Resilient Overlay Networks

Another routing overlay gaining in popularity is one that finds alternative routes for traditional unicast applications. Such overlays exploit the observation that the triangle inequality does not hold in the Internet. Figure 9.21 illustrates what we mean by this. It is not uncommon to find three sites in the Internet—call them A, B, and C—such that the latency between A and B is greater than the sum of the latencies from A to C, and from C to B. That is, sometimes you would be better off indirectly sending your packets via some intermediate node than sending them directly to the destination.

How can this be? Well, BGP never promised that it would find the *shortest* route between any two sites; it only tries to find *some* route. To make matters worse, there are countless opportunities for human-directed policies to override BGP's normal operation. This often happens, for example, at peering points between major backbone ISPs. In short, that the triangle inequality does not hold in the Internet should not come as a surprise.

How do we exploit this observation? The first step is to realize that there is a fundamental trade-off between the scalability and optimality of a routing algorithm. On the one hand, BGP scales to very large networks, but often does not select the best possible route and is slow to adapt to network outages. On the other hand, if you were only worried about finding the best route among a handful of sites, you could do a much better

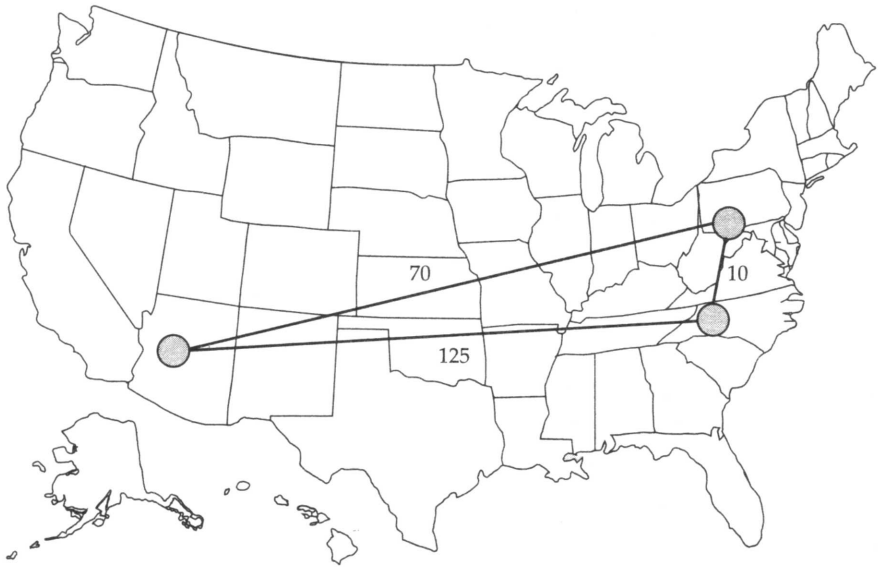


Figure 9.21 The triangle inequality does not necessarily hold in networks.

job of monitoring the quality of every path you might use, thereby allowing you to select the best possible route at any moment in time.

An experimental overlay, called Resilient Overlay Network (RON), does exactly this. RON scales to only a few dozen nodes because it uses an N×N strategy of closely monitoring (via active probes) three aspects of path quality—latency, available bandwidth, and loss probability—between every pair of sites. It is then able to both select the optimal route between any pair of nodes and rapidly change routes should network conditions change. Experience shows that RON is able to deliver modest performance improvements to applications, but more importantly, it recovers from network failures much more quickly. For example, during one 64-hour period in 2001, an instance of RON running on 12 nodes detected 32 outages lasting over 30 minutes, and it was able to recover from all of them in less than 20 seconds on average. This experiment also suggested that forwarding data through just one intermediate node is usually sufficient to recover from Internet failures.

Since RON does not scale, it is not possible to use RON to help random host A communicate with random host B; A and B have to know ahead of time that they are likely to communicate, and then join the same RON. However, RON seems like a good idea in certain settings, such as when connecting a few dozen corporate sites spread across the Internet, or allowing you and 50 of your friends to establish your own private overlay for the sake of running some application. The real question, though, is what happens when everyone starts to run their own RON. Does the overhead of millions of RONs aggressively probing paths swamp the network, and does anyone see improved behavior when many RONs compete for the same paths? These questions are still unanswered.

All of these overlays illustrate a concept that is central to computer networks in general: *virtualization*. That is, it is possible to build a virtual network from abstract (logical) resources on top of a physical network constructed from physical resources. Moreover, it is possible to stack these virtualized networks on top of each other, and for multiple virtual networks to coexist at the same level. Each virtual network, in turn, provides new capabilities that are of value to some set of users, applications, or higher-level networks.

9.4.2 Peer-to-Peer Networks (Gnutella, BitTorrent)

Music-sharing applications like Napster and KaZaA introduced the term "peer-to-peer" into the popular vernacular. But what exactly does it mean for a system to be peer-to-peer? Certainly in the context of sharing MP3 files it means not having to download music from a central site, but instead being able to access music files directly from whoever in the Internet happens to have a copy stored on their computer. More generally then, we could say that a peer-to-peer network allows a community of users to pool their resources (content, storage, network bandwidth, disk bandwidth, CPU), thereby

providing access to a larger archival store, larger video/audio conferences, more complex searches and computations, and so on, than any one user could afford individually.

Quite often, attributes like *decentralized* and *self-organizing* are mentioned when discussing peer-to-peer networks, meaning that individual nodes organize themselves into a network without any centralized coordination. If you think about it, terms like these could be used to describe the Internet itself. Ironically, however, Napster is not a true peer-to-peer system by this definition since it depends on a central registry of known files, and users have to search this directory to find what machine offers a particular file. It is only the last step—actually downloading the file—that takes place between machines that belong to two users, but this is little more than a traditional client/server transaction. The only difference is that the server is owned by someone just like you rather than a large corporation.

So we are back to the original question: What's interesting about peer-to-peer networks? One answer is that both the process of locating an object of interest and the process of downloading that object onto your local machine happen without your having to contact a centralized authority, and at the same time, the system is able to scale to millions of nodes. A peer-to-peer system that can accomplish these two tasks in a decentralized manner turns out to be an overlay network, where the nodes are those hosts that are willing to share objects of interest (e.g., music and other assorted files), and the links (tunnels) connecting these nodes represent the sequence of machines that you have to visit to track down the object you want. This description will become clearer after we look at two examples.

Gnutella

Gnutella is an early peer-to-peer network that attempted to distinguish between exchanging music (which likely violates somebody's copyright) and the general sharing of files (which must be good since we've been taught to share since the age of two). What's interesting about Gnutella is that it was one of the first such systems to not depend on a centralized registry of objects. Instead Gnutella participants arrange themselves into an overlay network similar to the one shown in Figure 9.22. That is, each node that runs the Gnutella software (i.e., implements the Gnutella protocol) knows about some set of other machines that also run the Gnutella software. The relationship "A and B know each other" corresponds to the edges in this graph. (We'll talk about how this graph is formed in a moment.)

Whenever the user on a given node wants to find an object, Gnutella sends a QUERY message for the object—for example, specifying the file's name—to its neighbors in the graph. If one of the neighbors has the object, it responds to the node that sent it the query with a QUERY RESPONSE message, specifying where the object can be downloaded (e.g., an IP address and TCP port number). That node can subsequently

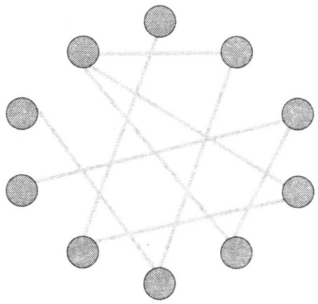


Figure 9.22 Example topology of a Gnutella peer-to-peer network.

use GET or PUT messages to access the object. If the node cannot resolve the query, it forwards the QUERY message to each of its neighbors (except the one that sent it the query), and the process repeats. In other words, Gnutella floods the overlay to locate the desired object. Gnutella sets a TTL on each query so this flood does not continue indefinitely.

In addition to the TTL and query string, each QUERY message contains a unique query identifier (QID), but it does not contain the identity of the original message source. Instead, each node maintains a record of the QUERY messages it has seen recently: both the QID and the neighbor that sent it the QUERY. It uses this history in two ways. First, if it ever receives a QUERY with a QID that matches one it has seen recently, the node does not forward the QUERY message. This serves to cut off forwarding loops more quickly than the TTL might have done. Second, whenever the node receives a QUERY RESPONSE from a downstream neighbor, it knows to forward the response to the upstream neighbor that originally sent it the QUERY message. In this way, the response works its way back to the original node without any of the intermediate nodes knowing who wanted to locate this particular object in the first place.

Returning to the question of how the graph evolves, a node certainly has to know about at least one other node when it joins a Gnutella overlay. The new node is attached to the overlay by at least this one link. After that, a given node learns about other nodes as the result of QUERY RESPONSE messages, both for objects it requested and for responses that just happen to pass through it. A node is free to decide which of the nodes it discovers in this way that it wants to keep as a neighbor. The Gnutella protocol provides PING and PONG messages by which a node probes whether or not a given neighbor still exists and that neighbor's response, respectively.

It should be clear that Gnutella as described here is not a particularly clever protocol, and subsequent systems have tried to improve upon it. One dimension along which improvements are possible is in how queries are propagated. Flooding has the nice prop-

erty that it is guaranteed to find the desired object in the fewest possible hops, but it does not scale well. It is possible to forward queries randomly, or according to the probability of success based on past results. A second dimension is to proactively replicate the objects, since the more copies of a given object there are, the easier it should be to find a copy. Alternatively, one could develop a completely different strategy, which is the topic we consider next.

Structured Overlays

At the same time file-sharing systems have been fighting to fill the void left by Napster, the research community has been exploring an alternative design for peer-to-peer networks. We refer to these networks as *structured*, to contrast them with the essentially random (unstructured) way in which a Gnutella network evolves. Unstructured overlays like Gnutella employ trivial overlay construction and maintenance algorithms, but the best they can offer is unreliable, random search. In contrast, structured overlays are designed to conform to a particular graph structure that allows reliable and efficient (probabilistically bounded delay) object location, in return for additional complexity during overlay construction and maintenance.

