VARM'
Xt =: rhs0
m =: lhs1

g1 =: (#@Xt)
g2 =: <.@(g1%m)
g3 =: g2*m
g4 =: (i.@g3) { Xt

f1 =: (-@m) +/\ g4
f2 =: f1%m
f3 =: var"1@f2

cocurrent < 'base'
varm_z_ =: f3_VARM_

NB. Discrete Fourier Transform
cocurrent < 'DFT'
h =: rhs0
k =: lhs1
n =: #@h
j =: i.@n   NB. j=1,2,..,n-1
eu =: ^@j.  NB. Euler's formula
f1 =: -@(+:@o.) @ (j*k%n) NB. -2pi jk/n
f2 =: h * (eu@f1)      NB. h*exp(-2pi jk/n)
f3 =: (+/@f2)%n
cocurrent < 'z'
dft1      =: f3_DFT_     NB. in complex form
dft2      =: *.@dft1     NB. phase and magnitude
dft       =: {."1 @ dft2 NB. magnitude only
dftphase  =: {:"1 @ dft2 NB. phase only
cocurrent < 'base'
```

```
NB. Autoregressive process
cocurrent <'AR'
et    =: rhs1
coefs =: rhs2

g1 =: coefs ip et
cnext =: 0:,}:@coefs
f1 =: *@ coefs
f2 =: |.@f1
f3 =: +/\@f2
mask1 =: |.@(*@f3)
mask2 =: not@mask1
f5 =: <:@(+/@mask1)
f6 =: +/@mask2
f7 =: f5 {. et      NB. -p to (i-1) elements
f8 =: (-@f6) {. et  NB.(i+1) to n elements
xnext =: f7,g1,f8   NB. insert u(i)
cocurrent < 'base'
ar_z_ =: xnext_AR_,:cnext_AR_

NB. Moving average

sma_z_ =: +/\%[ NB. simple MA

cocurrent < 'MA'
f1 =: (#@lhs0) +\ rhs0
f2 =: f1 ip [
cocurrent < 'base'
wma_z_ =: f2_MA_ NB. Weighted MA

NB. Fractional ARIMA process

NB. Coefficients
cocurrent < 'ARIMA'
d =: lhs1
n =: rhs0
f1 =: */\@ (i.@n-d) NB. -d, -d(1-d), -d(1-d)(2-d) ...
f2 =: 1:, -@f1  NB. 1, d, d(1-d), d(1-d)(2-d) ...
f3 =: !@i.@>:@n NB. 0!, 1!, 2! ...
f4 =: f2%f3
cocurrent < 'base'
fdcoefs_z_ =: f4_ARIMA_

NB. Fractional differencing function
cocurrent < 'ARIMA'
```

```
c =: lhs0
nc =: #@c
en =: [: >@{. ]
xt =: [: >@{: ]
xn =: xt {~i.@ <:@nc
et =: {.@en
enext =: }.@en
g1 =: c*(et,xn)
g2 =: +/@g1
g3 =: enext;(g2,xt)
cocurrent < 'base'
fdiff_z_ =: g3_ARIMA_

NB. Fractional differencing wrapper script
cocurrent < 'z'
fd=: 4 : 0
'nc nx' =: y.
d =: x.
c =: d fdcoefs nc
e1 =: rnorm nx
e2 =: rnorm nc
|. (i.nx) { >{: (c fdiff^:(nx) e1;e2)
)
cocurrent < 'base'

NB. Traffic simulation script
ts =: 4 : 0
'nf lf' =. y.
d =. x.
a =. 0
for_j. i. nf
do. a =. a + 0> d  fd 170&,lf
end.
a
)
```

A.4 Scripts from Chapter 6

Listing A.5 *Traffic Modeling, Script: onoff.ijs*

```
NB. Network Performance Analysis
NB.    by Alan Holt
NB.
NB. Chapter 6: Traffic Modeling
NB. and Simulation
NB.
NB. Execute script with command line:
NB.    load 'onoff.ijs'
NB.

