6
Performance Tuning

Car manufacturers often develop for the mass market, and strive to develop a "best fit" product that will be acceptable to the majority of customers. The product (in this case a car) will operate flawlessly for 90% of the people and give years of service. Sometimes, customers wish to tune their cars to work better under specific operating conditions. This may include the addition of snow tires, a supercharger or an engine additive. This optimisation will ensure that the car delivers the best performance for the customer relative to their environment. Networking hardware and software are a bit like this. Most times the default settings will be optimal, but there will be times when some optimisation is required to get the best performance for your specific environment.

The key to optimisation is understanding why you are optimising in the first place, and knowing the ramifications of the changes you are making. It was C.A.R. Hoare that said, "Premature optimization is the root of all evil in programming." If you upgrade your car by installing a massive supercharger because you are having trouble getting around corners, you are probably not going to achieve your goal. The car will go faster, but you will probably have an accident on your first corner when your woefully inadequate suspension decides to quit. If you understand the problem first, that the bad cornering was caused by poor suspension and not by engine output, then you can avoid an unnecessary (and expensive) upgrade as well as a potentially bad accident. There are a couple of things to remember when embarking on any optimisation:

1. **Practice change control.** Make backups of any configuration files you alter, so you can revert to an older version should it prove necessary.
2. **Take a baseline of the performance before and after changing any settings.** How else will you know if your settings have had a positive effect? This also may not be an exact numeric measurement, but could be the results of an operation. For example, if a user gets an error message when

they attempt to do something, making a change should make the error go away, or at least change to something more informative.

3. **Make sure you understand the problem you are trying to address.** If it is unclear precisely where the problem lies, it is even more important to make backups before you make any significant changes.
4. **Document the change.** This helps others understand why a change was made. Keeping a system change log can help you build a bigger picture of the state of the system, and may indicate long-term problems before they occur.
5. **Don't make a change unless you have a reason to.** If everything is working well enough that utilisation is acceptable and the users are happy, then why make changes?

There are system administrators who will swear blind that a particular optimisation change has fixed a problem, but they often cannot give actual proof when asked. This is usually because they do not understand what they have done or cannot measure the effect of the change. Defaults are there for a reason, so unless you have a valid reason to change the defaults, don't do it!

Of course, even when you fully understand the nature of the problem and the limitations of the system you are working with, optimisation can only take you so far. When you have exhausted optimisation techniques, it may be time to upgrade your hardware or software.

Squid cache optimisation

The default Squid settings are sufficient for most smaller networks. While these settings may work well for many installations, maximum performance can be achieved in large installations by making some changes to the defaults. Since software authors cannot know ahead of time how aggressively Squid may use system resources, the default settings are intentionally conservative. By fully optimising Squid to fit your server hardware, you can make the best possible use of the server resources and squeeze the maximum performance from your network.

Of course, no amount of configuration tweaking can help Squid run on hardware that simply can't handle the network load. Some of the parameters that may need to be changed to match your particular network include the server hardware itself, the disk cache and memory cache sizes, and even ACL lists. You will also need to know when (and how) to use multiple caches effectively when a single caching server simply isn't enough. This section will show you how to make the best possible caching solution for your network.

Squid optimisation is, of course, a complex subject in itself. For a more detailed explanation of this topic we suggest you refer to Duane Wessels' excellent book, *Squid: The Definitive Guide*.

Cache server hardware

The type of hardware you need to dedicate to your Squid cache depends on the amount of traffic that flows through it. Even moderately busy networks require little more than a typical desktop PC for use as the proxy server. Monitoring your CPU utilisation over time will help you to determine if you need to upgrade the processing power of your cache server. Squid runs as a single process, so using a multiprocessor machine will not give you much benefit. In terms of disks, the cache should ideally reside on a separate disk with its own data channel (e.g. the secondary IDE master with no slave on the same channel, or on its own SCSI or SATA bus) to give you the greatest performance benefit. It is very common to run the operating system on one disk and install the cache on another disk. This way, regular system activities (such as writing to the log files) do not interfere with access to the cache volumes. You will get better performance with a fast disk of adequate size, rather than a large disk with slower access times.

If you use RAID in your server, installing your cache on a RAID0 or RAID1 is acceptable. Squid already spreads the data load across multiple disks, so there is really no advantage to using RAID0. You definitely want to avoid using RAID5 for your cache. From the Squid FAQ:

> *Squid is the worst case application for RAID5, whether hardware or software, and will absolutely kill the performance of a RAID5. Once the cache has been filled Squid uses a lot of small random writes which [is] the worst case workload for RAID5, effectively reducing write speed to only little more than that of one single drive.*
>
> *Generally seek time is what you want to optimise for Squid, or more precisely the total amount of seeks/s your system can sustain. Choosing the right RAID solution generally decreases the amount of seeks/s your system can sustain significantly.*

You can still place the operating system (or other volumes) on a RAID 5 disk set for fault tolerance. Simply install the cache volumes on their own disks outside the RAID. The cache volumes are very quick to re-create in the case of a disk failure.

Squid also loves to use system RAM, so the more memory that is installed, the faster it will run. As we will see, the amount of RAM required increases as your disk cache size increases. We will calculate precisely how much is needed on page **181**.

There is also such a thing as having too many resources. An enormous disk and RAM cache is useless unless your clients are actually making cacheable requests. If your cache is very large and full of unusable data, it can increase cache access times. You should adjust the size of your cache to fit the amount of data your clients request.

Tuning the disk cache

A question often asked is, "How big should my cache volume be?" You might think that simply adding disk space will increase your hit rate, and therefore your performance will improve. This is not necessarily true. There is a point where the cache hit rate does not climb significantly even though you increase the available disk space. Cached objects have a ***TTL*** (***Time To Live***) that specifies how long the object can be kept. Every requested page is checked against the TTL, and if the cached object is too old, a new copy is requested from the Internet and the TTL is reset.

This means that huge cache volumes will likely be full of a lot of old data that cannot be reused. Your disk cache size should generally be between 10 and 18 GB. There are quite a few cache administrators that are running large cache disks of 140 GB or more, but only use a cache size of around 18GB. If you specify a very large cache volume, performance may actually suffer. Since Squid has to manage many concurrent requests to the cache, performance can decrease as the requests per second rise. With a very large cache, the bus connected to the disk may become a bottleneck. If you want to use a very large disk cache, you are better off creating smaller cache volumes on separate disks, and distributing the disks across multiple channels.

The **`cache_dir`** configuration directive specifies the size and type of disk cache, and controls how it is arranged in directories on the disk.

For a very fast link of 8 Mbps or more, or for a large transparent proxy at an ISP, you might want to increase the cache size to 16 GB. Adding this line to your **`squid.conf`** will create a 16 GB disk cache.

```
cache_dir aufs /var/spool/squid 16384 32 512
```

The **`aufs`** argument specifies that we want to use the new Squid on-disk storage format, formally referred to as Squid's async IO storage format. The cache will be stored in **`/var/spool/squid`**. The third argument (**`16384`**) specifies the maximum size of the cache, in megabytes. The last two arguments (**`32`** and **`512`**) specify the number of first and second level directories to be created under the parent directory of the cache. Together, they specify a total of x * y directories used by Squid for content storage. In the above example, the total number of directories created will be 16384 (32 top level directories, each containing 512 subdirectories, each containing many cached objects). Different filesystems have different characteristics when dealing with directories that contain large numbers of files. By spreading the cache files across more directories, you can sometimes achieve a performance boost, since the number of files in a given directory will be lower. The default value of 16 and 256 is nearly always sufficient, but they can be adjusted if desired.

If you are using Linux as your operating system, the best file systems to use are ext2fs/ext3fs or reiserfs. Cache filesystems using ext2fs/ext3fs or reiserfs should be mounted with the **`noatime`** option (specified in **`/etc/fstab`**). If you are using reiserfs, you should add the **`notail`** mount option as well. The **`noatime`** option tells the operating system not to preserve the access times of the file, and so saves overhead. The **`notail`** tells the file system to disable packing of files into the file system tree, which is used for saving space. Both these settings have a significant impact on disk performance.

For example, to mount the reiserfs filesystem on **`/dev/sdb1`** on **`/cache1`**, add this line to your **`/etc/fstab`**:

```
/dev/sdb1       /cache1 reiserfs notail,noatime 1 2
```

Remember that if you replace a cache volume, you will need to start Squid with the **`-z`** option once in order for the cache directories to be created. Also remember to make sure the cache directory is owned by the Squid user, otherwise Squid will not be able to write to the disk.

The size of your disk cache has a direct impact on the amount of memory needed on your server. In the next section, we will see how to calculate the required amount of system RAM for a given disk cache size.

Memory utilisation

Memory is very important to Squid. Having too little RAM in your machine will certainly reduce the effective size of your cache, and at worst can bring a system to its knees. If your Squid server is so low on RAM that it starts using swap space, the Squid process will quickly consume all available resources. The system will "thrash" as it attempts to allocate memory for Squid that causes the rest of the system, and eventually the Squid process itself, to be swapped out to disk.

While more RAM is nearly always better, a good rule of thumb is to allocate 10 MB of RAM per GB of cache specified by your **`cache_dir`** directive, plus the amount specified by the **`cache_mem`** directive (page **182**), plus another 20 MB for additional overhead. Since this memory is dedicated to Squid, you should also add enough additional memory to run your operating system.

For example, a 16 GB disk cache would require 160MB of memory. Assuming that **cache_mem** is set to 32 MB, this would require:

```
    160 MB + 32 MB + 20 MB = 212 MB
```

Squid itself will use approximately 212 MB of RAM. You should add enough additional RAM to accommodate your operating system, so depending on your needs, 512 MB or more of RAM would be a reasonable total amount for the cache server.

The default Squid settings allocate 100 MB for the disk cache and 8 MB for the memory cache. This means that you will need roughly 30 MB of RAM, plus enough for your operating system, in order to run Squid "out of the box." If you have less than this, Squid will likely start to swap at busy times, bringing network access to a crawl. In this case, you should either adjust your settings to use less than the default, or add more RAM.