If you think about what we are trying to do at a high level, there are two questions to consider: (1) How do we map objects onto nodes? and (2) How do we route a request to the node that is responsible for a given object? We start with the first question, which has a simple statement: How do we map an object with name x into the address of some node n that is able to serve that object? While traditional peer-to-peer networks have no control over which node hosts object x, if we could control how objects get distributed over the network, we might be able to do a better job of finding those objects at a later time.

A well-known technique for mapping names into addresses is to use a hash table, so that

$$hash(x) \longrightarrow n$$

implies object x is first placed on node n, and at a later time, a client trying to locate x would only have to perform the hash of x to determine that it is on node n. A hash-based approach has the nice property that it tends to spread the objects evenly across the set of nodes, but straightforward hashing algorithms suffer from a fatal flaw: How many possible values of n should we allow? (In hashing terminology, how many buckets should there be?) Naively, we could decide that there are, say, 101 possible hash values, and we use a modulo hash function, that is,

```
hash(x)
  return x % 101
```

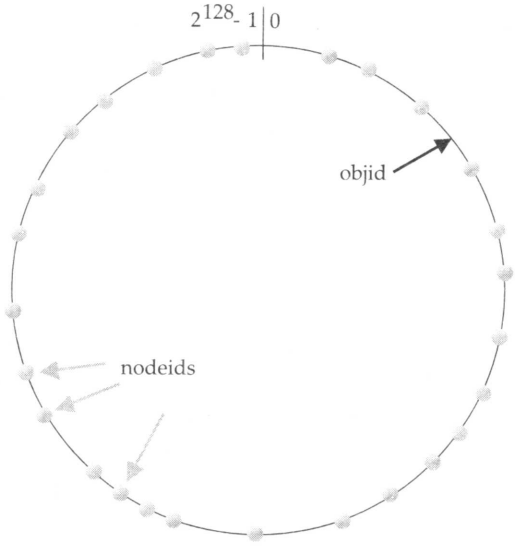


Figure 9.23 Both nodes and objects map (hash) onto the ID space, where objects are maintained at the nearest node in this space.

Unfortunately, if there are more than 101 nodes willing to host objects, then we can't take advantage of all of them. On the other hand, if we select a number larger than the largest possible number of nodes, then there will be some values of x that will hash into an address for a node that does not exist. There is also the not-so-small issue of translating the value returned by the hash function into an actual IP address.

To address these issues, structured peer-to-peer networks use an algorithm known as *consistent hashing*, which hashes a set of objects x uniformly across a large ID space. Figure 9.23 visualizes a 128-bit ID space as a circle, where we use the algorithm to place both objects

$$hash(object\_name) \longrightarrow objid$$

and nodes

$$hash(IP\_addr) \longrightarrow nodeid$$

onto this circle. Since a 128-bit ID space is enormous, it is unlikely that an object will hash to exactly the same ID as a machine's IP address hashes to. To account for this unlikelihood, each object is maintained on the node whose ID is *closest*, in this 128-bit space, to the object ID. In other words, the idea is to use a high-quality hash function to map both nodes and objects into the same large, sparse ID space; you then map objects

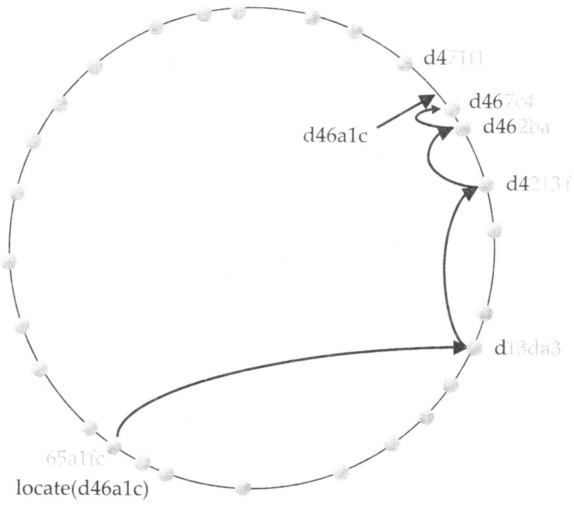


Figure 9.24 Objects are located by routing through the peer-to-peer overlay network.

to nodes by numerical proximity of their respective identifiers. Like ordinary hashing, this distributes objects fairly evenly across nodes, but unlike ordinary hashing, only a small number of objects have to move when a node (hash bucket) joins or leaves.

We now turn to the second question: How does a user that wants to access object x know which node is closest in x's ID in this space? One possible answer is that each node keeps a complete table of node IDs and their associated IP addresses, but this would not be practical for a large network. The alternative, which is the approach used by structured peer-to-peer networks, is to *route a message to this node!* In other words, if we construct the overlay in a clever way—which is the same as saying that we need to choose entries for a node's routing table in a clever way—then we find a node simply by routing toward it. Collectively, this approach is sometimes called *distributed hash tables (DHT)*, since conceptually, the hash table is distributed over all the nodes in the network.

Figure 9.24 illustrates what happens for a simple 28-bit ID space. To keep the discussion as concrete as possible, we consider the approach used by a particular peer-to-peer network called Pastry. Other systems work in a similar manner. (See the papers cited at the end of the chapter for additional examples.)

Suppose you are at the node with ID 65a1fc (hex) and you are trying to locate the object with ID d46a1c. You realize that your ID shares nothing with the object's, but you know of a node that shares at least the prefix d. That node is closer than you in the 128-bit ID space, so you forward the message to it. (We do not give the format of the message being forwarded, but you can think of it as saying "locate object d46a1c.")

Assuming node d13da3 knows of another node that shares an even longer prefix with the object, it forwards the message on. This process of moving closer in ID space continues until you reach a node that knows of no closer node. This node is, by definition, the one that hosts the object. Keep in mind that as we logically move through ID space the message is actually being forwarded, node to node, through the underlying Internet.

Each node maintains both a routing table (more below) and the IP addresses of a small set of numerically larger and smaller node IDs. This is called the node's *leaf set*. The relevance of the leaf set is that once a message is routed to any node in the same leaf set as the node that hosts the object, that node can directly forward the message to the ultimate destination. Said another way, the leaf set facilitates correct and efficient delivery of a message to the numerically closest node, even though multiple nodes may exist that share a maximal length prefix with the object ID. Moreover, the leaf set makes routing more robust because any of the nodes in a leaf set can route a message just as well as any other node in the same set. Thus, if one node is unable to make progress routing a message, one of its neighbors in the leaf set may be able to. In summary, the routing procedure is defined as follows:

```
Route(D)
 if D is within range of my leaf set
   forward to numerically closest member in leaf set
 else
  let l = length of shared prefix
  let d = value of l-th digit in D's address
  if RouteTab[l, d] exists
    forward to RouteTab[l, d]
  else
   forward to known node with at least as long a shared prefix
   and numerically closer than this node
```

The routing table, denoted RouteTab, is a two-dimensional array. It has a row for every hex digit in an ID (there such 32 digits in a 128-bit ID) and a column for every hex value (there are obviously 16 such values). Every entry in row i shares a prefix of length i with this node, and within this row, the entry in column j has the hex value j in the $i + 1$th position. Figure 9.25 shows the first three rows of an example routing table for node 65a1fc$x$, where x denotes an unspecified suffix. This figure shows the ID prefix matched by every entry in the table. It does not show the actual value contained in this entry—the IP address of the next node to route to.

Adding a node to the overlay works much like routing a "locate object message" to an object. The new node must know of at least one current member. It asks this member to route an "add node message" to the node numerically closest to the ID of the joining

Row 0	0	1	2	3	4	5		7	8	9	a	b	c	d	e	f
	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x
Row 1	6	6	6	6	6		6	6	6	6	6	6	6	6	6	6
	0	1	2	3	4		6	7	8	9	a	b	c	d	e	f
	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x
Row 2	6	6	6	6	6	6	6	6	6	6		6	6	6	6	6
	5	5	5	5	5	5	5	5	5	5		5	5	5	5	5
	0	1	2	3	4	5	6	7	8	9		b	c	d	e	f
	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x
Row 3	6		6	6	6	6	6	6	6	6	6	6	6	6	6	6
	5		5	5	5	5	5	5	5	5	5	5	5	5	5	5
	a		a	a	a	a	a	a	a	a	a	a	a	a	a	a
	0		2	3	4	5	6	7	8	9	a	b	c	d	e	f
	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x

Figure 9.25 Example routing table at the node with ID 65a1fc*x*.

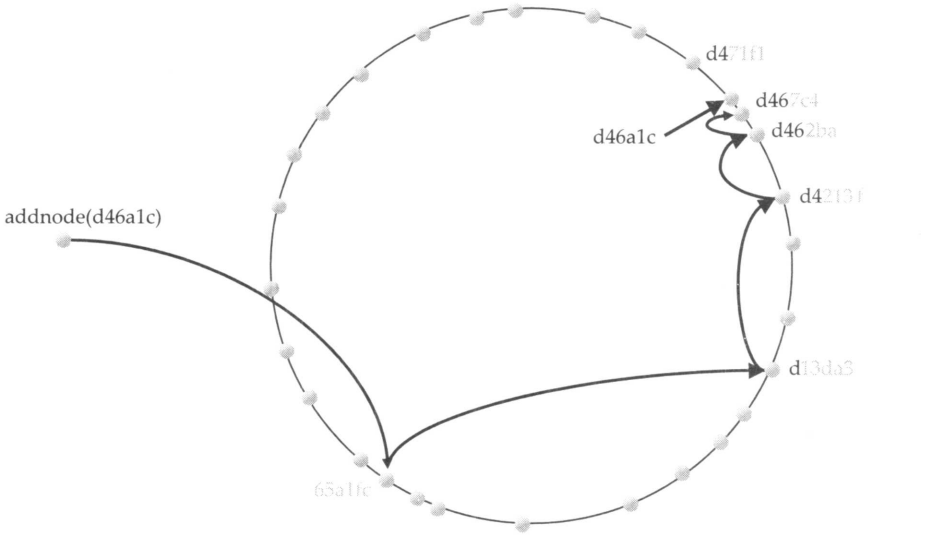


Figure 9.26 Adding a node to the network.

node, as shown in Figure 9.26. It is through this routing process that the new node learns about other nodes with a shared prefix, and is able to begin filling out its routing table. Over time, as additional nodes join the overlay, existing nodes also have the option of including information about the newly joining node in their routing tables. They do

this when the new node adds a longer prefix than they currently have in their table. Neighbors in the leaf sets also exchange routing tables with each other, which means that over time routing information propagates through the overlay.