NB. Prerequisites
load 'libs.ijs'
load 'stats.ijs'

fdist_z_ =: [: +/"1 =/
Fdist_z_ =: [: +/"1 >/
pdf_z_ =: fdist@%[:#]

mm_z_ =: +/@:* NB. matrix multiply

phasediag_z_ =: }: ,: }.

cocurrent < 'BIN'
p =: lhs1
n =: rhs1
k =: i.@>:@n
f1 =: !@n
f2 =: (!@k) * (!@(n-k))
f3 =: f1%f2
f4 =: (p^k) * (1:-p)^(n-k)
f5 =: f3*f4
cocurrent < 'base'
bindist_z_ =: f5_BIN_

cocurrent < 'POIS'
x  =: rhs0
lm =: lhs1
f1 =: lm^x
f2 =: ^@(-@lm)
f3 =: !@x
f4 =: f1*f2%f3
cocurrent < 'base'
poisdist_z_ =: f4_POIS_

cocurrent <'EB'
s =: rhs0
x =: >@lhs1
p =: >@lhs2
```

```
f1 =: ^@(s */ x)
f2 =: f1 (*"1) p
f3 =: +/"1 @f2
f4 =: (^.@f3) % s
cocurrent <'base'
eb_z_ =: f4_EB_

cocurrent < 'MD1'
rho =: lhs1 % rhs0
f1 =: >:@-@rho
f2 =: +:@f1
f3 =: (*:@rho) % f2
f4 =: lm + f3
cocurrent < 'base'
Emd1 =: f4_MD1_

cocurrent < 'z'
mjoin =: ,&(i.@#)
msel =: /:@mjoin
merg    =: msel { ,
cocurrent < 'base'

rnd_z_ =: ceil @ (0.5&+@])

cocurrent < 'OOGEOGEO'
p01 =: lhs1
p10 =: lhs2
n  =: rhs1
f1 =: %@p01 rgeo n
f2 =: %@p10 rgeo n
f3 =: f1,.f2
f4 =: f3 <@# 1:,0:
cocurrent < 'base'
oosrd_z_ =: ;@f4_OOGEOGEO_

cocurrent < 'OOPARGEO'
p01 =: lhs1
alpha =: lhs2
n  =: rhs1
f1 =: %@p01 rgeo n
f2 =: (alpha,1:) (ceil@rpar) n
f3 =: f1,.f2
f4 =: f3 <@# 1:,0:
cocurrent < 'base'
oolrd_z_ =: ;@f4_OOPARGEO_
```

A.5 Scripts from Chapter 7

Listing A.6 *Chaotic Maps, Script: cmap.ijs*

```
NB. Network Performance Analysis
NB.    by Alan Holt
NB.
NB. Chapter 7: Chaotic Maps
NB.
NB. Execute script with the command line:
NB.    load 'cmap.ijs'
NB.

NB. Prerequisites
load 'libs.ijs'

f_z_ =: 4&*@(* >:@-)
g_z_ =: lhs1 * [: f ]

diff_z_ =: 4 : 0
'v0 N a'  =. x. [ 'x0' =. rhs1 y.
'xn1 xn2' =. a g^:(N) x0, (x0+v0)
xn1, | xn1-xn2
)

lyap_z_ =: 4 : 0
'v0 N a R'  =. x. [ 'x0' =. y.
df =: rhs2"1 ((v0,N,a) diff^:(1 to R) x0)
(ln df%v0)%N
)

doall_z_ =: 3 : 0
L =: ''
for_a. y.
do.
  l =. mean (1e_6,20,a,100) lyap 0.1
  L =. L,l
end.
L
)

cocurrent < 'TENTMAP'
u =: lhs1
x =: rhs1
```

```
f1 =: u*x
f2 =: u*(>:@-@x)
select =: 0.5 &le@x
xnext =: f1 ` f2 @. select
cocurrent < 'base'
tmap_z_ =: xnext_TENTMAP_

NB. Logistic maps

NB. Bernoulli shift
cocurrent < 'BMAP'
d =: lhs1
x =: rhs1

f1 =: x%d
f2 =: (x-d)%(1:-d)
xnext =: f1 ` f2 @. (d<x)
NB. xnext =: (f1*I0) + (f2*I1)
cocurrent < 'base'
bshift_z_ =: xnext_BMAP_