Tuning the hot memory cache

The **cache_mem** directive specifies the maximum amount of RAM to be used for ***in-transit***, ***hot***, and ***negative-cached*** objects. In-transit objects represent incoming data, and take the highest priority. Hot objects are those that receive many on-disk cache hits. These are moved to RAM to speed up future responses when they are requested. Negative cached objects are objects that returned an error on retrieval, such as a refused connection or a **404 Not Found**. Note that the **cache_mem** directive does NOT specify the maximum size of the Squid server process, but only applies as a guideline to these three parameters, and may occasionally be exceeded. If you have a lot of memory available on your box you should increase this value, since it is much faster for Squid to get a cached file from memory rather than from the disk.

```
cache_mem 16 MB
```

The amount of memory is specified in 4 kilobyte blocks, and should not be too large. The default is 8 MB. Squid performance degrades significantly if the process begins to use swap space, so be sure there is always sufficient free memory when the cache server is being used at peak times. Use the equation above to calculate precisely how much RAM is required for your settings.

Cacheable content limits

The configuration directive **maximum_object_size** specifies the largest object that will be cached. Setting it low will probably increase the responsiveness of your cache, while setting it high will increase your byte hit ratio. Experience has shown that most requests are under 10 KB (you can confirm this for yourself with calamaris, page **81**). If you set this value too high (say, 1 GB or more to cache movies and other large content) then your hit rate will probably decline as your disk gets filled up with large files. The default is 4 MB.

```
maximum_object_size 64 MB
```

You can also configure the maximum size of objects in memory using the **`maximum_object_size_in_memory`** directive. Setting this to a low value is generally a good idea, as it prevents large objects from holding precious system memory. Typically around 90% of your requests will fit under 20K.

```
maximum_object_size_in_memory 20 KB
```

You can determine the optimum value for this setting by looking at your Squid logs once it is up and running for some time. An analysis of the Squid log files at my university shows that 88% of the requests are under 10K. The default maximum memory object size is 8 KB.

Access Control List (ACL) optimisation

Squid has two types of ACL components: ***ACL elements*** and ***ACL rules***. Elements allow you to match particular attributes of a given request (such as source IP, MAC address, user name, or browser type). You then use rules to determine whether access is granted or denied to requests that match a particular element.

ACL **elements** are processed with ***OR logic***. This means that Squid will stop processing the ACL as soon as a match occurs. To optimise the processing of your ACLs, you should arrange the list so that the most likely matches appear first. For example, if you define an element called **`badsites`** that contains a list of sites that are to be blocked, you should place the most likely site to be visited first. If you need to block *www.ccn.com* and *www.mynewsite.com* the ACL should be:

```
acl badsites dstdomain www.cnn.com www.mynewsite.com
```

If you find yourself making a large list of users (or any other information, such as IP or MAC addresses), you should place the items that have more chance of being matched at the front of the list.

When matching ACL **rules**, Squid uses ***AND logic***. Therefore, you should list the least-likely-to-match rules first. For example, if you have a ACL called **`localnet`** that matches any host on your local network, and an ACL called **`badsites`** that lists forbidden sites, you would define a rule this way:

```
http_access deny badsites localnet
```

This way, as soon as Squid determines that **`badsites`** does not match, it will immediately skip to the next **`http_access`** rule.

Some ACLs also require more processing time than others. For example, using a ***regex*** (***regular expression***) takes longer than matching against a literal list, since the pattern must be examined more closely. ACLs such as **`src_domain`** require a host name lookup, and must wait for a DNS response from the network. In such cases, using a caching DNS server (page **143**) can help to improve your Squid response times. If you can, avoid regex matches and other "expensive" ACLs whenever possible. Large access control lists are not generally a problem as Squid has an efficient way of storing them in memory. Since Squid can perform a literal search much faster than performing pattern matching, large lists can actually yield better performance than tiny pattern matches.

Examples of some commonly used elements and rules are available in **Appendix B** on page **269**.

Redirectors

Redirectors are typically used to direct Squid to serve content other than what was requested. Rather than simply denying access to a particular site or object with an abrupt error message, a redirector can send the user to a more informative web page or take some other action. Redirectors are often used to block sites based on content (e.g. games, pornography, or advertisements) or they can be used to redirect users to a page informing them of network status or that their access is disabled.

Redirectors are run as an external process, which can significantly increase processing time and resource consumption on your cache server. You should therefore make good use of ACLs to delay redirector processing until it is actually needed. For example, if you are blocking advertisements, you typically need to check every request for a match. If the redirector is slow, this could affect the overall performance of your cache. If you move advertisement blocking to a later stage (say, after all local and approved web sites are permitted without filtering), then you can reduce the server load significantly.

There are only a finite number of child processes available to answer all redirector requests. If your redirector takes too long to complete, you could run out of redirector processes. This happens when requests come in faster than they can be processed. This will be reflected in your **`cache.log`** as "FATAL: Too many queued redirector requests," and will cause your Squid process to prematurely exit. Make sure you allocate enough children to process your requests by using the **`redirect_children`** directive.

```
redirect_children 10
```

The default value is 5 children. If you specify too many, you will consume more system RAM and CPU. This will be reflected in your system monitoring logs (page **80**) or using a spot check tool such as **`top`** (page **74**).

You can also tell Squid to bypass the redirector should it run out of available children. This should obviously not be used in access control rules, but could be appropriate when using redirectors to block advertisements on a very busy network. This will allow you to use the redirector most of the time, but bypass it when the network load grows too large. You can enable this with the **`redirector_bypass`** directive.

```
redirector_bypass on
```

A good example of a real world redirector is ***adzapper***. Adzapper is a redirector that will intercept advertising and replace it with smaller static graphic files. This obviously saves bandwidth, and can help creating a more pleasing, advertisement-free environment. Adzapper matches each request using a pattern matching approach. If the request is matched, it will replace the request with a smaller static graphic which is often much smaller than the original, and is much faster to serve since it comes from the local machine.

A colleague at another university runs adzapper on his Squid servers in an interesting way. He has journalism students who want to see the adverts. In order to cater to their needs, he runs Squid on two different ports using the **`http_port`** directive. He then has a redirector ACL that will only match against requests on one of the ports using a **`myport`** ACL element. This way most users see no advertising, but if you connect to Squid on the alternate port you will get the full, unmodified page.

```
http_port 8080 8082
acl advertport myport 8082
redirector access deny advertport
redirector_access allow ALL
```

Adzapper is available from *http://adzapper.sourceforge.net/*.

Squidguard is another popular redirector program that is often used in conjunction with "blacklists" to block categories of sites, for example pornography. It is a very powerful and flexible script that offers a fine-grained level of access control. You can download it at *http://www.squidguard.org/*.

DansGuardian

An alternative to using a redirector to filter content is to use the very popular ***DansGuardian*** program. This program runs on your Squid (or other proxy) server, and intercepts the requests and responses between the browser and proxy. This allows you to apply very sophisticated filtering techniques to web requests and responses. DansGuardian is very often used for schools to block undesirable content, and has the side affect of saving bandwidth.

DansGuardian can be downloaded from *http://dansguardian.org/*.

Authentication helpers

Authentication introduces accountability to any service, and is a particularly important part of bandwidth control. By authenticating your Squid users, a unique identity is assigned to all requests passing through the proxy server. Should any particular user consume excessive bandwidth or otherwise violate the network access policy, access can be immediately revoked and the nature of the problem will be reflected in the system logs. Examples of how to implement authentication in Squid are provided in the **Implementation** chapter, on page **101**. More details on how to setup Squid authentication are covered in the Squid FAQ at *http://www.squid-cache.org/Doc/FAQ/FAQ-23.html* .

In Squid, authentication is not done for every page, but rather it expires after a specified amount of time. After the TTL expires, the client will again be asked for authentication. Two critical parameters that affect the performance of **basic authentication** are:

```
auth_param basic children 5
```

The **`children`** parameter tells Squid how many helper processes to use. The default value is 5, which is a good starting point if you don't know how many Squid needs to handle the load. If you specify too few, Squid warns you with messages in **`cache.log`**. Specifying more will allow you to accommodate a higher system load, but will require more resources. Another important parameter for basic authentication is **`credentialsttl`**:

```
auth_param basic credentialsttl 2 hours
```

A larger **`credentialsttl`** value will reduce the load on the external authenticator processes. A smaller value will decrease the amount of time until Squid detects changes to the authentication database. Think of this value as the maximum amount of time a user can continue to use the network before another authentication check is required. Note that this only affects positive results (i.e., successful validations). Negative results aren't cached by Squid. The default TTL value is two hours.

If you are using **digest authentication**, there are three variables to consider:

```
auth_param digest children 5
auth_param digest nonce_garbage_interval 5 minutes
auth_param digest nonce_max_duration 30 minutes
```

The **children** parameter for digest authentication is used the same way as it is for basic authentication.

The **nonce_garbage_interval** parameter tells Squid how often to clean up the nonce cache. The default value is every 5 minutes. A very busy cache with many Digest authentication clients may benefit from more frequent nonce garbage collection.

The **nonce_max_duration** parameter specifies how long each nonce value remains valid. When a client attempts to use a nonce value older than the specified time, Squid generates a **401 (Unauthorized)** response and sends along a fresh nonce value so the client can re-authenticate. The default value is 30 minutes. Note that any captured Authorisation headers can be used in a replay attack until the nonce value expires, and using a smaller value will limit the viability of this kind of attack. Setting the **nonce_max_duration** too low, however, causes Squid to generate 401 responses more often. Each 401 response essentially wastes the user's time as the client and server renegotiate their authentication credentials.

Note that if you are utilising a central authentication resource, you should ensure that this server can handle all the authentication requests that may be made of it. If the authentication source is unavailable, then users will not be able to browse the web at all.

Hierarchical caches

If you run a number of Squid cache servers you may want them to cooperate with each other. This can help to maximise the amount of cache hits, and eliminate redundant downloads.

Hierarchies can help when the volume of proxy traffic becomes too high for a single server to handle. For example, imagine a university that has a 5 Mb congested Internet connection. They originally ran two independent cache servers to provide fault tolerance and to split the load. The users were directed to the cache servers by using a DNS round-robin approach. If a user requested an object from cache A, and it did not have a cached copy of the object, it would then be fetched from the original site. The fact that this object may have existed on server B was never considered. The problem was that Internet bandwidth was wasted by fetching the object from the origin when it already existed on the local network.