The reader may have noticed that although structured overlays provide a probabilistic bound on the number of routing hops required to locate a given object—the number of hops in Pastry is bounded by $log_{16}N$, where N is the number of nodes in the overlay—each hop may contribute substantial delay. This is because each intermediate node may be at a random location in the Internet. (In the worst case, each node is on a different continent!) In fact, in a worldwide overlay network using the algorithm as described above, the expected delay of each hop is the average delay among all pairs of nodes in the Internet! Fortunately, one can do much better in practice. The idea is to choose each routing table entry such that it refers to a nearby node in the underlying physical network, among all nodes with an ID prefix that is appropriate for the entry. It turns out that doing so achieves end-to-end routing delays that are within a small factor of the delay between source and destination node.

Finally, the discussion up to this point has focused on the general problem of locating objects in a peer-to-peer network. Given such a routing infrastructure, it is possible to build different services. For example, a file sharing service would use file names as object names. To locate a file, you first hash its name into a corresponding object ID, and then route a "locate object message" to this ID. The system might also replicate each file across multiple nodes to improve availability. Storing multiple copies on the leaf set of the node to which a given file normally routes would be one way of doing this. Keep in mind that even though these nodes are neighbors in the ID space, they are likely to be physically distributed across the Internet. Thus, while a poweroutage in an entire city might take down physically close replicas of a file in a traditional file system, one or more replicas would likely survive such a failure in a peer-to-peer network.

Services other than file-sharing can also be built on top of distributed hash tables. Consider multicast applications, for example. Instead of constructing a multicast tree from a mesh, one could construct the tree from edges in the structured overlay, thereby amortizing the cost of overlay construction and maintenance across several applications and multicast groups.

BitTorrent

BitTorrent is a peer-to-peer file sharing protocol devised by Bram Cohen. It is based on replicating the file, or rather, replicating segments of the file, which are called *pieces*. Any particular piece can usually be downloaded from multiple peers, even if only one peer has the entire file. The primary benefit of BitTorrent's replication is avoiding the bottleneck of having only one source for a file. The beauty of BitTorrent is that replication is a natural side effect of the downloading process: as soon as a peer downloads a particular

piece, it becomes another source for that piece. The more peers downloading pieces of the file, the more piece replication occurs, distributing the load proportionately. Pieces are downloaded in random order to avoid a situation where peers find themselves lacking the same set of pieces.

Each file is shared via its own independent BitTorrent network, called a *swarm*. (A swarm could potentially share a set of files, but we describe the single file case for simplicity.) The lifecycle of a typical swarm is as follows. The swarm starts as a singleton peer with a complete copy of the file. A node that wants to download the file joins the swarm, becoming its second member, and begins downloading pieces of the file from the original peer. In doing so, it becomes another source for the pieces it has downloaded, even if it has not yet downloaded the entire file. (In fact, it is common for peers to leave the swarm once they have completed their downloads, although they are encouraged to stay longer.) Other nodes join the swarm and begin downloading pieces from multiple peers, not just the original peer. (See Figure 9.27.)

If the file remains in high demand, with a stream of new peers replacing those who leave the swarm, the swarm could remain active indefinitely; if not, it could shrink back to include only the original peer until new peers join the swarm.

Now that we have an overview of BitTorrent, we can ask how requests are routed to the peers that have a given piece. To make requests, a would-be downloader must first join the swarm. It starts by downloading a .torrent file containing metainformation about the file and swarm. The .torrent file, which may be easily replicated, is typically

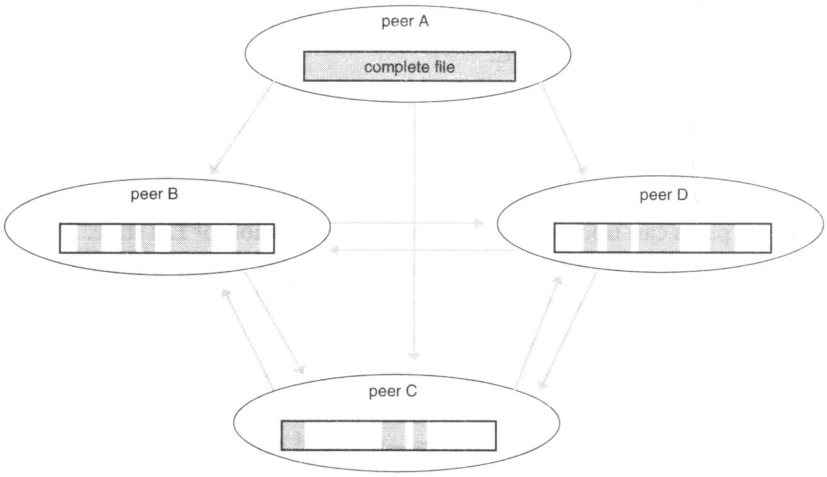


Figure 9.27 Peers in a BitTorrent swarm download from other peers that may not yet have the complete file.

downloaded from a web server and discovered by following links from web pages. It contains:

- The target file's size;
- The piece size;
- SHA-1 hash values (Section 8.1.4) precomputed from each piece;
- The URL of the swarm's *tracker*.

A tracker is a server that tracks a swarm's current membership. We'll see later that BitTorrent can be extended to eliminate this point of centralization, with its attendant potential for bottleneck or failure.

The would-be downloader then joins the swarm, becoming a peer, by sending a message to the tracker giving its network address and a peer ID that it has generated randomly for itself. The message also carries a SHA-1 hash of the main part of the `.torrent` file, which is used as a swarm ID.

Let's call the new peer P. The tracker replies to P with a partial list of peers giving their IDs and network addresses, and P establishes connections, over TCP, with some of these peers. Note that P is directly connected to just a subset of the swarm, although it may decide to contact additional peers or even request more peers from the tracker. To establish a BitTorrent connection with a particular peer after their TCP connection has been established, P sends P's own peer ID and swarm ID, and the peer replies with its peer ID and swarm ID. If the swarm IDs don't match, or the reply peer ID is not what P expects, the connection is aborted.

The resulting BitTorrent connection is symmetric: each end can download from the other. Each end begins by sending the other a bitmap reporting which pieces it has, so each peer knows the other's initial state. Whenever a downloader D finishes downloading another piece, it sends a message identifying that piece to each of its directly connected peers, so those peers can update their internal representation of D's state. This, finally, is the answer to the question of how a download request for a piece is routed to a peer that has the piece, because it means that each peer knows which directly connected peers have the piece. If D needs a piece that none of its connections has, it could connect to more or different peers (it can get more from the tracker), or occupy itself with other pieces in hopes that some of its connections will obtain the piece from their connections.

How are objects—in this case, pieces—mapped onto peer nodes? Of course each peer eventually obtains all the pieces, so the question is really about which pieces a peer has at a given time before it has all the pieces, or equivalently, about the order in which a peer downloads pieces. The answer is that they download pieces in random order, to keep them from having a strict subset or superset of the pieces of any of their peers.

The BitTorrent described so far utilizes a central tracker that constitutes a single point of failure for the swarm and could potentially be a performance bottleneck. Also, providing a tracker can be a nuisance for someone who would like to make a file available via BitTorrent. Newer versions of BitTorrent additionally support trackerless swarms that use a DHT-based implementation. BitTorrent client software that is trackerless-capable implements not just a BitTorrent peer, but also what we'll call a peer finder (the BitTorrent terminology is simply "node") that the peer uses to find peers.

Peer finders form their own overlay network, using their own protocol over UDP to implement a DHT. Furthermore, a peer finder network includes peer finders whose associated peers belong to different swarms. In other words, while each swarm forms a distinct network of BitTorrent peers, a peer finder network instead spans swarms.

Peer finders randomly generate their own finder IDs, which are the same size (160 bits) as swarm IDs. Each finder maintains a modest table containing primarily finders (and their associated peers) whose IDs are close to its own, plus some finders whose IDs are more distant. The following algorithm ensures that finders whose IDs are close to a given swarm ID are likely to know of peers from that swarm; the algorithm simultaneously provides a way to look them up. When a finder F needs to find peers from a particular swarm, it sends a request to the finders in its table whose IDs are close to that swarm's ID. If a contacted finder knows of any peers for that swarm, it replies with their contact information. Otherwise, it replies with the contact information of the finders in its table that are close to the swarm, so that F can iteratively query those finders.

After the search is exhausted, because there are no finders closer to the swarm, F inserts the contact information for itself and its associated peer into the finders closest to the swarm. The net effect is that peers for a particular swarm get entered in the tables of the finders that are close to that swarm.

The above scheme assumes that F is already part of the finder network, that is, that it already knows how to contact some other finders. This assumption is true for finder installations that have run previously, because they are supposed to save information about other finders, even across executions. If a swarm uses a tracker, its peers are able to tell their finders about other finders (in a reversal of the peer and finder roles) because the BitTorrent peer protocol has been extended to exchange finder contact information. But how can a newly installed finder discover other finders? The `.torrent` files for trackerless swarms include contact information for one or a few finders, instead of a tracker URL, for just that situation.

An unusual aspect of BitTorrent is that it deals head-on with the issue of fairness, or good "network citizenship." Protocols often depend on the good behavior of individual peers without being able to enforce it. For example, an unscrupulous Ethernet peer could get better performance by using a backoff algorithm that is more aggressive than

exponential backoff, or an unscrupulous TCP peer could get better performance by not cooperating in congestion control.