NB. Double Intermittency map

cocurrent < 'DIMAP'
d  =: lhs1"1     NB. d
m  =: lhs2"1     NB. m
e1 =: lhs3"1     NB. e1
e2 =: lhs4"1     NB. e2
x  =: rhs0

c1 =: (>:@-@e1 - d) % d ^ m
c2 =: -@(e2 - d) % >:@-@d ^ m

select =: d<x

f1 =: e1 + x + c1 * x ^ m
f2 =: -@e2 + x - c2 * >:@-@x ^ m

I0 =: x le d
I1 =: d < x

xnext =: f1 ` f2 @. (d<x)
NB. xnext =: (I0*f1) + (I1*f2)
cmap =: xnext
```

```
cocurrent < 'z'

dimap =: xnext_DIMAP_
```

A.6 Scripts from Chapter 8

Listing A.7 *ATM Quality of Service, Script: atm.ijs*

```
NB. Network Performance Analysis
NB.    by Alan Holt
NB.
NB. Chapter 8: ATM Quality of Service
NB.
NB. Execute script with the command line:
NB.    load 'atm.ijs'
NB.

NB. Prerequisites
load 'libs.ijs'


NB. Interpacket arrivals
ipa_z_ =: }. - }:

NB. Get conforming cells
conform_z_ =: >@rhs2

NB. Burst tolerance
cocurrent < 'BT'
Is =: lhs1
Ip =: lhs2
mbs =: rhs0
f1 =: <:@mbs * Is-Ip
cocurrent < 'base'
burst_z_ =: f1_BT_

NB. Generic Cell Rate Algorithm
NB. common functions to VSA and LB
cocurrent < 'GCRA'
I =: lhs1
L =: lhs2
Is =: lhs1
Ip =: lhs2
```

```
Ls =: lhs3
Lp =: lhs4
ta    =: >@rhs1  NB. list of arrival times
clist =: >@rhs2  NB. conformance vector
ta1   =: {.@ta   NB. 1st element of ta list
tn    =: }.@ta   NB. tail of ta list
cocurrent < 'base'

NB. Virtual Scheduling Algorithm
cocurrent < 'VSA'
TAT    =: >@rhs3   NB. Theoretical Arrival Time
g1 =: TAT - L_GCRA_
g2 =: max @ (ta1_GCRA_, TAT) + I_GCRA_

conform =: ta1_GCRA_ < g1

f1 =: tn_GCRA_;(clist_GCRA_,0:);g2
f2 =: tn_GCRA_;(clist_GCRA_,1:);TAT
f3 =: f1 ` f2 @. conform
cocurrent < 'z'
vsa_z_ =: f3_VSA_

NB. Dual Virtual Scheduling Algorithm
cocurrent < 'DVSA'
TATs    =: >@rhs3  NB. Theoretical Arrival Time
TATp    =: >@rhs4  NB. Theoretical Arrival Time

g1 =: TATs -  Ls_GCRA_
g2 =: TATp -  Lp_GCRA_
g3 =: min @ (g1,g2)
conform =: ta1_GCRA_ < g3
TATsnext =: max @ (ta1_GCRA_, TATs) + Is_GCRA_
TATpnext =: max @ (ta1_GCRA_, TATp) + Ip_GCRA_

f1 =: tn_GCRA_;(clist_GCRA_,0:);TATsnext;TATpnext
f2 =: tn_GCRA_;(clist_GCRA_,1:);TATs;TATp
f3 =: f1 ` f2 @. conform
cocurrent < 'base'
NB.dualvsa_z_  =: f1_DVSA_ ` f2_DVSA_ @. conform_DVSA_
dvsa_z_  =: f3_DVSA_

NB. Leaky bucket
cocurrent < 'LB'
LCT   =: >@rhs3
B     =: >@rhs4
```

```
g1 =: B - (ta1_GCRA_ - LCT)
g2 =: max0@g1 + I_GCRA_
f1 =: tn_GCRA_;(clist_GCRA_,0:);ta1_GCRA_;g2
f2 =: tn_GCRA_;(clist_GCRA_,1:);LCT;B
conform =: g1 > L_GCRA_

f3 =: f1 ` f2 @. conform
cocurrent < 'base'
lb_z_   =: f3_LB_

NB. Dual Leaky Bucket
cocurrent < 'DLB'
LCT   =: >@rhs3
Bs    =: >@rhs4
Bp    =: >@rhs5