A better solution would be to create a hierarchy between the two caches. That way, when a request is received by one of the caches, it would first ask the other cache if it has a local copy before making a request directly to the original server. This scenario works because the two servers are on a LAN, and access between the two servers is much faster than making requests from the Internet. A hierarchical cache configuration is shown in Figure 6.1.

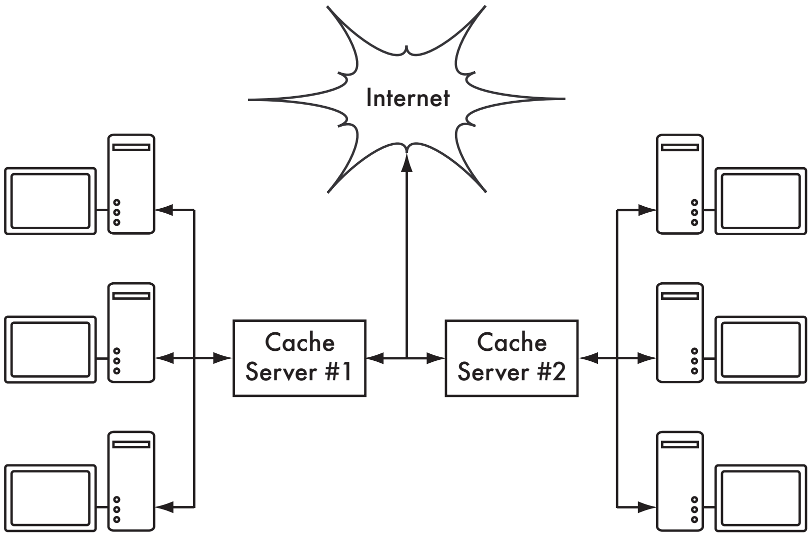

Figure 6.1: Multiple caches can distribute the load on a busy network across multiple servers, improving efficiency and response times.

One pitfall to watch out for when implementing cache hierarchies is the accidental creation of ***forwarding loops***. Forwarding loops can happen when two caches consider each other as the parent cache. When cache A requests an object from cache B, the request is forwarded back to cache A, then to cache B, and so on, until the cache servers run out of resources. For this reason, it is critically important that parent caches are never defined in a reciprocal manner.

Squid caches communicate with each other using the ***Internet Cache Protocol*** (***ICP***), as defined in RFC2186 and RFC2187. To optimise inter-cache performance and cut down on ICP queries, you should use ***cache digests***. Cache digests provide a very compact summary of the available objects in a particular cache, eliminating the need for caches to request individual objects just to determine whether or not they are available from the peer. This can dramatically speed up communication between your caches. There are no runtime configuration options to tune cache digests; simply compile Squid after passing the –**`enable-cache-digests`** to the **`configure`** script, and your cache hierarchies will use digests. There is a great deal of technical detail about cache digests in the Squid FAQ at: *http://www.squid-cache.org/Doc/FAQ/FAQ-16.html*

Since a hierarchy can improve your hit rate by 5% or more, it can be beneficial to overutilised and congested Internet links. The golden rule to remember with hierarchical caches is that your neighbour must be able to provide the data faster than the origin server for them to be worthwhile.

Configuring delay pools

Squid can throttle bandwidth usage to a certain amount through the use of ***delay pools***. Delay pools utilise the ***Token Bucket Filter*** (***TBF***) algorithm to limit bandwidth to a particular rate while still allowing short full-speed bursts.

Conceptually, delay pools are "pools" of bandwidth that drain out as people browse the web, and fill up at the rate specified. This can be thought of as a pot of coffee that is continually being filled. When there is bandwidth available in the "pot," requests are served at the best possible speed. Once the pot has emptied, it will only "refill" at a constrained rate as the coffee machine brews more coffee. To the user, this creates a small amount of fast bursting (for quickly loading a page or two). As requests continue, this is followed by a period of slower access, ensuring fairness and discouraging excessive use. As the user's requests are reduced, the coffee pot again has a chance to fill at the specified rate. This can be useful when bandwidth charges are in place, to reduce overall bandwidth usage for web traffic while continuing to provide quick responses to reasonable requests.

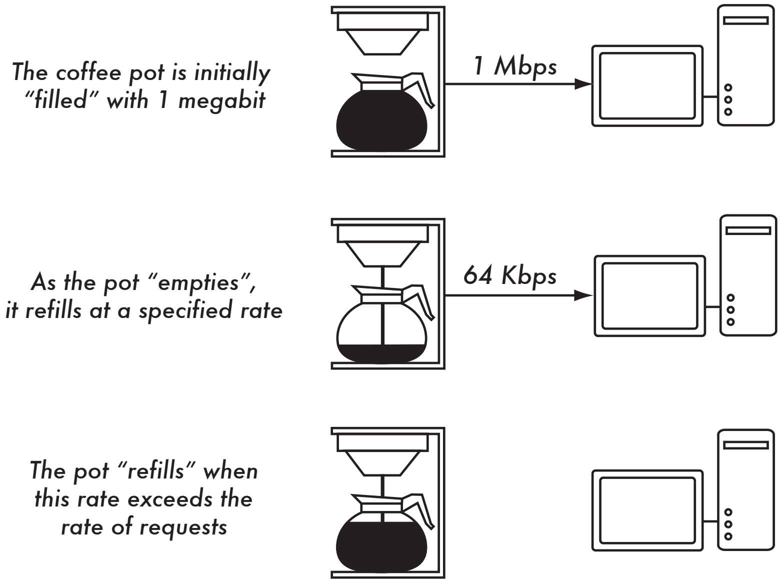
The coffee pot is initially "filled" with 1 megabit
1 Mbps
As the pot "empties", it refills at a specified rate
64 Kbps
The pot "refills" when this rate exceeds the rate of requests

Figure 6.2: Bandwidth is available as long as there is "coffee" in the "pot." When the delay pool is empty, requests are filled at the specified rate.

Delay pools can do wonders when combined with ACLs. This allows us to limit the bandwidth of certain requests based on any criteria. Delay behaviour is selected by ACLs (page **269**). For example, traffic can be prioritised based on destination, staff vs. student requests, authenticated vs. unauthenticated requests, and so on. Delay pools can be implemented at ISPs to improve the quality of service on a particular network. To enable delay pool support, Squid needs to be configured with the **`--enable-delay-pools`** option.

There are five classes of delay pools as of Squid 3.0:

1. Use one aggregate bucket for all traffic. This allows you to absolutely limit all traffic to a particular rate across the entire network.
2. One aggregate bucket for all traffic, and 256 additional buckets for all hosts in the class C network. In addition to setting an absolute rate, you can set individual rates for each IP address within the same class C network as the cache server.
3. One aggregate bucket for all traffic, 256 network buckets, and 256 individual buckets for each network (for a total of 65,536 individual buckets). With the class 3 delay pool, you have all of the limiting functionality of classes 1 and 2, and you can further limit traffic based on the source network.
4. Everything in the class 3 delay pool, with an additional limit for each user. This allows you to further limit the data rate based on the unique authenticated username.
5. Requests are grouped according to their tag (when using external authentication with the **`external_acl_type`** feature).

For example, this limits everyone to a global rate of 64 kbps, using a class 1 delay pool:

```
# Replace the network below with your own
acl delay_pool_1_acl src 192.168.1.0/255.255.255.0
# Define 1 delay pool, class 1
delay_pools 1
delay_class 1 1
# Manage traffic from our network with the delay pool
delay_access 1 allow delay_pool_1_acl
delay_access 1 deny all
# Lock users to 64kbps
delay_parameters 1 8000/8000
```

The last line specifies the rate at which the bucket is filled, and the maximum size of the bucket. Sizes are specified in bytes, **not** bits. In this example, the fill rate is 8000 bytes (64 000 bits) per second, and the maximum size of the bucket is also 8000 bytes. This defines a hard rate of 64 kbps with no bursting.

This example uses a class 2 delay pool to limit the overall rate to 128 kbps, restricting each IP address a maximum of 64 kbps:

```
# Replace the network below with your own
acl delay_pool_1_acl src 192.168.1.0/255.255.255.0
# Define 1 delay pool, class 2
```

```
delay_pools 1
delay_class 1 2
# Manage traffic from our network with the delay pool
delay_access 1 allow delay_pool_1_acl
delay_access 1 deny all
# Lock everyone to 128kbps and each IP to a maximum of 64kbps
delay_parameters 1 16000/16000 8000/8000
```

To use a class 3 delay pool, let's assume that each department or lab uses its own class-C IP block. This will limit the entire network to a maximum of 512 kbps, each class-C network will be limited to 128 kbps, and each individual IP address will be limited to a maximum of 64 kbps.

```
# Replace the network below with your own
acl delay_pool_1_acl src 192.168.0.0/255.255.0.0
# Define 1 delay pool, class 3
delay_pools 1
delay_class 1 3
# Manage traffic from our network with the delay pool
delay_access 1 allow delay_pool_1_acl
delay_access 1 deny all
# Lock everyone to 512kbps, each class-c to 128kbps,
# and each IP to 64kbps
delay_parameters 1 64000/64000 16000/16000 8000/8000
```

Finally, this example uses a class 4 delay pool to limit individual authenticated users to 64 kbps, no matter how many lab machines they are logged into:

```
# Replace the network below with your own
acl delay_pool_1_acl src 192.168.0.0/255.255.0.0
# Define 1 delay pool, class 4
delay_pools 1
delay_class 1 4
# Manage traffic from our network with the delay pool
delay_access 1 allow delay_pool_1_acl
delay_access 1 deny all
# Lock everyone to 512kbps, each class-c to 128kbps,
# each IP to 64kbps, and each user to 64kbps
delay_parameters 4 64000/64000 16000/16000 8000/8000 8000/8000
```

When using multiple delay pools, remember that the user will be placed in the first delay pool that matches them. If you define a fast ACL for sites that all users must be able to access quickly, and a slow ACL for everything else, you should place the **`delay_access`** match for the fast pool before that of the slow pool. Also note the **`deny all`** at the and of each delay pool match statement. This causes Squid to stop searching for a match for that particular pool.