The good behavior that BitTorrent depends on is peers uploading pieces to other peers. Since the typical BitTorrent user just wants to download the file as quickly as possible, there is a temptation to implement a peer that tries to download all the pieces while doing as little uploading as possible—this is a bad peer. To discourage bad behavior, the BitTorrent protocol includes mechanisms that allow peers to reward or punish each other. If a peer is misbehaving by not nicely uploading to another peer, the second peer can *choke* the bad peer: it can decide to stop uploading to the bad peer, at least temporarily, and send it a message saying so. There is also a message type for telling a peer that it has been unchoked. The choking mechanism is also used by a peer to limit the number of its active BitTorrent connections, to maintain good TCP performance. There are many possible choking algorithms, and devising a good one is an art.

9.4.3 Content Distribution Networks

We have already seen how HTTP running over TCP allows web browsers to retrieve pages from web servers. However, anyone that has waited an eternity for a web page to return knows that the system is far from perfect. Considering that the backbone of the Internet is now constructed from OC-192 (10-Gbps) links, it's not obvious why this should happen. It is generally agreed that when it comes to downloading web pages, there are three potential bottlenecks in the system:

- The first mile. The Internet may have high-capacity links in it, but that doesn't help you download a web page any faster when you're connected by a 56-Kbps modem.
- The last mile. The link that connects the server to the Internet, along with the server itself, can be overloaded by too many requests.
- Peering points. The handful of ISPs that collectively implement the backbone of the Internet may internally have high-bandwidth pipes, but they have little motivation to provide high-capacity connectivity to their peers. If you are connected to ISP A and the server is connected to ISP B, then the page you request may get dropped at the point A and B peer with each other.

There's not a lot anyone except you can do about the first problem, but it is possible to use replication to address the second and third problems. Systems that do this are often called content distribution networks (CDN). Akamai is probably the best-known CDN.

The idea of a CDN is to geographically distribute a collection of *server surrogates* that cache pages normally maintained in some set of *backend servers*. Thus, rather than have millions of users wait forever to contact www.cnn.com when a big news story

breaks—such a situation is known as a *flash crowd*—it is possible to spread this load across many servers. Moreover, rather than having to traverse multiple ISPs to reach www.cnn.com, if these surrogate servers happen to be spread across all the backbone ISPs, then it should be possible to reach one without having to cross a peering point. Clearly, maintaining thousands of surrogate servers all over the Internet is too expensive for any one site that wants to provide better access to its web pages. Commercial CDNs provide this service for many sites, thereby amortizing the cost across many customers.

Although we call them surrogate servers, in fact, they can just as correctly be viewed as caches. If they don't have a page that has been requested by a client, they ask the backend server for it. In practice, however, the backend servers proactively replicate their data across the surrogates rather than wait for surrogates to request it on-demand. It's also the case that only static pages, as opposed to dynamic content, are distributed across the surrogates. Clients have to go to the backend server for any content that either changes frequently (e.g., as sports scores and stock quotes) or is produced as the result of some computation (e.g., a database query).

Having a large set of geographically distributed servers does not fully solve the problem. To complete the picture, CDNs also need to provide a set of *redirectors* that forward client requests to the most appropriate server, as shown in Figure 9.28. The

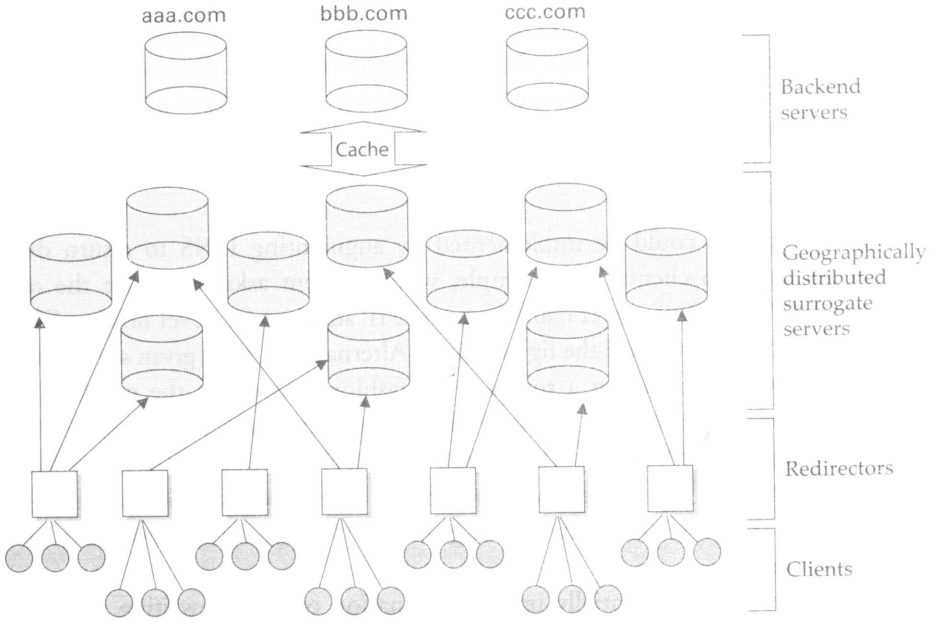


Figure 9.28 Components in a content distribution network (CDN).

primary objective of the redirectors is to select the server for each request that results in the best *response time* for the client. A secondary objective is for the system as a whole to process as many requests-per-second as the underlying hardware (network links and web servers) is able to support. The average number of requests that can be satisfied in a given time period—known as the *system throughput*—is primarily an issue when the system is under heavy load; for example, when a flash crowd is accessing a small set of pages or a distributed denial of service (DDoS) attacker is targeting a particular site, as happened to CNN, Yahoo, and several other high-profile sites in February 2000.

CDNs use several factors to decide how to distribute client requests. For example, to minimize response time, a redirector might select a server based on its *network proximity*. In contrast, to improve the overall system throughput, it is desirable to evenly *balance* the load across a set of servers. Both throughput and response time are improved if the distribution mechanism takes *locality* into consideration, that is, selects a server that is likely to already have the page being requested in its cache. The exact combination of factors that should be employed by a CDN is open to debate. This section considers some of the possibilities.

Mechanisms

As described so far, a redirector is just an abstract function, although it sounds like something a router might be asked to do since it logically forwards a request message much like a router forwards packets. In fact, there are several mechanisms that can be used to implement redirection. Note that for the purpose of this discussion we assume that each redirector knows the address of every available server. (From here on, we drop the "surrogate" qualifier and talk simply in terms of a set of servers.) In practice, some form of out-of-band communication takes place to keep this information up-to-date as servers come and go.

First, redirection could be implemented by augmenting DNS to return different server addresses to clients. For example, when a client asks to resolve the name www.cnn.com, the DNS server could return the IP address of a server hosting CNN's web pages that is known to have the lightest load. Alternatively, for a given set of servers, it might just return addresses in a round-robin fashion. Note that the granularity of DNS-based redirection is usually at the level of a site (e.g., cnn.com) rather than a specific URL (e.g., http://www.cnn.com/2002/WORLD/europe/06/21/william.birthday/index.html). However, when returning an embedded link, the server can rewrite the URL, thereby effectively pointing the client at the most appropriate server for that specific object.

Commercial CDNs essentially use a combination of URL rewriting and DNS-based redirection. For scalability reasons, the high-level DNS server first points to a regional-level DNS server, which replies with the actual server address. In order to

respond to changes quickly, the DNS servers tweak the TTL of the resource records they return to a very short period, such as 20 seconds. This is necessary so clients don't cache results, and thus fail to go back to the DNS server for the most recent URL-to-server mapping.

Another possibility is to use the HTTP redirect feature: the client sends a request message to a server, which responds with a new (better) server that the client should contact for the page. Unfortunately, server-based redirection incurs an additional round-trip time across the Internet, and even worse, servers can be vulnerable to being overloaded by the redirection task itself. Instead, if there is a node close to the client—for example, a local web proxy—that is aware of the available servers, then it can intercept the request message and instruct the client to instead request the page from an appropriate server. In this case, either the redirector would need to be on a choke point so that all requests leaving the site pass through it, or the client would have to cooperate by explicitly addressing the proxy (as with a classical, rather than transparent, proxy).

At this point you may be wondering what CDNs have to do with overlay networks, and while viewing a CDN as an overlay is a bit of a stretch, they do share one very important trait in common. Like an overlay node, a proxy-based redirector makes an application-level routing decision. Rather than forward a packet based on an address and its knowledge of the network topology, it forwards HTTP requests based on a URL and its knowledge of the location and load of a set of servers. Today's Internet architecture does not support redirection directly—where by "directly" we mean the client sends the HTTP request to the redirector, which forwards it to the destination—so instead redirection is typically implemented indirectly by having the redirector return the appropriate destination address and the client contacts the server itself.

Policies

We now consider some example policies that redirectors might use to forward requests. Actually, we have already suggested one simple policy—round-robin. A similar scheme would be to simply select one of the available servers at random. Both of these approaches do a good job of spreading the load evenly across the CDN, but they do not do a particularly good job of lowering the client-perceived response time.

It's obvious that neither of these two schemes take network proximity into consideration, but just as a importantly, they also ignore locality. That is, requests for the same URL are forwarded to different servers, making it less likely that the page will be served from the selected server's in-memory cache. This forces the server to retrieve the page from its disk, or possibly even from the backend server. How can a distributed set of redirectors cause requests for the same page to go to the same server (or small set of servers) without global coordination? The answer is surprisingly simple: All redirectors use some form of hashing to deterministically map URLs into a small range of values.

The primary benefit of this approach is that no interredirector communication is required to achieve coordinated operation; no matter which redirector receives a URL, the hashing process produces the same output.

So what makes for a good hashing scheme? The classic *modulo* hashing scheme—which hashes each URL modulo the number of servers—is not suitable for this environment. This is because should the number of servers change, the modulo calculation will result in a diminishing fraction of the pages keeping their same server assignments. While we do not expect frequent changes in the set of servers, the fact that addition of new servers into the set will cause massive reassignment is undesirable.