g1 =: -/@ (Bs,ta1_GCRA_,LCT)
g2 =: -/@ (Bp,ta1_GCRA_,LCT)
NB.g1 =: Bs - (ta1_GCRA_-LCT)
NB.g2 =: Bp - (ta1_GCRA_-LCT)

g3 =: max0@g1 + Is_GCRA_
g4 =: max0@g2 + Ip_GCRA_
conform =: (g1 > Ls_GCRA_) +. g2 > Lp_GCRA_

f1 =: tn_GCRA_;(clist_GCRA_,0:);ta1_GCRA_;g3;g4
f2 =: tn_GCRA_;(clist_GCRA_,1:);LCT;Bs;Bp

f3 =: f1 ` f2 @. conform
cocurrent < 'base'
duallb_z_  =: f1_DLB_ ` f2_DLB_ @. conform_DLB_
dlb_z_  =: f3_DLB_
cocurrent < 'base'
```

A.7 Scripts from Chapter 9

Listing A.8 *Internet Congestion Control, Script: congestion.ijs*

```
NB. Network Performance Analysis
NB.    by Alan Holt
NB.
NB. Chapter 9: Congestion Control
NB. Internet congestion control
NB.
NB. Execute script with the command line:
NB.    load 'congestion.ijs'
NB.

NB. Prerequisites
load 'libs.ijs'

NB. Simple congestion control algorithm
cocurrent <'CWND'
c     =: lhs1
alpha =: lhs2
beta  =: lhs3
w     =: rhs0
a     =: +/@w
Icon  =: a > c
Incon  =: -.@Icon
f1 =:  w + ((alpha * Incon) - (beta * Icon))
wnext =:  0.0001&max@(w + (alpha * Incon) - beta * Icon)
cocurrent <'base'
cwnd_z_ =: wnext_CWND_


NB. Binomial congestion control algorithms
cocurrent <'TCPC'
RTT   =: lhs1
c     =: lhs2
b     =: lhs3
alpha =: lhs4
beta  =: lhs5
k     =: lhs6
l     =: lhs7

t        =: >@rhs1
w        =: >@rhs2
q        =: >@rhs3
ack      =: >@rhs4
tx       =: >@rhs5

acknext =: ceil@(t%RTT)   NB. next acknowledgement
tnext =: >:@t             NB. increment t
Inak =: ack=acknext       NB. waiting for ack
Iack =: -.@Inak           NB. ack received

txnext =: w%RTT
a =: +/@txnext              NB. aggregate flow
```

```
backlog =: max0@(a + q - c) NB. calculate queue size
Incon =: backlog le b       NB. no congestion
Icon =: -.@Incon        NB. congestgion
qnext =: b <. backlog

wi =: ceil@(w + alpha % w ^ k)
wd =: ceil@(w - beta * w ^ l)

h1 =: (w * Inak) + wi * Iack
h2 =: (w * Inak) + wd * Iack
h3 =: (h1 * Incon) + h2 * Icon

wnext =: max1@h3 NB. ensure window at least 1 seg
h4 =: tnext; wnext ; qnext ; acknext ; txnext

cocurrent <'base'
tcpf_z_ =: h4_TCPC_


NB. TCP congestion control

cocurrent <'TCP'
flow =: lhs6
ssthresh =: >@rhs6

txnext =: w_TCPC_ % RTT_TCPC_

k =: -@(w_TCPC_ le ssthresh)
wi =: ceil@(w_TCPC_ + alpha_TCPC_ % w_TCPC_ ^ k)
wd =: 1:

h1 =: (w_TCPC_ * Inak_TCPC_) + wi * Iack_TCPC_
h2 =: (w_TCPC_ * Inak_TCPC_) + wd * Iack_TCPC_
wnext =: (h1 * Incon_TCPC_) + h2 * Icon_TCPC_

g1 =: ceil@-:@ w_TCPC_
g2 =: max1@g1
g3 =: (ssthresh * Inak_TCPC_) + (g2 * Iack_TCPC_)
ssthnext =: (ssthresh * Incon_TCPC_) + (g3 * Icon_TCPC_)

h4 =: (flow%RTT_TCPC_) ,: txnext_TCPC_
txnext =: min"2@h4   NB. apply minimum cwnd
```