More information

Squid is like other complex software, in that you should understand what you are doing before implementing it on a live network. It can be very handy to have a test Squid server that you can use to test potential changes before rolling the changes out on your production network. Change control is also very important when tuning Squid, as it is very easy to make a simple change that completely breaks the cache.

Squid is a tremendously powerful piece of software, and it is not possible do justice to all the different ways that Squid can be configured in this text. For more in-depth knowledge of Squid, we highly recommend *Squid: the Definitive Guide* by Duane Wessels, published by O'Reilly Media. The Squid FAQ (*http://wiki.squid-cache.org/*) and Google in general have plenty of Squid examples that you can use to make Squid perfectly fit your environment.

Monitoring your Squid performance

Squid provides two interfaces for monitoring its operation: SNMP and the cache manager.

The ***cachemgr*** interface is a command line interface that is accessed via the **`squidclient`** program (which ships with Squid). There is also a web-based interface called ***cachemgr.cgi*** that displays the cachemgr interface via a web page.

The SNMP interface is nice because it is easy to integrate into your existing system management package, such as Cacti (page **84**) or MRTG (page **83**). Squid's SNMP implementation is disabled by default, and must be enabled at compile time using the **`--enable-snmp`** option. Once you have an SNMP-enabled Squid, you also need to enable SNMP in your **`squid.conf`** using an ACL:

```
acl snmppublic snmp_community public
snmp_access allow snmppublic localhost
snmp_access deny all
```

Change the "public" community string to something secret. The only drawback to using SNMP is that it cannot be used to monitor all of the metrics that are available to the cachemgr interface.

All SNMP community strings in our examples start with the prefix **`enterprises.nlanr.squid.cachePerf`**. This prefix has been omitted in these examples for clarity. For example, the full SNMP community string in the next example is:

```
enterprises.nlanr.squid.cachePerf.cacheSysPerf.cacheSysPageFaults
```

For a full list of all available SNMP community strings, see the comprehensive list at: *http://www.squid-cache.org/SNMP/snmpwalk.html*

Some of the metrics you should monitor include:

- **Page Fault Rate**. Page faults occur when Squid needs to access data that has been swapped to disk. This can cause severe performance penalties for Squid. A high page rate (> 10 per second) may cause Squid to slow considerably.

  To monitor the page fault rate with the cachemgr interface:

```
# squidclient mgr:info | grep 'Page faults'
   Page faults with physical i/o: 3897
```

  The SNMP community string for tracking page faults is:

```
.cacheSysPerf.cacheSysPageFaults
```

- **HTTP request rate**. This is simply a measure of the total number of HTTP requests made by clients. If you monitor this number over time, you will have a good idea of the average load on your Squid cache throughout the day.

```
# squidclient mgr:info | grep 'Number of HTTP requests'
Number of HTTP requests received:       126722
# squidclient mgr:info | grep 'Average HTTP requests'
Average HTTP requests per minute since start:   340.4
```

  The SNMP community string is:

```
.cacheProtoStats.cacheProtoAggregateStats.cacheProtoClientHttpRequests
```

- **HTTP and DNS Service time**. These two metrics indicate the amount of time HTTP and DNS requests take to execute. They are valuable to monitor as they can indicate network problems beyond the cache server. A reasonable HTTP service time should be between 100 and 500 ms. If your service requests rise above this, you should take a look at your network connection as it may indicate a congested network. High DNS service times may indicate a potential problem with your caching DNS server (page **143**) or upstream network problems. These times may also rise if the Squid server itself is overloaded.

```
# squidclient mgr:5min | grep client_http.all_median_svc_time
client_http.all_median_svc_time = 0.100203 seconds
# squidclient mgr:5min | grep dns.median_svc_time
dns.median_svc_time = 0.036745 seconds
```

  Squid can also warn you if the HTTP service time goes over a specified threshold. It will log the warning to disk, which can notify you immediately if you are using Nagios (page **88**) or another log watching package (page **80**).

To enable high response time warnings, add this to your **squid.conf**. Times are specified in milliseconds.

```
high_response_time_warning 700
```

For SNMP, you can request the average response times in the last 1, 5, or 60 minutes. The relevant community strings are:

```
.cacheProtoStats.cacheMedianSvcTable.cacheMedianSvcEntry.cacheHttpAllSvcTime.1
.cacheProtoStats.cacheMedianSvcTable.cacheMedianSvcEntry.cacheHttpAllSvcTime.5
.cacheProtoStats.cacheMedianSvcTable.cacheMedianSvcEntry.cacheHttpAllSvcTime.60

.cacheProtoStats.cacheMedianSvcTable.cacheMedianSvcEntry.cacheDnsSvcTime.1
.cacheProtoStats.cacheMedianSvcTable.cacheMedianSvcEntry.cacheDnsSvcTime.5
.cacheProtoStats.cacheMedianSvcTable.cacheMedianSvcEntry.cacheDnsSvcTime.60
```

- **Open File Descriptors**. A Squid server that runs out of file descriptors will perform like a sick puppy. If you start running out of file descriptors, you will need to increase the number of file handles available to Squid.

```
# squidclient mgr:info | grep -i 'file desc'
File descriptor usage for squid:
        Maximum number of file descriptors:   8192
        Largest file desc currently in use:    815
        Number of file desc currently in use:  268
        Available number of file descriptors: 7924
        Reserved number of file descriptors:   100
```

On Linux, you may need to increase the overall number of file descriptors available to the system, as well as the number allowed per process. To increase the total number of available file descriptors on a Linux system, write the new value to **/proc/sys/fs/file-max**. For example:

```
echo 32768 > /proc/sys/fs/file-max
```

You should add that command to your **/etc/rc.d/rc.local** (or the equivalent in **/etc/sysctl.conf**) to preserve the change across boots. To increase the number of descriptors available to an individual process, use **ulimit**:

```
# ulimit -n 16384
```

This sets the number of file descriptors for the current process. You can call this in the initialisation script that runs Squid, just prior to launching the daemon.

The SNMP community string for available file descriptors is:

```
 .cacheSysPerf.cacheCurrentUnusedFDescrCnt
```

- **Hit ratio**. The hit ratio gives you an idea of the effectiveness of your cache. A higher the ratio means that more requests are served from the cache rather than the network.

```
#squidclient mgr:info | grep 'Request Hit Ratios'
        Request Hit Ratios:     5min: 28.4%, 60min: 27.4%
```

  As with HTTP and DNS service response time, you can request the 1 minute, 5 minute, or 60 minute average using the proper SNMP community string:

```
.cacheProtoStats.cacheMedianSvcTable.cacheMedianSvcEntry.cacheRequestHitRatio.1
.cacheProtoStats.cacheMedianSvcTable.cacheMedianSvcEntry.cacheRequestHitRatio.5
.cacheProtoStats.cacheMedianSvcTable.cacheMedianSvcEntry.cacheRequestHitRatio.60
```

- **CPU Utilisation**. While you are likely already monitoring overall CPU utilisation, it can be useful to know what percentage is being used by Squid. Periods of constant 100% utilisation may indicate a problem that needs investigation. The CPU can also become a hardware bottleneck, and constant high utilisation may mean that you need to optimise your Squid, upgrade your server, or lighten the load by bringing up another cache box (page **187**).

```
# squidclient mgr:5min | grep cpu_usage
cpu_usage = 1.711396%
```

  The SNMP community string for CPU usage is:

```
enterprises.nlanr.squid.cachePerf.cacheSysPerf.cacheCpuUsage
```

Graphing Squid metrics

Since the values can be extracted via SNMP or the cachmgr interface, they can also be easily graphed. This can show you trends that will help you predict how your cache will perform in the future.

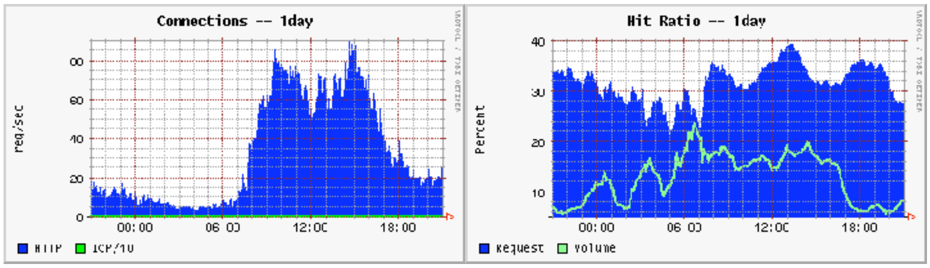

Figure 6.3: The graph on the left shows the number of connections per second to your Squid cache. The graph on the right shows the cache hit ratio as expressed by the number of requests (the dark area) and volume (the light line).

Examples of how to set up RRDtool to graph your Squid cache can be found at *http://www.squid-cache.org/~wessels/squid-rrd/*.

Traffic shaping

Traffic shaping is used to strictly enforce a particular data rate based on some predefined criteria. Let's say you are controlling your HTTP usage via a Squid proxy with delay pools, your mail has spam controls, and you enforce usage quotas for your users through proxy authentication and accounting. One day while running Ntop (page **76**), you notice that there is surprisingly large amount of traffic on port 22 (SSH). It appears that some users have figured out how to tunnel their P2P requests via the SSH port. You can't disable SSH access throughout your entire organisation, and while you could simply disable access for these users, you would need to repeat the process each time users figure out a new way to circumvent the firewall.

This is where traffic shaping can be very useful. By using traffic shaping techniques, you can limit all traffic using port 22 to a low rate, such as 16 Kbps. This will permit legitimate SSH shell traffic, while limiting the impact that tunneling and large file transfers have on the network.

Traffic shaping is best using in conjunction with traffic monitoring (page **83**) so that you have a clear idea of how much bandwidth is used by various services and machines. Before you can decide how to shape your traffic, it is paramount that you first understand how bandwidth is consumed on your network.