An alternative is to use the same *consistent hashing* algorithm discussed in Section 9.4.2. Specifically, each redirector first hashes every server into the unit circle. Then for each URL that arrives, the redirector also hashes the URL to a value on the unit circle, and the URL is assigned to the server that lies closest on the circle to its hash value. If a node fails in this scheme, its load shifts to its neighbors (on the unit circle), so the addition/removal of a server only causes local changes in request assignments. Note that unlike the peer-to-peer case, where a message is routed from one node to another in order to find the server whose ID is closest to the objects, each redirector knows how the set of servers map onto the unit circle, so they can each independently select the "nearest" one.

This strategy can easily be extended to take server load into account. Assume the redirector knows the current load of each of the available servers. This information may not be perfectly up-to-date, but we can imagine the redirector simply counting how many times it has forwarded a request to each server in the last few seconds, and using this count as an estimate of that server's current load. Upon receiving a URL, the redirector hashes the URL plus each of the available servers, and sorts the resulting values. This sorted list effectively defines the order in which the redirector will consider the available servers. The redirector then walks down this list until it finds a server whose load is below some threshold. The benefit of this approach compared to plain consistent hashing is that server order is different for each URL, so if one server fails, its load is distributed evenly among the other machines. This approach is the basis for the Cache Array Routing Protocol (CARP), and is shown in pseudocode below:

```
SelectServer(URL, S)
  for = each server s_i in server set S
    weight_i = hash(URL, address(s_i))
  sort weight
  for each server s_j in decreasing order of weight_j
    if = Load(s_j) < threshold then
      return s_j
  return server with highest weight
```

As the load increases, this scheme changes from using only the first server on the sorted list to spreading requests across several servers. Some pages normally handled by busy servers will also start being handled by less busy servers. Since this process is based on aggregate server load rather than the popularity of individual pages, servers hosting some popular pages may find more servers sharing their load than servers hosting collectively unpopular pages. In the process, some unpopular pages will be replicated in the system simply because they happen to be primarily hosted on busy servers. At the same time, if some pages become extremely popular, it is conceivable that all of the servers in the system could be responsible for serving them.

Finally, it is possible to introduce network proximity into the equation in at least two different ways. The first is to blur the distinction between server load and network proximity by monitoring how long a server takes to respond to requests, and using this measurement as the server load parameter in the preceding algorithm. This strategy tends to prefer nearby/lightly-loaded servers over distant/heavily-loaded servers. A second approach is to factor proximity into the decision at an earlier stage by limiting the candidate set of servers considered by the above algorithms (S) to only those that are nearby. The harder problem is deciding which of the potentially many servers are suitably close. One approach would be to select only those servers that are available on the same ISP as the client. A slightly more sophisticated approach would be to look at the map of autonomous systems produced by BGP, and select only those servers within some number of hops from the client as candidate servers. Finding the right balance between network proximity and server cache locality is a subject of ongoing research.

9.5 Summary

We have seen four client/server-based application protocols: SMTP used to exchange electronic mail, HTTP used to walk the World Wide Web, the DNS protocol used by the domain naming system, and SNMP used to query remote nodes for the sake of network management. We have seen how application-to-application communication is driving the creation of new protocol development frameworks such as SOAP and REST. And we have examined session control protocols, such as SIP and H.323, which are used to control multimedia applications such as voice over IP. Finally, we looked at emerging applications—including overlay, peer-to-peer, and content distribution networks—that blend application processing and packet forwarding in innovative ways.

Application protocols are a curious lot. In many ways, the traditional client/server applications are like another layer of transport protocol, except they have application-specific knowledge built into them. You could argue that they are just specialized transport protocols, and that transport protocols get layered on top of each other until producing the precise service needed by the application. Similarly, the overlay and

peer-to-peer protocols can be viewed as providing an alternative routing infrastructure, but again, one that is tailored for a particular application's needs. One sure lesson we draw from this observation is that designing application-level protocols is really no different than designing core network protocols, and that the more one understands about the latter, the better they will do designing the former. We also observe that the systems approach—understanding how functions and components interact to build a complete system—applies at least as much in the design of applications as in any other aspect of networking.

OPEN ISSUE
New Network Architecture

It's difficult to put a finger on a specific open issue in the realm of application protocols—the entire field is open as new applications are invented every day, and the networking needs of these applications are, well, application dependent. The real challenge to network designers is to recognize that what applications need from the network changes over time, and these changes drive the transport protocols we develop and the functionality we put into network routers.

Developing new transport protocols is a reasonably tractable problem. You may not be able to get the IETF to bless your transport protocol as an equal of TCP or UDP, but there's certainly nothing stopping you from designing the world's greatest multimedia application that comes bundled with a new end-to-end protocol that runs on top of UDP, much like what happens with RTP.

On the other hand, pushing application-specific knowledge into the middle of the network—into the routers—is a much more difficult problem. This is because in order to effect a particular application, any new network service or functionality may need to be loaded into many, if not all, of the routers in the Internet. Overlay networks provide a way of introducing new functionality into the network without cooperation of all (or even any) of the routers, but in the long run, we can expect the underlying network architecture will need to change to accommodate these overlays. We saw this issue with RON—how RON and BGP route selection interact with each other—and can expect it to be a general question as overlay networks become more prevalent.

One possibility is that an alternative *fixed* architecture does not evolve, but instead, the next network architecture will be highly adaptive. In the limit, rather than defining an infrastructure for carrying data packets, the network architecture might allow packets to carry both data and code (or possibly pointers to code) that tell the routers how it should process the packet. Such a network raises a host of issues, not the least of which is

how to enforce security in a world where arbitrary applications can effectively program routers.

FURTHER READING

Our first article provides an interesting perspective on the early design and implementation of the World Wide Web, written by its inventors before it had taken the world by storm. The development of DNS is well described by Mockapetris and Dunlap. Although overlays and peer-to-peer networks are still an emerging area, the last six research papers provide a good place to start understanding the issues.

- Berners-Lee, T., R. Caillia, A. Luotonen, H. Nielsen, and A. Secret. "The World Wide Web." *Communications of the ACM* 37(8), pp. 76–82, 1994.
- Mockapetris, P., and K. Dunlap. "Development of the Domain Name System." *Proceedings of the SIGCOMM '88 Symposium*, pp. 123–133, August 1988.
- Karger, D., et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web." *Proceedings of the ACM Symposium on Theory of Computing*, pp. 654–663, 1997.
- Chu, Y., S. Rao, and H. Zhang. "A Case for End System Multicast." *Proceedings of the ACM SIGMETRICS '00 Conference*, pp. 1–12, June 2000.
- Andersen, D., et al. "Resilient Overlay Networks." *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 131–145, October 2001.
- Rowstron, A., and P. Druschel. "Storage Management and Caching in PAST, a Large-Scale Persistent Peer-to-Peer Storage Utility." *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 188–201, October 2001.
- Stoica, I., et al. "Chord: A Peer-to-Peer Lookup Service for Internet Applications." *Proceedings of the ACM SIGCOMM Conference*, September 2001.
- Ratnasamy, S., et al. "A Scalable Content-Addressable Network." *Proceedings of ACM SIGCOMM '01*, August 2001.

SMTP was originally defined in RFC 821 [Pos82], and of course, RFC 822 is RFC 822 [Cro82]. They have been, in IETF terminology, "obsoleted" by [Kle01] and [Res01], respectively. MIME is defined in a series of RFCs; the original specification was in RFC 1521 [BF93] and the most recent version is defined in RFC 2045 [FB96].

Version 1.0 of HTTP is specified in RFC 1945 [BLFF96], and the latest version (1.1) is defined in RFC 2616 [FGM+99]. There are a wealth of papers written about

web performance, especially web caching. A good example is a paper by Danzig on web traffic and its implications on the effectiveness of caching [Dan98]. Roy Fielding's Ph.D. thesis [Fie00] is the ultimate reference for REST.

There are a wealth of papers on naming, as well as on the related issue of resource discovery (finding out what resources exist in the first place). General studies of naming can be found in Terry [Ter86], Comer and Peterson [CP89], Birrell et al. [BLNS82], Saltzer [Sal78], Shoch [Sho78], and Watson [Wat81]; attribute-based (descriptive) naming systems are described in Peterson [Pet88] and Bowman et al. [BPY90]; and resource discovery is the subject of Bowman et al. [BDMS94].

Network management is a sufficiently large and important field that the IETF devotes an entire area to it. There are well over 100 RFCs describing various aspects of SNMP and MIBs. The two key references, however, are RFC 2578 [MPS99], which defines the structure of management information for version 2 of SNMP (SNMPv2), and RFC 3416 [Pre02], which defines the protocol operations for SNMPv2. Many of the other SNMP/MIB-related RFCs define extensions to the core set of MIB variables—for example, variables that are specific to a particular network technology or to a particular vendor's product. Perkins et al. [PM97] provides a good introduction to SNMP and MIBS.

SIP is defined in RFC 3261 [SCJ+02], which contains a helpful tutorial section as well as the detailed specification of the protocol.

The National Research Council report on the ossification of the Internet can be found in [NRC01], and a proposal to use overlay networks to introduce disruptive technology was made by Peterson, Anderson, Culler, and Roscoe [PACR02]. The original case for overriding BGP routes is made by Savage, Collins, Hoffman, Snell, and Anderson [SCH+99]. The idea of using DNS to load-balance a set of servers is described in RFC 1794 [Bri95]. The Cache Array Routing Protocol (CARP) is defined in an Internet Draft [CPVR97]. A comprehensive treatment of the issue of web caching versus replicated servers can be found in Rabinovich and Spatscheck's book [RS02]. Wang, Pai, and Peterson explore the design space for redirectors [WPP02].

Finally, we recommend the following live reference to help keep tabs on the rapid evolution of the Web and for a wealth of information related to web services:

- http://www.w3.org: World Wide Web Consortium.

EXERCISES

1 ARP and DNS both depend on caches; ARP cache entry lifetimes are typically 10 minutes, while DNS cache is on the order of days. Justify this difference.

What undesirable consequences might there be in having too long a DNS cache entry lifetime?

2 IPv6 simplifies ARP out of existence by allowing hardware addresses to be part of the IPv6 address. How does this complicate the job of DNS? How does this affect the problem of finding your local DNS server?