```
h5 =: tnext_TCPC_ ; wnext ; qnext_TCPC_
h6 =: acknext_TCPC_ ; txnext_TCPC_ ; ssthnext
h7 =: h5,h6

cocurrent <'base'
tcp_z_ =: h7_TCP_
```

Abbreviations

ABR:	Available Bit Rate
ACF:	Autocorrelation Function
AIAD:	Additive Increase Additive Decrease
AIMD:	Additive Increase Multiplicative Decrease
AR:	Autoregressive
ARMA:	Autoregressive Moving Average
ARIMA:	Autoregressive Integrated Moving Average
ATM:	Asyncronous Transmission Mode
BT:	Burst Tolerance
CBR:	Constant Bit Rate
CDVT:	Cell Delay Variation Tolerance
DiffServ:	Differentiated Services
DSCP:	Differentiated Services Code Point
DCCP:	Datagram Congestion Control Protocol
FCFS:	First-Come First-Served
GCRA:	Generic Cell Rate Algorithm
IntServ:	Integrated Services
IP:	Internet Protocol
ISP:	Internet Service Provider
LBE:	Less-than Best Effort
LCT:	Last Conformance Time
LRD:	Long-range Dependent

MA: Moving Average

MBS: Maximum Burst Size

MCR: Minumum Cell Rate

MIAD: Multiplicative Increase Additive Decrease

MIMD: Multiplicative Increase Multiplicative Decrease

MPLS: MultiProtocol Label Switching

OSPF: Open Shortest-Path First

PCR: Peak Cell Rate

QoS: Quality of service

RIP: Routing Internet Protocol

RSPEC: Resource Specification

SCR: Sustained Cell Rate

SIC: Sensitive to Initial Conditions

TAT: Theoretical Arrival Time

TCP: Transmission Control Protocol

TSPEC: Traffic Specification

UBR: Unspecified Bit Rate

VBR: Variable Bit Rate

VoIP: Voice over IP

VSA: Virtual Scheduling Algorithm

References

1. R Adler, Raisa Feldman, and Murad Taqqu. *A Practical Guide to Heavy Tails*. Birkhäuser, Boston, 1998. 5.1
2. V Alwayn. *Advanced MPLS Design and Implementation*. Cisco Press, Indianapolis, 2002. 1.3
3. D Bansal and H Balakrishman. Binomial congestion control algorithms. *IEEE INFOCOM 2001*, 2001. 9.2
4. J S Bendat and A G Piersol. *Random Data Analysis and Measurement Procedures*. John Wiley and Sons, New York, 1986.
5. J Beran. *Statistics for Long-Memory Processes*. Chapman and Hall, New York, 1994.
6. J Beran, R Sherman, M S Taqqu, and W Willinger. Long-range dependence in variable-bit-rate video traffic. *IEEE/ACM Transactions on Communications*, 43(2/3/4), 1995. 1, 1.2, 5, 6
7. D. R. Boggs, J. C. Mogul, and C. A. Kent. Measured capacity of an ethernet: myths and reality. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 222–234, New York, 1988. ACM Press. 1.2
8. G Box, W Hunter, and J Hunter. *Statistics for Experimenters*. Wiley and Sons, New York, 1978.
9. V G Cerf. U.S. Senate Committee on Commerce, science, and Transportation hearing on "Network Neutrality", 9 2006. http://commerce.senate.gov/pdf/cerf-020706.pdf. 1.1
10. Cheng-Shang Chang. *Performance Guarantees in Communication Networks*. Springer, New York, 2000. 4.3.6, 4.4, 5.4
11. D Chiu and R Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.
12. D Comer. *Internetworking With TCP/IP Volume 1: Principles Protocols, and Architecture*. Prentice Hall, Englewood CLiffs, NJ, 5 edition, 2006. 1.1, 1.3
13. M E Corvella and A Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6), 12 1997. 1, 1.2, 6
14. L G Cuthbert and J C Sapanel. *ATM the Broadband Telecommunications Solution*. The Institution of Electrical Engineers, London, 1996.
15. P E and Makoto Maejima. *Self-similar Processes*. Princeton, New Jersey, 2002.
16. A Erramilli, O Narayan, and W Willinger. Experimental queueing analysis with long-range dependent packet traffic. *IEEE/ACM Transactions Networking*, 4(2):209–223, 1996. 1, 1.2, 5