It is important to remember that traffic can only be shaped on transmit, not on receive. By the time inbound packets have been received, they have already traversed the network, and delaying their delivery further is usually pointless.

Linux traffic control and QoS tools

Linux has very good tools for managing bandwidth use, which are unfortunately not well known or widely used. This is probably because the tools to manipulate them are complex and poorly documented. You may well find BWM tools easier to use, but we will describe the kernel tools because they are more powerful.

To perform traffic shaping in Linux, you will need the ***iproute2*** package. It is available from *http://linux-net.osdl.org/index.php/Iproute2* and is a standard package in most distributions. This gives you the **`tc`** command, which is used for all traffic control and ***Quality of Service*** (***QoS***) configuration under Linux. Another useful component is the **`ipt_CLASSIFY`** kernel module, which comes with recent Linux 2.6 kernels. This integrates **`iptables`** with traffic control, allowing you to write netfilter rules that group traffic into certain classes.

Every network interface in Linux has a ***queuing discipline*** (***qdisc***) associated with it. The queuing discipline controls when and how the interface is allowed to send packets. You can define a queuing discipline on the external network interface of your router in order to control traffic that you send to the Internet. You can also define a queuing discipline on the internal interface to control traffic that your users receive.

None of these queuing disciplines will help you unless the queue is "owned" by the firewall itself. For example, if your firewall is connected to an ADSL modem by Ethernet, then the Ethernet link to the modem is much faster than the ADSL line itself. Therefore, your firewall can send traffic to the modem at a rate faster than the modem can send out. The modem will happily queue traffic for you, but this means that the queue is on the modem and not on the firewall, and therefore it cannot easily be shaped.

Similarly, your internal LAN is usually much faster than your Internet connection, and so packets destined for your network will build up on the Internet provider's router, rather than on the firewall. The way to prevent this is to ensure that the firewall sends less traffic to the ADSL line than the line's maximum upload speed, and less traffic to your LAN than the line's maximum download speed.

There are two types of queuing discipline: ***classful*** and ***classless***. Classful qdiscs can have other qdiscs attached to them. They modify the behaviour of all attached qdiscs. Classless qdiscs cannot have any other qdiscs attached, and are much simpler.

Various queuing disciplines are available. The simplest is ***pfifo_fast***, which separates traffic into three priority classes based on the ***Type Of Service*** (***TOS***) field in the packet header. High priority packets are always dequeued (sent) first, followed by medium and low. The pfifo_fast qdisc is classless, and the only thing that can be configured is the mapping between TOS field values and priority levels. It does not allow you to throttle your traffic in order to take ownership of the queue. See page **199** for a more detailed discussion of the TOS field.

Another useful queuing discipline is the ***Token Bucket Filter*** (***TBF***). This qdisc is also classless, but it provides a simple way to throttle traffic on a given interface. For example, the following command throttles outbound traffic on the first Ethernet interface (eth0) to 220 kbps, and limits the maximum latency to 50 ms:

```
# tc qdisc add dev eth0 root tbf rate 220kbit latency 50ms burst 1540
```

If you want more control over your bandwidth, you will need to use a classful qdisc such as ***Hierarchical Token Buckets*** (***HTB***). This allows you to place arbitrary traffic in classes and restrict the amount of bandwidth available to each class. You can also allow classes to borrow unused bandwidth from other classes. A very simple example is given below. It will throttle outgoing traffic to 50 kbps.

```
# tc qdisc add dev eth0 handle 1 root htb default 10
# tc class add dev eth0 classid 1:1 htb rate 100kbit ceil 100kbit
# tc class add dev eth0 classid 1:10 parent 1:1 htb rate 50kbit ceil 50kbit
```

To remove the current qdisc setup from an interface and return to the default, use the following command:

```
# tc qdisc del dev eth0 root
```

The following example uses the netfilter CLASSIFY module to place HTTP and SSH traffic into their own classes (1:20 and 1:30 respectively), and places individual limits on them. If you have tried another example, delete the root qdisc before running this one.

```
# tc qdisc add dev eth0 handle 1 root htb default 10
# tc class add dev eth0 classid 1:1 htb rate 100kbit ceil 100kbit
# tc class add dev eth0 classid 1:10 parent 1:1 htb rate 50kbit ceil 50kbit
# tc class add dev eth0 classid 1:20 parent 1:1 htb rate 30kbit ceil 30kbit
# tc class add dev eth0 classid 1:30 parent 1:1 htb rate 20kbit ceil 100kbit
# iptables -t mangle -F
# iptables -t mangle -A OUTPUT -p tcp --sport 80 -j CLASSIFY --set-class 1:20
# iptables -t mangle -A OUTPUT -p tcp --sport 22 -j CLASSIFY --set-class 1:30
```

The above example creates a root class (1:1) which is limited to 100 kbps, and 3 classes underneath (1:10, 1:20 and 1:30), which are guaranteed 50 kbps, 30 kbps and 20 kbps respectively. The first two are also limited to their guaranteed rates, whereas the third is allowed to borrow unused bandwidth from the other two classes up to a maximum of 100 kbps.

Two common pitfalls in traffic control are misusing branch nodes and rates. If a node has any children, then it is a ***branch node***. Nodes without children are ***leaf nodes***. You may not enqueue traffic for a branch node. That means that a branch node must not be specified as the default class, nor used in an iptables CLASSIFY rule.

Because the rate on each node is a guarantee, the rate of a branch node must be exactly equal to the total rates of all its children (leaf nodes). In the above example, we could not have specified a rate other than 100 kbps for class 1:1. Finally, it does not make sense to specify a ceil (maximum bandwidth) for any class that is greater than the ceil of its parent class.

Using SFQ to promote fairness over a 56k modem

If you have a device which has an identical link speed and actual available rate, such as a normal 56k analogue modem, you may want to use ***Stochastic Fairness Queuing*** (***SFQ***) to promote fairness. SFQ is a fair queueing algorithm designed to require fewer calculations than other algorithms while being almost perfectly fair. SFQ prevents a single TCP session or UDP stream from flooding your link at the expense of others. Rather than allocating a queue for each session, it uses an algorithm that divides traffic over a limited number of queues using a hashing algorithm. This assignment is nearly random, hence the name "stochastic."

This uses tc to enable SFQ on the device **`ppp0`**:

```
# tc qdisc add dev ppp0 root sfq perturb 10
```

That's all there is to it. Traffic that leaves ppp0 is now subject to the SFQ algorithm, and individual streams (such as downloads) should not overpower other streams (such as interactive SSH sessions).

Implementing basic Quality of Service (QoS)

If your physical link is saturated and you wish to implement basic quality of service to prioritise one type of traffic over another, you can use the ***PRIO*** queueing discipline. A packet's ***Type Of Service*** (***TOS***) bits determine whether a packet should be prioritised to minimise delay (***md***), maximise throughput (***mt***), maximise reliability (***mr***), minimise monetary cost (***mmc***), or some combination of these. Applications request the appropriate TOS bits when transmitting packets. For example, interactive applications like **`telnet`** and **`ssh`** may set the "minimise delay" bits, while file transfer applications like **`ftp`** may wish to "maximise throughput."

When a packet arrives at the router, it is queued into one of three ***bands***, depending on the TOS bits. The first band is tried first, and higher bands are only used if the lower classes have no packets queued to be sent out. According to the Linux Advanced Routing & Traffic Control HOWTO, the different combinations of TOS bits result in the following assignment of bands. All of the possible combinations of TOS bits are enumerated in the table on the next page.

Service requested	Linux priority	Band
Normal	Best Effort	1
Minimise Monetary Cost	Filler	2
Maximise Reliability	Best Effort	1
MMC + MR	Best Effort	1
Maximise Throughput	Bulk	2
MMC + MT	Bulk	2
MR + MT	Bulk	2
MMC + MR + MT	Bulk	2
Minimise Delay	Interactive	0
MMC + MD	Interactive	0
MR + MD	Interactive	0
MMC + MR + MD	Interactive	0
MT + MD	Interactive Bulk	1
MMC + MT + MD	Interactive Bulk	1
MR + MT + MD	Interactive Bulk	1
MMC + MR + MT + MD	Interactive Bulk	1

The PRIO qdisc doesn't actually shape traffic to match a particular rate. It simply assigns priority to different types of traffic as it leaves the router. Therefore it only makes sense to use it on a fully saturated link where granting priority to certain kinds of traffic makes sense. On an unsaturated link, PRIO will have no discernible performance impact.

To implement basic QoS with fairness, we can use a combination of PRIO and SFQ.

```
# tc qdisc add dev eth0 root handle 1: prio
# tc qdisc add dev eth0 parent 1:1 handle 10: sfq
# tc qdisc add dev eth0 parent 1:2 handle 20: sfq
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
```

This creates the following priority decision tree:

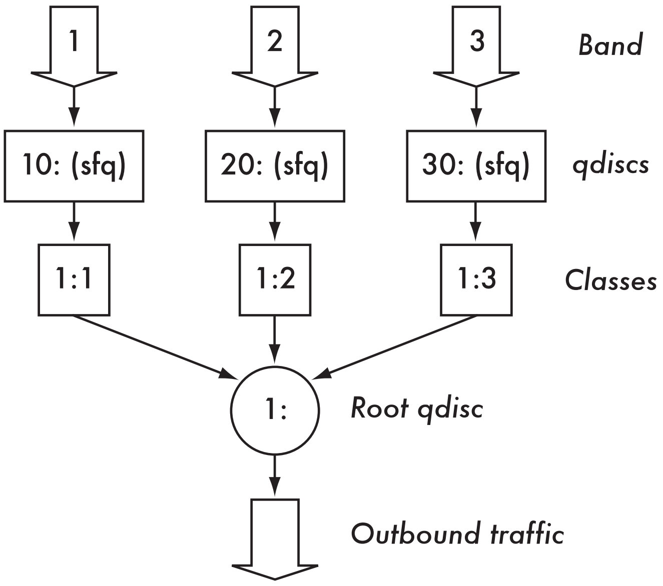

Figure 6.4: The priority decision tree used by PRIO. Note that while this example uses SFQ for each band, you could use a different qdisc (such as HTB) for every band.