3 DNS servers also allow reverse lookup; given an IP address 128.112.169.4, it is reversed into a text string 4.169.112.128.in-addr.arpa and looked up using DNS PTR records (which form a hierarchy of domains analogous to that for the address domain hierarchy). Suppose you want to authenticate the sender of a packet based on its host name and are confident that the source IP *address* is genuine. Explain the insecurity in converting the source address to a name as above and then comparing this name to a given list of trusted hosts. (Hint: Whose DNS servers would you be trusting?)

4 What is the relationship between a domain name (e.g., cs.princeton.edu) and an IP subnet number (e.g., 192.12.69.0)? Do all hosts on the subnet have to be identified by the same name server? What about reverse lookup, as in the previous exercise?

5 Suppose a host elects to use a name server not within its organization for address resolution. When would this result in no more total traffic, for queries not found in any DNS cache, than with a local name server? When might this result in a better DNS cache hit rate and possibly less total traffic?

6 Figure 9.7 shows the hierarchy of name servers. How would you represent this hierarchy if one name server served multiple zones? In that setting, how does the name server hierarchy relate to the zone hierarchy? How do you deal with the fact that each zone may have multiple name servers?

7 Use the whois utility/service to find out who is in charge of your site, at least as far as the InterNIC is concerned. Look up your site both by DNS name and by IP network number; for the latter you may have to try an alternative whois server (e.g., whois -h whois.arin.net...). Try princeton.edu and cisco.com as well.

8 Many smaller organizations have their websites maintained by a third party. How could you use whois to find if this is the case, and, if so, the identity of the third party?

9 One feature of the existing DNS .com hierarchy is that it is extremely "wide."

(a) Propose a more hierarchical reorganization of the .com hierarchy. What objections might you foresee to your proposal's adoption?

(b) What might be some of the consequences of having most DNS domain names contain four or more levels, versus the two of many existing names?

10 Suppose, in the other direction, we abandon any pretense at all of DNS hierarchy, and simply move all the .com entries to the root name server: www.cisco.com would become www.cisco, or perhaps just cisco. How would this affect root name server traffic in general? How would this affect such traffic for the specific case of resolving a name like cisco into a web server address?

11 What DNS cache issues are involved in changing the IP address of, say, a web server host name? How might these be minimized?

12 Take a suitable DNS-lookup utility (e.g., dig) and disable the recursive lookup feature (e.g., with +norecursive), so that when your utility sends a query to a DNS server, and that server is unable to fully answer the request from its own records, the server sends back the next DNS server in the lookup sequence rather than automatically forwarding the query to that next server. Then carry out manually a name lookup such as that in Figure 9.8; try the host name www.cs.princeton.edu. List each intermediate name server contacted. You may also need to specify that queries are for NS records rather than the usual A records.

13 Discuss how you might rewrite SMTP or HTTP to make use of a hypothetical general-purpose request/reply protocol. Could an appropriate analog of persistent connections be moved from the application layer into such a transport protocol? What other application tasks might be moved into this protocol?

14 Most Telnet clients can be used to connect to port 25, the SMTP port, instead of to the Telnet port. Using such a tool, connect to an SMTP server and send yourself (or someone else, with permission) some forged email. Then examine the headers for evidence the message isn't genuine.

15 What features might be used by (or added to) SMTP and/or a mail daemon such as sendmail to provide some resistance to email forgeries as in the previous exercise?

16 Find out how SMTP hosts deal with unknown commands from the other side, and how in particular this mechanism allows for the evolution of the protocol (e.g., to "extended SMTP"). You can either read the RFC or contact an SMTP server as in Exercise 14 and test its responses to nonexistent commands.

17 As presented in the text, SMTP involves the exchange of several small messages. In most cases, the server responses do not affect what the client sends subsequently. The client might thus implement *command pipelining*: sending multiple commands in a single message.

(a) For what SMTP commands *does* the client need to pay attention to the server's responses?

(b) Assume the server reads each client message with gets() or the equivalent, which reads in a string up to a <LF>. What would it have to do even to detect that a client had used command pipelining?

(c) Pipelining is nonetheless known to break with some servers; find out how a client can negotiate its use.

18 Find out what other features DNS MX records provide in addition to supplying an alias for a mail server; the latter could, after all, be provided by a DNS CNAME record. MX records are provided to support email; would an analogous web record be of use in supporting HTTP?

19 One of the central problems faced by a protocol such as MIME is the vast number of data formats available. Consult the MIME RFC to find out how MIME deals with new or system-specific image and text formats.

20 MIME supports multiple representations of the same content using the multipart/alternative syntax; for example, text could be sent as text/plain, text/richtext, and application/postscript. Why do you think plaintext is supposed to be the *first* format, even though implementations might find it easier to place plaintext after their native format?

21 Consult the MIME RFC to find out how base64 encoding handles binary data of a length not evenly divisible by three bytes.

22 In HTTP version 1.0, a server marked the end of a transfer by closing the connection. Explain why, in terms of the TCP layer, this was a problem for servers. Find out how HTTP version 1.1 avoids this. How might a general-purpose request/reply protocol address this?

23 Find out how to configure an HTTP server so as to eliminate the **404 not found** message, and have a default (and hopefully friendlier) message returned instead. Decide if such a feature is part of the protocol or part of an implementation, or is technically even permitted by the protocol. (Documentation for the **apache** HTTP server can be found at **www.apache.org**.)

24 Why does the HTTP GET command

```
GET http://www.cs.princeton.edu/index.html HTTP/1.1
```

contain the name of the server being contacted? Wouldn't the server already know its name? Use Telnet, as in Exercise 14, to connect to port 80 of an HTTP server and find out what happens if you leave the host name out.

25 When an HTTP server initiates a **close()** at its end of a connection, it must then wait in TCP state FIN_WAIT_2 for the client to close the other end. What mechanism within the TCP protocol could help an HTTP server deal with noncooperative or poorly implemented clients that don't close from their end? If possible, find out about the programming interface for this mechanism, and indicate how an HTTP server might apply it.

26 The POP3 Post Office Protocol only allows a client to retrieve email, using a password for authentication. Traditionally, to *send* email a client would simply send it to its server and expect that it be relayed.

(a) Explain why email servers often no longer permit such relaying from arbitrary clients.

(b) Propose an SMTP option for remote client authentication.

(c) Find out what existing methods are available for addressing this issue.

27 Suppose a very large website wants a mechanism by which clients access whichever of multiple HTTP servers is "closest" by some suitable measure.

(a) Discuss developing a mechanism within HTTP for doing this.

(b) Discuss developing a mechanism within DNS for doing this.

Compare the two. Can either approach be made to work without upgrading the browser?

28 Find out if there is available to you an SNMP node that will answer queries you send it. If so, locate some SNMP utilities (e.g., the **ucd-snmp** suite) and try the following:

(a) Fetch the entire **system** group, using something like

```
snmpwalk nodename public system
```

Also try the above with **1** in place of **system**.

(b) Manually walk through the **system** group, using multiple SNMP **GETNEXT** operations (e.g., using **snmpgetnext** or equivalent), retrieving one entry at a time.

29 Using the SNMP device and utilities of the previous exercise, fetch the **tcp** group (numerically group **6**), or some other group. Then do something to make some of the group's counters change, and fetch the group again to show the change. Try to do this in such a way that you can be sure your actions were the cause of the change recorded.

30 What information provided by SNMP might be useful to someone planning the IP spoofing attack of Exercise 17 in Chapter 5? What other SNMP information might be considered sensitive?

31 Application protocols such as FTP and SMTP were designed from scratch, and they seem to work reasonably well. What is it about business-to-business and Enterprise Application Integration protocols that calls for a Web Services protocol framework?

32 Choose a web service with equivalent REST and SOAP interfaces, such as those offered by Amazon.com. Compare how equivalent operations are implemented in the two styles.

33 Get the WSDL for some SOAP-style web service and choose an operation. In the messages that implement that operation, identify the fields.

34 Suppose some receivers in a large conference can receive data at a significantly higher bandwidth than others. What sorts of things might be implemented to address this? (Hint: Consider both the session announcement protocol (SAP) and the possibility of utilizing third-party mixers.)

35 How might you encode audio (or video) data in two packets so that if one packet is lost, then the resolution is simply reduced to what would be expected

with half the bandwidth? Explain why this is much more difficult if a JPEG-type encoding is used.

36 Explain the relationship between uniform resource locators (URLs) and uniform resource identifiers (URIs). Given an example of a URI that is *not* a URL.

37 What problem would a DNS-based redirection mechanism encounter if it wants to select an appropriate server based on current load information?

38 Consider the following simplified BitTorrent scenario. There is a swarm of 2^n peers and, during the time in question, no peers join or leave the swarm. It takes a peer 1 unit of time to upload or download a piece, during which time it can only do one or the other. Initially one peer has the whole file and the others have nothing.

(a) If the swarm's target file consists of only 1 piece, what is the minimum time necessary for all the peers to obtain the file? Ignore all but upload/download time.

(b) Let x be your answer to the preceding question. If the swarm's target file instead consisted of 2 pieces, would it be possible for all the peers to obtain the file in less than $2x$ time units? Why or why not?

SOLUTIONS TO SELECT EXERCISES

Chapter 1

6. We will count the transfer as completed when the last data bit arrives at its destination.

(a) 1.5 MB = 12,582,912 bits. 2 initial RTT's (160 ms) + 12,582,912/10,000,000 bps (transmit) + RTT/2 (propagation) $\approx$ 1.458 seconds.

(b) Number of packets required = 1.5 MB/1 KB = 1,536. To the above we add the time for 1,535 RTTs (the number of RTTs between when packet 1 arrives and packet 1,536 arrives), for a total of $1.458 + 122.8 = 124.258$ seconds.

(c) Dividing the 1,536 packets by 20 gives 76.8. This will take 76.5 RTTs (half an RTT for the first batch to arrive, plus 76 RTTs between the first batch and the 77th partial batch), plus the initial 2 RTTs, for 6.28 seconds.