17. A Erramilli, R Singh, and P. Pruthi. Modeling packet traffic with chaotic maps, 7 1994. Royal Institute of Technology, ISRN KTH/IT/R-94/18–SE, Stockholm-Kista, Sweden. 7.1, 7.2
18. J Evers. Spammers now own email's dirty reputation, 2006. http://www.silicon.com /research/specialreports/thespamreport/0,39025001,39160781,00.htm. 1.1
19. T Ferrari, T Chown, N Simar, R Sabatino, S Venaas, and S Leinen. Experiments with less than best effort (LBE) Quality of Service, 08 2002. http://www.dante.net/tf-ngn/ D9.9-lbe.pdf. 1.1
20. Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transations on Networking*, 1(4):397–413, 8 1993. 1.3
21. J E Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, New York, 1998.
22. N Giroux and S Ganti. *Quality of Service in ATM Networks*. Prentice Hall, Englewood Cliffs, NJ, 1999. 1.1
23. A Gupta, D O Stahl, and A B Whinston. The economics of network management. *Communications of the ACM*, 42(9), 9 1999. 1.1
24. M Handley, S Floyd, J Padhye, and J Widmer. TCP friendly rate control (TFRC): Protocol specification. 1 2003. 9
25. Y Hayel, D Ros, and B Tuffin. Less-than-best-effort services, pricing and scheduling. In *INFOCOMM 2004*, Albuquerque, 2004. 1.1
26. C W Helstrom. *Probability and Stochastic Processes for Engineers*. Macmillan, New York, 2nd edition, 1984. 6.2, 6.2
27. A Holt. *Improving the Performance of Wide Area Networks*. PhD thesis, The Open University, 1999.
28. A Holt. Long-range dependence and self-similarity in world-wide web proxy cache references. *IEE Proceeding Communications*, 147(6):317–321, 2000. 1, 1.2
29. Van Jacobson. Congestion avoidance and control. *Proc ACM SIGCOMM*, pages 314–329, 8 1988. 1, 9
30. S Jin, G Liang, I Matta, and A Bestavros. A spectrum of TCP-friendly window-based congestion control algorithms. *IEEE/ACM Transaction on Networking*, 11(3):341–355, 6 2003. 9.2
31. F Kelly. Effective bandwidths at multi-class queues. *Queuing Systems*, 28:5–16, 1992. 6.4
32. M Kendall and J K Ord. *Time Series*. Edward Arnold, London, 3rd edition, 1990.
33. L Kleinrock. *Queuing Systems*. John Wiley and Sons, New York, 1975. 4
34. E Kohler, M Handley, and S Floyd. Designing DCCP: Congestion control without reliability, 2003. http://citeseer.ist.psu.edu/kohler03designing.html. 1
35. D Kouvatos. *Performance Evaluation and Application of ATM Networks*. Kluwer Academic Publishers, London, 2000.
36. Jean-Yves le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems of the Internet*. Springer, New York, 2001. 4.3.6, 4.6
37. W E Leland, M S Taqqu, W Willinger, and D Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, 2(1), 2 1994. 1, 6
38. J Li, A Wolisz, and R Popescu-Zeletin. Modelling and simulation of fractional ARIMA processes based on importance sampling. In *SAC '98: Proceedings of the 1998 ACM Symposium on Applied Computing*, pages 453–455, New York, 1998. ACM Press. 5.3.3
39. D Lindley. The theory of queues with a single server. *Proc. Camb. Phil. Soc.*, 48(1052):277–289, 1952. 1.4, 4.4, 5.4, 9.2
40. M Loukides. *System Performance Testing*. O'Reilly and & Associates, Sepastopol, California, 1991.

41. Paul A Lynn and Wolfgang Fuerst. *Introductory Digital Signal Processing with Computer Applications*. John Wiley & Sons, New York, USA, 1994. 3
42. G Maedel. *Mathematics for Radio and Communication, Book II Trigonometry, Alegbra, Complex Numbers with Answers*. Maedel Publishing House, New York, 1939.
43. S Makridakis, S Wheelwright, and R Hyndman. *Forecasting Methods and Applications*. John Wiley and Sons, New York, 3rd edition, 1998.
44. M G Marsh. *Policy Routing using Linux*. Sams, USA, 2001. 1.3