Traffic for band 0 gets queued into qdisc 10:, band 1 traffic is sent to qdisc 20:, and band 2 traffic is sent to qdisc 30:. Each qdisc is released according to the SFQ algorithm, with lower numbered qdiscs taking priority.

Class Based Queueing (CBQ)

Another popular queueing discipline is ***Class Based Queueing*** (***CBQ***). CBQ is similar to the PRIO queue in that lower priority classes are polled after higher ones have been processed. While CBQ is likely the most widely known queueing algorithm, it is also one of the most complex and least accurate. But it can work well in many circumstances. Remember that it's a good idea to benchmark your network performance (page **89**) after making changes to your traffic shaping configuration to be sure that the shaper is working as you intend.

Let's assume that we have a server connected to a 2 Mbps link. We want to give web and mail a combined 1.5 Mbps of bandwidth, but not allow web traffic to exceed 1 Mbps and not allow mail to exceed 750 Kbps.

```
# Setup CBQ on the interface
tc qdisc add dev eth0 root handle 1:0 cbq bandwidth 100Mbit avpkt 1000 cell 8
# Lock us down to 1Mbit
tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 100Mbit \
  rate 1.5Mbit weight 150Kbit prio 8 allot 1514 cell 8 \
  maxburst 20 avpkt 1000 bounded
```

```
# Create a class for our web traffic
tc class add dev eth0 parent 1:1 classid 1:3 cbq bandwidth 100Mbit \
  rate 1Mbit weight 100Kbit prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000
# Create a class for our mail traffic
tc class add dev eth0 parent 1:1 classid 1:4 cbq bandwidth 100Mbit \
  rate 750Kbit weight 75Kbit prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000
# Add SFQ for fairness
tc qdisc add dev eth0 parent 1:3 handle 30: sfq
tc qdisc add dev eth0 parent 1:4 handle 40: sfq
# Classify the traffic
tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip sport 80 \
  0xffff flowid 1:3
tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip sport 25 \
  0xffff flowid 1:4
```

For more details about this complex qdisc, see the Linux Advanced Routing and Traffic Control HOWTO at *http://lartc.org/lartc.html* .

WonderShaper

WonderShaper (*http://lartc.org/wondershaper/*) is a relatively simple shell script that attempts to achieve the following:

- Maintain low latency for interactive traffic
- Allow web browsing at reasonable speeds while uploading or downloading
- Make sure uploads don't impact downloads, and vice-versa

The WonderShaper script can use the CBQ or HTB packet schedulers, and is configured by simply setting some variables at the top of the script. While it is intended for use with residential DSL networks, it provides a good example of CBQ and HTB queueing that can be used as the starting point for a more complex traffic shaping implementation.

BWM Tools

BWM Tools (*http://freshmeat.net/projects/bwmtools*) is a full firewall, shaping, monitoring, logging, and graphing package. It is implemented using userspace utilities, so any Linux kernel that supports the iptables **`-j QUEUE`** target will work. Shaping of traffic is easily accomplished by defining classes and creating flows.

The configuration file for BWM Tools is defined in XML format. A Class for SMTP traffic can be defined as follows:

```
<class name="smtp_traffic">
    <address name="inbound" dst="192.168.1.1" proto="tcp" dst-port="25">
    <address name="outbound" src="192.168.1.1" proto="tcp" src-port="25">
</class>
```

Change 192.168.1.1 to match your external IP address. To shape the traffic to allow an absolute 128 kbps inbound and 64 kbps outbound, use this:

```
<traffic>
    <flow name="smtp_inbound" max-rate="16384">
        inbound;
    </flow>
    <flow name="smtp_outbound" max-rate="8192">
        outbound;
    </flow>
</traffic>
```

Note that rates are specified in bytes per second, so you should multiply the rate by 8 to get the bits per second.

Traffic shaping with BSD

Packet Filter (***PF***) is the system for configuring packet filtering, network address translation, and packet shaping in FreeBSD and OpenBSD. PF uses the ***Alternate Queuing*** (***ALTQ***) packet scheduler to shape traffic. PF is available on FreeBSD in the basic install, but without queueing/shaping ability. To enable queueing, you must first activate ALTQ in the kernel. This is an example of how to do so on FreeBSD.

```
# cd /usr/src/sys/i386/conf
# cp GENERIC MYSHAPER
```

Now open **MYSHAPER** file with your favourite editor and add the following at the bottom of the file:

```
device pf
device pflog
device pfsync
options         ALTQ
options         ALTQ_CBQ        # Class Bases Queuing (CBQ)
options         ALTQ_RED        # Random Early Detection (RED)
options         ALTQ_RIO        # RED In/Out
options         ALTQ_HFSC       # Hierarchical Packet Scheduler (HFSC)
options         ALTQ_PRIQ       # Priority Queuing (PRIQ)
options         ALTQ_NOPCC      # Required for Multi Processors
```

Save the file and recompile the kernel by using the following commands:

```
# /usr/sbin/config MYSHAPER
# cd ../compile/MYSHAPER
# make cleandepend
# make depend
# make
# make install
```

Ensure PF is activated on boot. Edit your **/etc/rc.conf** and add this:

```
gateway_enable="YES"
pf_enable="YES"
pf_rules="/etc/pf.conf"
pf_flags=""
pflog_enable="YES"
pflog_logfile="/var/log/pflog"
pflog_flags=""
```

PF with ALTQ is now installed.

An example of how to rate limit SMTP traffic to 256 Kbps and HTTP traffic to 512 Kbps is shown below. First, create the file **/etc/pf.conf** and begin by identifying the interfaces. PF supports macros, so you don't have to keep repeating yourself.

```
#The interfaces
gateway_if  = "vr0"
lan_if      = "vr1"
```

You should replace vr0 and vr1 with your network interface card names. Next, identify the ports by using Macros:

```
mail_port = "{ 25, 465 }"
ftp_port = "{ 20, 21 }"
http_port = "80"
```

Identify the host or hosts:

```
mailsrv = "192.168.16.1"
proxysrv = "192.168.16.2"
all_hosts = "{" $mailsrv $proxysrv "}"
```

If you have several IP address blocks, a table will be more convenient. Checks on a table are also faster.

```
table <labA> persist { 192.168.16.0/24 }
table <labB> persist { 10.176.203.0/24 }
```

Now that your interfaces, ports, and hosts are defined, it's time to begin shaping traffic. The first step is to identify how much bandwidth we have at our disposal (in this case, 768 Kbps). We wish to limit mail to 256 Kbps and web traffic to 512 Kbps. ALTQ supports ***Class Based Queueing*** (***CBQ***) and ***Priority Queueing*** (***PRIQ***).

Class Based Queueing is best used when you want to define several bucket queues within primary queues. For example, suppose you have Labs A, B, and C on different subnets, and you want each individual lab to have different bandwidth allocations for mail and web traffic. Lab A should receive the lion's share of the bandwidth, then Lab B, and Lab C should receive the least.

Priority Queuing is more suitable when you want certain ports or a range of IPs to have priority, for example when you want to offer QoS. It works by assigning multiple queues to a network interface based on protocol, port, or IP address, with each queue being given a unique priority level. The queue with the highest priority number (from 7 to 1) is always processed ahead of a queue with a lower priority number. In this example. we will use CBQ.

Use your favourite editor to edit the file **`/etc/pf.conf`** and add the following at the bottom:

```
altq on $ gateway_if bandwidth 768Kb cbq queue { http, mail }
  queue http bandwidth 512Kb cbq (borrow)
  queue mail bandwidth 256Kb
```

The **`borrow`** keyword means that HTTP can "borrow" bandwidth from the mail queue if that queue is not fully utilised.

You can then shape the outbound traffic like this:

```
pass in quick on lo0 all
pass out quick on lo0 all

pass out on $gateway_if proto { tcp, udp } from $proxysrv to any \
  port http keep state queue http

pass out on $gateway_if proto { tcp, udp } from $mailsrv to any \
  port smtp keep state queue mail

block out on $gateway_if all

pass in on $gateway_if proto { tcp, udp } from any to $mailsrv \
  port smtp keep state
pass in on $gateway_if proto { tcp, udp } from any to $proxysrv \
  keep state
pass in on $lan_if proto { tcp, udp }from $mailsrv to any keep state
pass in on $lan_if proto { tcp, udp } from $proxysrv to any keep state
block in on $lan_if all
```

Farside colocation

Under many circumstances, you can save money and improve Internet speeds by moving public-facing services into a ***colocation facility***. This service can be provided by your ISP or a third party hosting service. By moving your public servers out of your organisation and into a facility closer to the backbone, you can reduce the load on your local connection while improving response times.

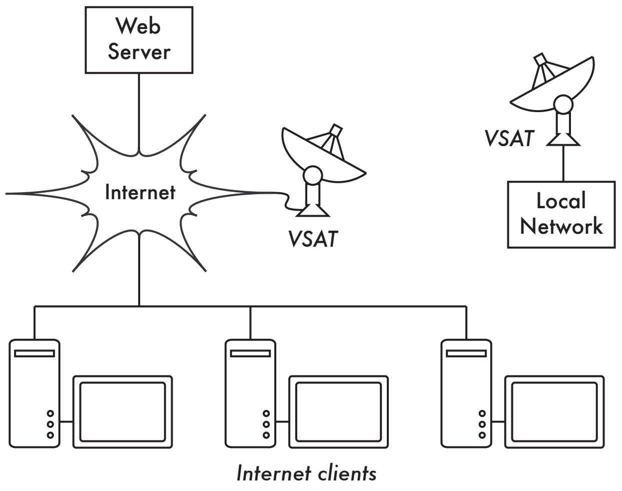

Figure 6.5: Colocation can remove load from your local Internet connection.

Colocation (often simply referred to as ***colo***) works very will with services such as:

- **Web servers**. If your public web servers require high bandwidth, colocation makes a lot of sense. Bandwidth in some countries is extremely cheap and affordable. If you have a lot of visitors to your site, choosing to host it "off-shore" or "off-site" will save you time, money, and bandwidth. Hosting services in Europe and the United States are a fraction of the cost of equivalent bandwidth in Africa.