(d) Right after the handshaking is done we send one packet. One RTT after the handshaking we send two packets. At n RTTs past the initial handshaking we have sent $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1$ packets. At $n = 10$ we have thus been able to send all 1,536 packets; the last batch arrives 0.5 RTT later. Total time is 2 + 10.5 RTTs, or 1 second.

8. Propagation delay is 50×10^3 m/(2×10^8 m/sec) = 250 μs 800 bits/250 μs is 3.2 Mbits/sec. For 512-byte packets, this rises to 16.4 Mbit/sec.

16. (a) Propagation delay on the link is $(55 \times 10^9)/(3 \times 10^8) = 184$ seconds. Thus the RTT is 368 seconds.

(b) The delay $\times$ bandwidth product for the link is $= 184 \times 128 \times 10^3 =$ 2.81 MB.

(c) After a picture is taken it must be transmitted on the link, and be completely propagated before Mission Control can interpret it. Transmit delay for 5 MB of data is 41,943,040 bits/$128 \times 10^3 = 328$ seconds. Thus, the total time required is transmit delay + propagation delay = 328 + 184 = 512 seconds.

19. (a) For each link, it takes 1 Gbps/5 kb = 5 μs to transmit the packet on the link, after which it takes an additional 10 μs for the last bit to propagate across the link. Thus, for a LAN with only one switch that starts forwarding only after receiving the whole packet, the total transfer delay is two transmit delays + two propagation delays = 30 μs.

(b) For three switched and thus four links, the total delay is four transmit delays + four propagation delays = 60 μs.

(c) For "cut-through," a switch need only decode the first 128 bits before beginning to forward. This takes 128 ns. This delay replaces the switch transmit delays in the previous answer for a total delay of one transmit delay + three cut-through decoding delays + four propagation delays = 45.384 μs.

29. (a) $1{,}920 \times 1{,}080 \times 24 \times 30 = 1{,}492{,}992{,}000 \approx 1.5$ Gbps.

(b) $8 \times 8{,}000 = 64$ Kbps.

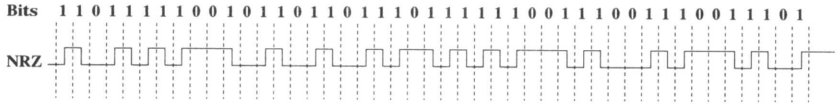
(c) $260 \times 50 = 13$ Kbps.

(d) $24 \times 88{,}200 = 216{,}800 \approx 2.1$ Mbps.

Chapter 2

3. The 4B/5B encoding of the given bit sequence is the following.

11011 11100 10110 11011 10111 11100 11100 11101



7. Let $\wedge$ mark each position where a stuffed 0 bit was removed. There was one error where the sever consecutive 1s are detected (***err***). At the end of the bit sequence, the end of frame was detected (***eof***).

$$01101011111 \wedge 101001111111_{err}0\ 110\ \underline{01111110}_{eof}$$

19. (a) We take the message 1011 0010 0100 1011, append 8 zeros and divide by 1 0000 0111 ($x^8 + x^2 + x^1 + 1$). The remainder is 1001 0011. We transmit the original message with this remainder appended, resulting in 1011 0010 0100 0011 1001 0011.

(b) Inverting the first bit gives 0011 0010 0100 1011 1001 0011. Dividing by 1 0000 0111 ($x^8 + x^2 + x^1 + 1$) gives a a remainder of 1011 0110.

25. One-way latency of the link is 100 msec. (Bandwidth) × (roundtrip delay) is about 125 pps × 0.2 sec, or 25 packets. SWS should be this large.

(a) If RWS = 1, the necessary sequence number space is 26. Therefore, 5 bits are needed.

(b) If RWS = SWS, the sequence number space must cover twice the SWS, or up to 50. Therefore, 6 bits are needed.

32. Figure 1 gives the timeline for the first case. The second case reduces the total transaction time by roughly 1 RTT.

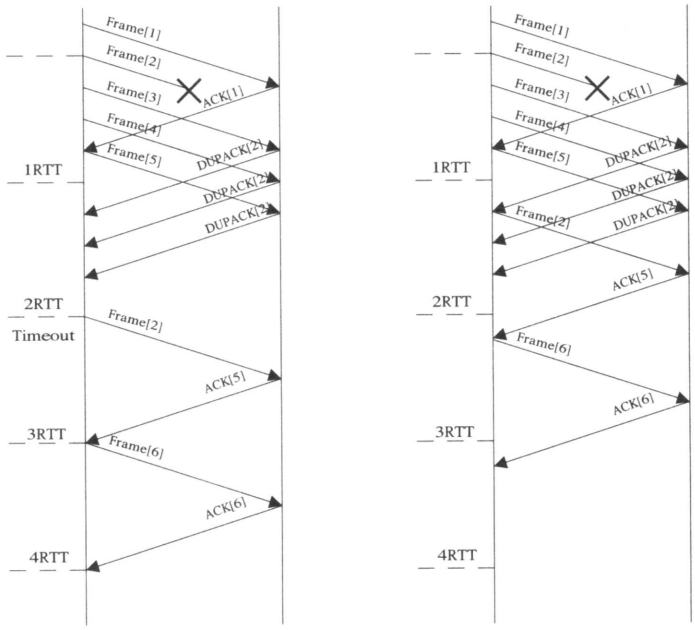


Figure 1

Chapter 3

2. See Table 1. This table is cumulative; at each part the VCI tables consist of the entries at that part and also all previous entries.

Exercise Part	Switch	Input		Output	
		Port	VCI	Port	VCI
(a)	1	0	0	1	0
	2	3	0	1	0
	4	3	0	0	0
(b)	2	0	0	1	1
	3	3	0	0	0
	4	3	1	1	0
(c)	1	1	1	2	0
	2	1	2	3	1
	4	2	0	3	2
(d)	1	1	2	3	0
	2	1	3	3	2
	4	0	1	3	3
(e)	2	0	1	2	0
	3	2	0	0	1
(f)	2	1	4	0	2
	3	0	2	1	0
	4	0	2	3	4

Table 1

14. The following list shows the mapping between LANs and their designated bridges.

B1 dead[B7]

B2 A,B,D

B3 E,F,G,H

B4 I

B5 idle

B6 J

B7 C

16. All bridges see the packet from D to C. Only B3, B2, and B4 see the packet from C to D. Only B1, B2, and B3 see the packet from A to C.

B1 A-interface : A B2-interface : D (not C)

B2 B1-interface : A B3-interface : C B4-interface : D

B3 C-interface : C B2-interface : A,D

B4 D-interface : D B2-interface : C (not A)

33. Since the I/O bus speed is less than the memory bandwidth, it is the bottleneck. Effective bandwidth that the I/O bus can provide is 1,000/2 Mbps because each packet crosses the I/O bus twice. Therefore, the number of interfaces is $\lfloor 500/45 \rfloor = 11$.

Chapter 4

5. By definition, Path MTU is 512 bytes. Maximum IP payload size is $512 - 20 = 492$ bytes. We need to transfer $2{,}048 + 20 = 2{,}068$ bytes in the IP payload. This would be fragmented into 4 fragments of size 492 bytes and 1 fragment of size 100 bytes. There are 5 packets in total if we use Path MTU. In the previous setting we needed 7 packets.

16. (a) See Table 2.

(b) See Table 3.

Information Stored at Node	Distance to Reach Node					
	A	B	C	D	E	F
A	0	2	∞	5	∞	∞
B	2	0	2	∞	1	∞
C	∞	2	0	2	∞	3
D	5	∞	2	0	∞	∞
E	∞	1	∞	∞	0	3
F	∞	∞	3	∞	3	0

Table 2

Information Stored at Node	Distance to Reach Node					
	A	B	C	D	E	F
A	0	2	4	5	3	∞
B	2	0	2	4	1	4
C	4	2	0	2	3	3
D	5	4	2	0	∞	5
E	3	1	3	∞	0	3
F	∞	4	3	5	3	0

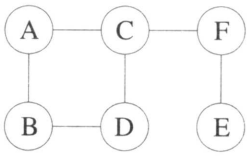
Table 3

Information Stored at Node	Distance to Reach Node					
	A	B	C	D	E	F
A	0	2	4	5	3	6
B	2	0	2	4	1	4
C	4	2	0	2	3	3
D	5	4	2	0	5	5
E	3	1	3	5	0	3
F	6	4	3	5	3	0

Table 4

(c) See Table 4.

19. The following is an example network topology.



22. Apply each subnet mask and if the corresponding subnet number matches the SubnetNumber column, then use the entry in Next-Hop.

(a) Applying the subnet mask 255.255.254.0, we get 128.96.170.0. Use interface 0 as the next hop.

(b) Applying subnet mask 255.255.254.0, we get 128.96.166.0 (next hop is Router 2). Applying subnet mask 255.255.252.0, we get 128.96.164.0 (next hop is Router 3). However, 255.255.254.0 is a longer prefix. Use Router 2 as the next hop.

(c) None of the subnet number entries match, hence use default Router R4.

(d) Applying subnet mask 255.255.254.0, we get 128.96.168.0. Use interface 1 as the next hop.

(e) Applying subnet mask 255.255.252.0, we get 128.96.164.0. Use Router 3 as the next hop.