45. D McDysan and D Spohn. *ATM Theory and Applications*. McGraw-Hill, New York, 1999.
46. L W McKnight and J P Bailey. Internet ecomomics: When constituencies collide in cyberspace. *IEEE Internet Computing*, 6(1):30–37, 11 1997. 1.1
47. R M Metcalfe and D R Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(5):395–404, 7 1976. 1.2
48. W B Norton. Internet service providers and peering. In *Proceedings of NANOG 19*, Albuquerque, New Mexico, June 2000. 1.1
49. W B Norton. Art of peering: The peering playbook, 2006. http://arneill-py.sacramento.ca.us/ipv6mh/playbook.pdf. 1.1
50. Consumers' Institute of New Zealand. Internet neutrality? http://www.consumer.org.nz/newsitem.asp?docid=2624&category=News&topic=ISPs\able\to\slow\down\VoIP\services, 6 2006. 1.1
51. D Passmore. Ethernet: Not just for LANs anymore, 07 2000. http://www.burtongroup.com/promo/columns/column.asp?articleid=80&employeeid=56. 1.2
52. H Peitgen, H Jurgens, D Saupe, E Maletsky, T Pericante, and L Yunker. *Fractals for the Classroom, Part One*. Springer-Verlag, New York, 1992. 5.1, 7.1
53. J M Pitts and J A Schormans. *Introduction to IP and ATM Design and Performance*. John Wiley and Sons, Chichester, 1996.
54. W Press, W Vettering, S Taukolsky, and B Flannery. *Numerical Recipes in C*. Cambridge Press, Cambridge, 2002. 5.1
55. P Pruthi and A Erramilli. Heavy-tailed on/off source behavior and self-similarity. volume 1, pages 445–450, 6 1995. 6.5
56. K Ramakrishnan and R Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM, Transaction on Computer Systems*, 8(2):158–181, 5 1990. 9.2
57. H Rich. J for C programmers. http://www.jsoftware.com/jwiki/Doc/Books, 10 2004.
58. R Y Rubinstein. *Simulation and the Monte Carlo Methods*. John Wiley and Sons, New York, 1981.
59. Lionel Salem, Frederic, and Coralie Salam. *The Most Beautiful Mathematical Formulas*. John Wiley & Sons, New York, USA, 1992. 3.2.3
60. Christian Sandvig. Network neutrality is the new common carrier. *The Journal of Policy, Regulation, and Strategy*, 7 2006. 1.1
61. M Schuyler. Measuring network traffic, 2001. http://www.solarwinds.net/Awards/View.htm. 1.2
62. Claude E Shannon and Warren Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Chicago, USA, 1963. 3.2.2, 3.2.4
63. R Sircant. *The Mathematics of Internet Congestion Control*. Birkhauser, Boston, 2004. 9.1, 9.3
64. W R Stevens. *TCP/IP Illustrated Volume 1, The Protocols*. Addison-Wesley, Boston, 1994.
65. C Stoll. *The Cukcoo's Egg*. Bodley Head, UK, 1989. 1.1
66. A S Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, NJ, 1989. 1.2

67. R Taylor. The great firewall of China. *BBC News*, 6 2006. http://news.bbc.co.uk/2/hi/programmes/click_online/4587622.stm. 1.1
68. D Teare. *CCDA Self Study: Designing for Cisco Internetwork Solutions*. Ciscopress, 2003. 1.2, 1.3
69. T M Thomas. *OSPF Network Design Solution*. Ciscopress, Indianapolis, 1998. 1.3
70. N Thomson. J for engineers and computing professionals. *IEE Computing & Control Engineering Journal*, 12(5):212–216, 10 2001.
71. N Thomson. *J: The Natural Language for Analytical Computing*. Research Studies Press, Baldock Hertfordshire, 2001. 5.1
72. T. Tuan and K. Park. Congestion control for self-similar network traffic. *Technical Report CSD-TR-98014*, 1998.
73. S Vegesna. *IP Quality of Service*. Ciscopress, Indianapolis, 2001. 1.1, 1.3
74. M Ward. More than 95% of e-mail is junk, 2006. http://news.bbc.co.uk/2/hi/technology/5219554.stm. 1.1
75. K Xu and N Ansari. Stability and fairness of rate estimation-based AIAD congestion control in TCP. *IEEE Communications Letters*, 9(4):378–380, 4 2004. 9

Index