- **Public / backup DNS servers**. DNS allows you to create redundant servers in an effort to increase service reliability. If your primary and secondary servers are hosted at the same physical location, and the network connectivity to that location goes down, your domains will effectively "fall off the Internet." Using colocation to host DNS gives you redundant name service with very little chance of both DNS servers being down at the same time. This is especially important if you're doing a lot of web hosting or if people are paying you for a hosting service, as a loss of DNS service means that all of your customer's services will be unreachable.

  Installing a public DNS server at a colo can also help improve performance, even on an unsaturated line. If your organisation uses a VSAT or other high latency connection, and your only DNS server is hosted locally, then Internet requests for your domain will take a very long time to complete.

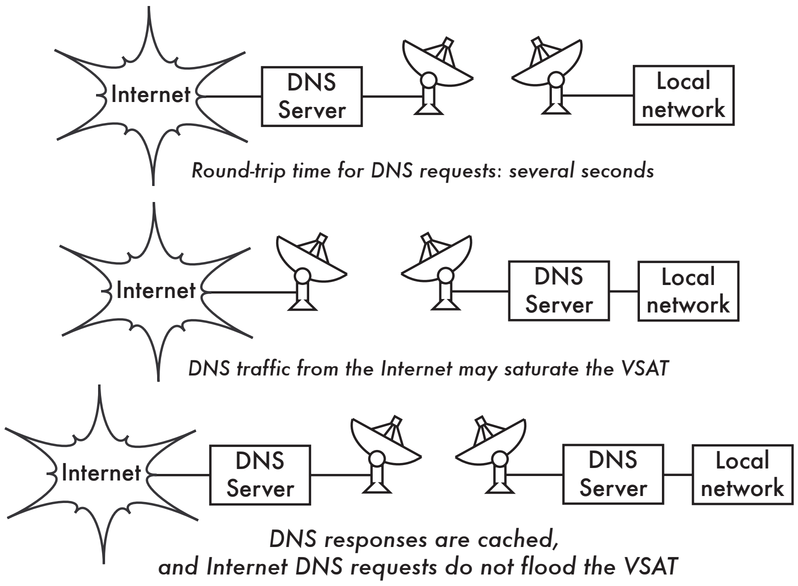
Internet
DNS Server
Local network
Round-trip time for DNS requests: several seconds
Internet
DNS Server
Local network
DNS traffic from the Internet may saturate the VSAT
Internet
DNS Server
DNS Server
Local network
DNS responses are cached,
and Internet DNS requests do not flood the VSAT

Figure 6.6: Inefficient DNS configuration can cause unnecessary delays and wasted bandwidth.

If you use private IP addressing internally (i.e., you are using NAT) then your DNS server will need to send different responses depending on where the requests come from. This is called ***split horizon DNS***, and is covered on page **212**.

- **Email**. By combining anti-virus and anti-spam measures at a colo server, you can drastically reduce bandwidth wasted on undesired content. This technique is sometimes called ***far-side scrubbing***. If you rely on client-side virus scanners and spam filters, the filtering can only occur after the entire mail has been received. This means that you have needlessly downloaded the entire message before it can be classified as spam and discarded. Often times, 80% or more of all inbound mail in a normal educational setting can be spam and virus content, so saving this bandwidth is vital.

  To implement far-side scrubbing, you should set up an anti-virus and anti-spam solution as discussed in chapter four: **Implementation**, and host that server at a colocation facility. You should also install a simple internal email server, with firewall rules in place that only allow connections from the colo. After mail has been filtered at the colo, the remote email server should forward all mail to the internal server. This server can then simply deliver the mail without further filtering.

Choosing a colo or ISP

When shopping for an ISP or colocation facility, do not overlook the details of the ***Service Level Agreement*** (***SLA***). This document describes the precise level of service that will be provided, including technical support, minimum uptime statistics, emergency contact procedures, and liability for unforeseen service outages. Remember that a promise of 99.99% uptime means that an ISP can be down for about seven hours **per month** before their SLA is violated. Keep this in mind, as it can and most probably will happen at least once as every colo provider experiences denial of service attacks, hardware failure, and simple human errors. While it's almost certain that your service will get interrupted eventually, the SLA will determine what course of action is available to you when it does.

You should also pay attention to the technical specifications of a potential data centre. Do they have backup power? Is the facility well ventilated? Do they have a multi-homed Internet connection with enough capacity to meet your needs as well as the needs of the rest of their customers? Do they use trusted equipment and professional installations, or is the data centre a haphazard tangle of wires and hardware? Take a tour of the facility and be sure you are comfortable with the organisation before making an agreement.

Billing considerations

Flat rate billing is when you're allocated a certain amount bandwidth, and are capped at this rate. For instance you may be allocated 1.5 Mbps of inbound and outbound bandwidth. You can use up to this amount for as long as you want with no fear of being billed extra at the end of the month. The only drawback to this is if you wish to have the ability to ***burst***, using more than the allocated 1.5 Mbps during busy times.

The ***95th percentile*** method allows for bursting. Bandwidth rates are polled every 5 minutes. At the end of the month, the top 95% spikes are removed and the maximum value is taken as the billed rate for the entire month. This method can lead to unexpectedly large bandwidth bills. For example, if you use 10 Mbps for 36 hours straight, and you use less than 1 Mbps for the rest of the month, you will be billed as if you used 10 Mbps for the entire month. The advantage of this method is that you can occasionally burst to serve much more traffic than normal, without being billed for the bursts. As long as your peak times fall in the top 5% of overall traffic, your bill will not increase.

You may also be billed by ***actual usage***, also known as ***by-the-bit***. ISPs may choose to bill you for every byte of traffic you transmit and receive, although this isn't commonly done for colo hosting. Actual usage billing is normally associated with a dedicated connection, such as a 100 Mbit or 1 Gbit line.

Protocol tuning

Most times, the default TCP parameters provide a good balance between performance and reliability. But sometimes it is necessary to tune TCP itself in order to achieve optimal performance. This is particularly important when using high latency, high throughput networks such as VSAT. You can drastically improve performance on such a link by eliminating unnecessary acknowledgments.

TCP window sizes

The ***TCP window size*** determines the size of a chunk of data that is sent before an ACK packet is returned from the receiving side. For instance, a window size of 3000 would mean that two packets of 1500 bytes each will be sent, after which the receiving end will ACK the chunk or request retransmission (and reduce the window size at the same time).

Large window sizes can speed up high-throughput networks, such as VSAT. For example, a 60 000 byte window size would allow the entire chunk to be sent to the receiving end before an ACK reply is required. Since satellite bandwidth has such high latency (about 1 second in Africa), using a small window size greatly reduces the available throughput. The standard window size of 1500 would require an ACK for each packet to be sent, introducing an additional 1 second of latency per packet. In this case, your available throughput would be roughly 1-2 Kbps maximum, even though the available bandwidth of the VSAT is much higher.

The TCP window size and other TCP tuning parameters can be easily adjusted in Linux and BSD.

Linux

RFC1323 defines two important high performance TCP extensions. It provides the TCP "Window Scale" option to permit window sizes of greater than 64 Kb. This will enable window scale support in Linux:

```
echo "1" > /proc/sys/net/ipv4/tcp_window_scaling
```

RFC1323 also establishes a mechanism for improving ***round trip time*** (***RTT***) calculations through the use of timestamps. A more accurate RTT means that TCP will be better able to react to changing network condition. This command enables timestamp support:

```
echo "1" > /proc/sys/net/ipv4/tcp_timestamps
```

To set the maximum size of the TCP receive and transmit windows respectively:

```
echo [size] > /proc/sys/net/core/rmem_max
echo [size] > /proc/sys/net/core/wmem_max
```

You can also adjust the default size of TCP receive and transmit windows:

```
echo [size] > /proc/sys/net/core/rmem_default
echo [size] > /proc/sys/net/core/wmem_default
```

As the available bandwidth increases, the transmit queue should also be increased. This is particularly important on very high bandwidth connections. The length of transmit queue can be set with **`ifconfig`**:

```
ifconfig eth0 txqueuelen [size]
```

FreeBSD

You can activate window scaling and timestamp options (as per RFC1323) with a single command in FreeBSD:

```
sysctl net.inet.tcp.rfc1323=1
```

To set the maximum TCP window size:

```
sysctl ipc.maxsockbuf=[size]
```

The default size of the TCP receive and transmit windows are set like this:

```
sysctl net.inet.tcp.recvspace=[size]
sysctl net.inet.tcp.sendspace=[size]
```

For more information about TCP window size and other protocol tuning, see:

- *http://proj.sunet.se/E2E/tcptune.html*
- *http://www.psc.edu/networking/projects/tcptune/*
- *http://www.hep.ucl.ac.uk/~ytl/tcpip/linux/txqueuelen/*

Link aggregation

By combining two or more network connections into a single logical connection, you can increase your throughput and add a layer of redundancy to your network. There are two mechanisms available to aggregate network links in Linux: via the bonding driver, and using routing.

Bonding

Bonding is one method for combining the throughput of two or more network connections. When using bonding, two or more physical interfaces are combined to create one virtual interface capable of the combined throughput. Bonding requires both sides of the connection to support the technology.

Let's assume we have two hosts that each have two 100 Mbit interfaces, eth0 and eth1. By bonding these two interfaces together, we can create a logical device (bond0) that provides a 200 Mbit link between the two hosts.

Run the following on both hosts.

```
# Make sure the bonding driver is loaded
modprobe bonding
# Set our IP, .10 for the first host, .11 for the second
ip addr add 192.168.100.10/24 brd + dev bond0
# Bring the interface up
ip link set dev bond0 up
# Add our slave interfaces
ifenslave bond0 eth0 eth1
```

If you use bonding, you should connect the bonded machines via cross-over cables, or use a switch that supports port trunking. Since both physical devices will use the same hardware (MAC) address, this can confuse conventional switches. For more information about bonding, see:

- *http://linux-net.osdl.org/index.php/Bonding*
- *http://linux-ip.net/html/ether-bonding.html*

Aggregate routing

You can also aggregate links by using routing alone. You can either use all links in a round-robin fashion (called ***equal cost routing***), or you can fail over to a second connection when the first becomes saturated. The first option is appropriate when the monetary cost of both links is equal. The second allows you to use a less inexpensive link for most of your traffic, and only fail over to the more expensive connection when the demand is high.