29. See Table 5.

46. (a): F (b): B (c): E (d): A (e): D (f): C

58. Figure 2 illustrate the multicast trees for sources D and E.

Step	Confirmed	Tentative
1	(A,0,-)	
2	(A,0,-)	(B,1,B) (D,5,D)
3	(A,0,-) (B,1,B)	(D,4,B) (C,7,B)
4	(A,0,-) (B,1,B) (D,4,B)	(C,5,B) (E,7,B)
5	(A,0,-) (B,1,B) (D,4,B) (C,5,B)	(E,6,B)
6	(A,0,-) (B,1,B) (D,4,B) (C,5,B) (E,6,B)	

Table 5

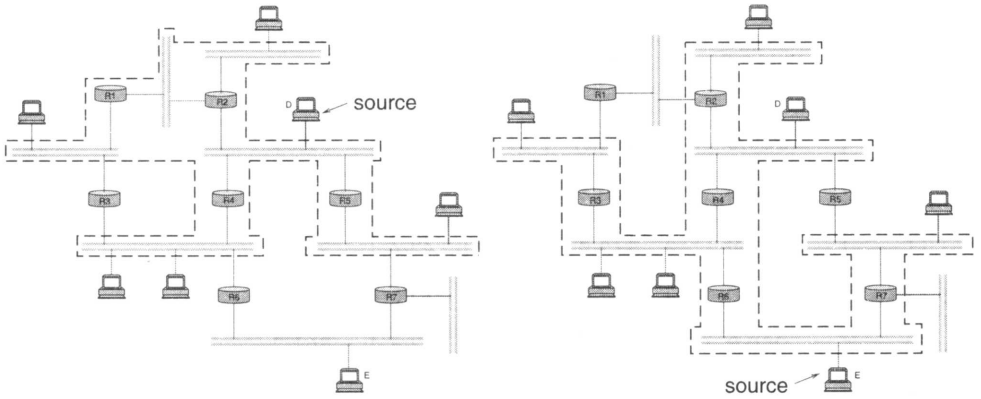


Figure 2

Chapter 5

10. The advertised window should be large enough to keep the pipe full; delay (RTT) × bandwidth here is 140 ms × 1 Gbps = 10 Mb = 17.5 MB of data. This requires 25 bits ($2^{25} = 33{,}554{,}432$) for the AdvertisedWindow field. The sequence number field must not wrap around in the maximum segment lifetime. In 60 seconds, 7.5 GB can be transmitted. 33 bits allows a sequence space of 8.6 GB, and so will not wrap in 60 seconds.

13. (a) 2^{32} B/(5 GB) = 859 ms.

(b) 1,000 ticks in 859 ms is once each 859 μs indicating wraparound in 3.7 Ms or approximately 43 days.

27. Using initial Deviation = 50 it took 20 iterations for TimeOut to fall below 300.0. See Table 6.

Iteration	SampleRTT	EstRTT	Dev	diff	TimeOut
0	200.0	90.0	50.0		
1	200.0	103.7	57.5	110.0	333.7
2	200.0	115.7	62.3	96.3	364.9
3	200.0	126.2	65.0	84.3	386.2
4	200.0	135.4	66.1	73.8	399.8
5	200.0	143.4	66.0	64.6	407.4
6	200.0	150.4	64.9	56.6	410.0
7	200.0	156.6	63.0	49.6	408.6
8	200.0	162.0	60.6	43.4	404.4
9	200.0	166.7	57.8	38.0	397.9
10	200.0	170.8	54.8	33.3	390.0
11	200.0	174.4	51.6	29.2	380.8
12	200.0	177.6	48.4	25.6	371.2
13	200.0	180.4	45.2	22.4	361.2
14	200.0	182.8	42.0	19.6	350.8
15	200.0	184.9	38.9	17.2	340.5
16	200.0	186.7	36.0	15.1	330.7
17	200.0	188.3	33.2	13.3	321.1
18	200.0	189.7	30.6	11.7	312.1
19	200.0	190.9	28.1	10.3	303.3
20	200.0	192.0	25.8	9.1	295.2

Table 6

Chapter 6

11. (a) First we calculate the finishing times F_i. We don't need to worry about clock speed here since we may take $A_i = 0$ for all the packets. F_i thus becomes just the cumulative per-flow size, that is, $F_i = F_{i-1} + P_i$. See

Packet	Size	Flow	F_i
1	200	1	200
2	200	1	400
3	160	2	160
4	120	2	280
5	160	2	440
6	210	3	210
7	150	3	360
8	90	3	450

Table 7

Table 7. We now send in increasing order of F_i: Packet 3, Packet 1, Packet 6, Packet 4, Packet 7, Packet 2, Packet 5, Packet 8.

(b) To give flow 1 a weight of 2 we divide each of its F_i by 2, that is, $F_i = F_{i-1} + P_i/2$; To give flow 2 a weight of 4 we divide each of its F_i by 4, that is, $F_i = F_{i-1} + P_i/4$; To give flow 3 a weight of 3 we divide each of its F_i by 3, that is, $F_i = F_{i-1} + P_i/3$; again we are using the fact that there is no waiting. See Table 8. Transmitting in increasing order of the weighted F_i we send as follows: Packet 3, Packet 4, Packet 6, Packet 1, Packet 5, Packet 7, Packet 8, Packet 2.

Packet	Size	Flow	Weighted F_i
1	200	1	100
2	200	1	200
3	160	2	40
4	120	2	70
5	160	2	110
6	210	3	70
7	150	3	120
8	90	3	150

Table 8

15. (a) For the ith arriving packet on a given flow we calculate its estimated finishing time F_i by the formula $F_i = \max\{A_i, F_{i-1}\} + 1$, where the clock used to measure the arrival times A_i runs slow by a factor equal to the number of active queues. The A_i clock is global; the sequence of F_i's calculated as above is local to each flow.

Table 9 lists all events by wallclock time. We identify packets by their flow and arrival time; thus, packet A4 is the packet that arrives on flow A at wallclock time 4, that is, the third packet. The last three columns are the queues for each flow for the subsequent time interval, *including* the packet currently being transmitted. The number of such active queues determines the amount by which A_i is incremented on the subsequent line. Multiple packets appear on the same line if their F_i values are all the same; the F_i values are in italic when $F_i = F_{i-1} + 1$ (versus $F_i = A_i + 1$).

Wallclock	A_i	Arrivals	F_i	Sent	A's Queue	B's Queue	C's Queue
1	1.0	A1,B1,C1	2.0	A1	A1	B1	C1
2	1.333	C2	*3.0*	B1		B1	C1,C2
3	1.833	A3	*3.0*	C1	A3		C1,C2
4	2.333	B4	3.333	A3	A3	B4	C2,C4
		C4	*4.0*				
5	2.666	A5	*4.0*	C2	A5	B4	C2,C4
6	3.0	A6	*5.0*	B4	A5,A6	B4	C4,C6
		C6	*5.0*				
7	3.333	B7	4.333	A5	A5,A6	B7	C4,C6,C7
		C7	*6.0*				
8	3.666	A8	*6.0*	C4	A6,A8	B7,B8	C4,C6,C7
		B8	*5.333*				
9	4	A9	*7.0*	B7	A6,A8,A9	B7,B8,B9	C6,C7
		B9	*6.333*				
10	4.333			A6	A6,A8,A9	B8,B9	C6,C7

Table 9

Wallclock	A_i	Arrivals	F_i	Sent	A's Queue	B's Queue	C's Queue
11	4.666	A11	*8.0*	C6	A8,A9,A11	B8,B9	C7
12	5	C12	*7.0*	B8	A8,A9,A11	B8,B9	C7,C12
13	5.333	B13	*7.333*	A8	A8,A9,A11	B9,B13	C7,C12
14	5.666			C7	A9,A11	B9,B13	C7,C12
15	6.0	B15	*8.333*	B9	A9,A11	B9,B13,B15	C12
16	6.333			A9	A9,A11	B13,B15	C12
17	6.666			C12	A11	B13,B15	C12
18	7			B13	A11	B13,B15	
19	7.5			A11	A11	B15	
20	8			B15		B15	

Table 9 (Continued).

(b) For weighted fair queuing we have, for flow B,

$$F_i = max\{A_i, F_{i-1}\} + 0.5$$

For flows A and C, F_i is as before. Table 10 corresponds to Table 9.

35. (a) We have

$$\mathsf{TempP} = \mathsf{MaxP} \times \frac{\mathsf{AvgLen} - \mathsf{MinThreshold}}{\mathsf{MaxThreshold} - \mathsf{MinThreshold}}$$

AvgLen is halfway between MinThreshold and MaxThreshold, which implies that the fraction here is $1/2$ and so $\mathsf{TempP} = \mathsf{MaxP}/2 = p/2$. We now have

$$\mathsf{P}_{\mathsf{count}} = \mathsf{TempP}/(1 - \mathsf{count} \times \mathsf{TempP}) = 1/(x - \mathsf{count}),$$

where $x = 2/p$. Therefore,

$$1 - \mathsf{P}_{\mathsf{count}} = \frac{x - (\mathsf{count} + 1)}{x - \mathsf{count}}$$

Evaluating the product

$$(1 - \mathsf{P}_1) \times \cdots \times (1 - \mathsf{P}_n)$$

Wallclock	A_i	Arrivals	F_i	Sent	A's Queue	B's Queue	C's Queue
1	1.0	A1,C1	2.0	B1	A1	B1	C1
		B1	1.5				
2	1.333	C2	*3.0*	A1			C1,C2
3	1.833	A3	*3.0*	C1	A1		C1,C2
4	2.333	B4	2.833	B4	A3	B4	C2,C4
		C4	*4.0*				
5	2.666	A5	*4.0*	A3	A3,A5		C2,C4
6	3.166	A6	*5.0*	C2	A5,A6		C2,C4,C6
		C6	*5.0*				
7	3.666	B7	4.167	A5	A5,A6	B7	C4,C6,C7
		C7	*6.0*				
8	4.0	A8	*6.0*	C4	A6,A8	B7,B8	C6,C7
		B8	*4.666*				
9	4.333	A9	*7.0*	B7	A6,A8,A9	B7,B8,B9	C6,C7
		B9	*5.166*				
10	4.666			B8	A6,A8,A9	B8,B9	C6,C7
11	5.0	A11	*8.0*	A6	A6,A8,A9,A11	B9	C6,C7
12	5.333	C12	*7.0*	C6	A8,A9,A11	B9	C6,C7,C12
13	5.666	B13	*6.166*	B9	A8,A9,A11	B9,B13	C7,C12
14	6.0			A8	A9,A11	B13	C7,C12
15	6.333	B15	*6.833*	C7	A9,A11	B13,B15	C12
16	6.666			B13	A9,A11	B13,B15	C12
17	7.0			B15	A11	B15	C12
18	7.333			A9	A11		C12
19	7.833			C12	A11		C12
20	8.333			A11	A11		

Table 10