To perform equal cost routing between eth1 and eth2:

```
# ip route add default dev eth1 nexthop eth2
```

To only use eth2 when the traffic on eth1 saturates the link:

```
# ip route add default dev eth1 weight 1 nexthop eth2 weight 2
```

For more examples of how and when to use aggregate routing, see the Linux Advanced Routing & Traffic Control HOWTO at *http://lartc.org/lartc.html* .

DNS optimisation

Optimising a DNS cache will provide users with fast resolution of DNS queries, thus providing fast initial "startup" times for connections. Faster DNS response times make everything else on the network seem to "go faster."

Without memory restrictions, a DNS cache can run wild and use whatever memory is available. For example, a misconfigured BIND installation can easily eat up 4 GB of RAM when operating as a DNS cache server for a small ISP. To limit RAM consumption on BIND, add a **`max-cache-size`** option to your options section:

```
options {
  max-cache-size 16M;
}
```

DJBDNS uses 1 MB of memory for its cache by default, which may be a bit small for some installations. This will change the cache size to 16 MB:

```
echo 16000000 > /service/dnscache/env/CACHESIZE
echo 16777216 > /service/dnscache/env/DATALIMIT
svc -t /service/dnscache
```

It can be difficult to find the best location for hosting your public web server. If you host it on the local LAN, then local users will be able to access it very quickly, but connections from the public Internet will consume your bandwidth. If you host it at a colocation facility (as mentioned on page **205**), then the Internet at large will be able to access it very quickly, but local users (for example, students, and faculty) will need to use the Internet connection to access it.

One approach to this problem is to use a combination of mirroring (page **144**) and ***split horizon DNS***. You can mirror the contents of the web server and have a local copy as well as a public copy hosted at a colo. You can then configure your DNS server to return the IP address of the internal server when it is requested from the local LAN, and otherwise return the IP address at the colo.

In this example, hosts with a source IP address of 192.168.0.0/24 are given private responses to DNS queries, while all other addresses are given public responses.

```
acl internal {
    192.168.0.0/24;
}
```

```
view "internal" {
  match-clients {
    localhost;
    internal;
  };
  recursion yes;

  zone "." {
    type hint;
    file "caching/root.cache";
  };

  zone "companynet" in {
    type master;
    file "master/companynet-private";
  };

  zone "0.168.192.in-addr.arpa" {
    type master;
    file "master/0.168.192.in-addr.arpa";
  };
};


view "public" {

  recursion yes;

  zone "." {
    type hint;
    file "caching/root.cache";
  };

 zone "companynet" in {
    type master;
    file "master/companynet-public";
  };
};
```

This configuration will direct traffic to the most appropriate server.

You can use split horizon in any situation where clients should be redirected based on their source IP address. For example, you may wish to direct email clients to an internal email server when they are physically at the office, but to a different server when travelling. By assigning a different view for your mail host (one pointing at an internal IP address, the other pointing at a public IP) your users' email will continue to work without the need to change settings while travelling. A network using split horizon DNS is shown in Figure 6.7.

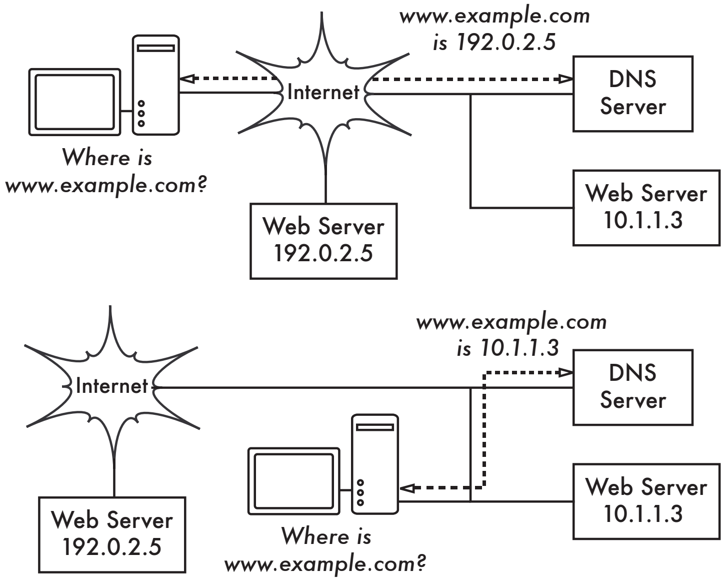

Figure 6.7: Split horizon directs users to the appropriate server depending on their source IP address.

Web access via email

While Internet continues to quickly expand, there remains a large community of users who only have access to e-mail. This can be because Internet service providers do not offer full Internet connections (due to inadequate infrastructure and low-bandwidth lines) or because the users simply cannot afford to pay for full Internet capabilities. Many of these users live in remote areas of developing countries and rely on e-mail not only for interpersonal communication, but also to access essential medical, business, and news information.

Despite the lack of broad and unlimited access to Internet in remote areas, there is still a plethora of creative and diverse content that scientists, artists, and people in general can share over, and retrieve from the net. Indeed, an important lesson learnt in recent years is that high-bandwidth access to the Internet is not essential for bridging the digital divide. To some extent, the exchange and transfer of knowledge and technology is possible using only e-mail (to retrieve general content Web pages) or Web-to-email gateways (to retrieve eJournals).

Some of these facilities, which mostly available for free, are discussed in this section. It is possible to access nearly any site on the Internet through e-mail.

There exists a moderated mailing list called ACCMAIL as a forum for communicating news, comments, and questions about e-mail only methods of accessing the Internet. To contribute to the discussion, e-mail to *accmail@listserv.aol.com* and to subscribe, send an e-mail to *listserv@listserv.aol.com* with SUBSCRIBE ACCMAIL as the message body.

www4mail

This is an open source application that allows you to browse and search the whole Web via e-mail by using any standard Web browser and any MIME (Multipurpose Internet Mail Exchange) aware e-mail program. E-mail messages sent to www4mail servers get automatically passed to the e-mail agent after selecting one or more buttons that link to other Web documents within a requested Web page. There are many options available, including user quotas, e-mail uuencoding reply, etc. All of these are described in the www4mail users' manual available at *http://www.www4mail.org/*.

By default, www4mail servers deliver web pages as attached HTML without including any images. In this way, returned e-mails are smaller, arrive faster, download quicker, and take up less space in an e-mail box. It is also possible to retrieve images via e-mail. Further information on the use of www4mail can be retrieved by sending an e-mail To: *www4mail@wm.ictp.trieste.it* and writing in the body of the message: "help" (without quotes).

To request the homepage of the International Centre for Theoretical Physics in Trieste, Italy, you would send an e-mail To: *www4mail@wm.ictp.trieste.it* and simply write in your e-mail message: *www.ictp.it* (without leading spaces). You should receive a reply in a few minutes, depending on Internet traffic.

Long URLs in the message body should be wrapped by using a backslash "\" without quotes. For example, the url:

```
http://cdsagenda5.ictp.it/full_display.php?smr=0&ida=a05228
```

...can be wrapped into:

```
http://cdsagenda5.ictp.it/full_display.php\
?smr=0&ida=a05228
```

To search in Yahoo for "peanuts" using www4mail, simply send another e-mail with the message: **`search yahoo peanuts`** .

web2mail

Just send an email to: *www@web2mail.com* with the web address (URL) of the web page you want as the Subject: of your email message. Another available web2mail e-mail addresses is: *web2mail@connectweb.de* .

PageGetter.com

Send an e-mail, including one or more URLs in the subject or body of your message, to *web@PageGetter.com* . You will automatically receive the full requested web page, complete with embedded graphics. Web pages with frames will be split into multiple e-mails, as many e-mail clients can not support frames. But if your e-mail client supports frames (i.e., Outlook Express) you can receive all frames in a single e-mail. In this case send the e-mail to *frames@PageGetter.com*. To receive a text-only version of the requested page, write instead to: *text@PageGetter.com* . This is especially useful for Personal Digital Assistants (PDAs), cell phones, and text only e-mail systems. You can also send an e-mail to: *HTML@PageGetter.com* to receive the full HTML page with no graphics.

GetWeb

GetWeb is another useful web to e-mail automatic service run by the Health-Net organisation . Send an e-mail To: *getweb@healthnet.org* with these three lines in the message body:

```
. begin
. help
. end
```

You will receive a reply with further instructions. If your mail software automatically inserts text (such as a signature) at the beginning or end of your message, an error will occur. To prevent this, GetWeb requires you to enclose retrieval commands in a "begin" and "end" block as shown above. The GetWeb server ignores any text that appears before or after this block.

To retrieve a particular website like "www.ictp.it" use the command get:

```
. begin
. get http://www.ictp.it
. end
```

For searching the words "malaria" and "Africa" with GetWeb use instead the strings:

```
. begin
. search yahoo malaria and Africa
. end
```

Time Equals Knowledge (TEK)

TEK is an open source web-to-email client that uses email as a transport mechanism for displaying web pages. It empowers low-connectivity communities by providing a full Internet experience, including web searching, caching, and indexing. Rather than viewing the web with an email client, users run TEK from within a normal web browser. The TEK distribution includes a customised version of Firefox that is pre-configured to talk to TEK rather than make requests directly from the Internet. It is also straightforward to configure other web browsers to talk to TEK.

TEK clients connect to a proxy server hosted at MIT that provides simplification and compression of pages, conversion of PS and PDF to HTML, and an interface to Google. The TEK client is released under the GNU GPL, and is free to download from *http://tek.sourceforge.net/*

Other useful web-to-email applications

Using GetWeb, the HealthNet SATELLIFE (*www.healthnet.org*) has pilot projects aimed at expanding access to health and medical information and supporting data collection and analysis through the use of handheld computers (PDAs) connected via the local GSM cellular telephone network in African regions.

The eJDS -free electronic Journal Delivery Service (*www.ejds.org*) is an application of www4mail geared to facilitate the access to current scientific literature free of cost in the fields of Physics and Mathematics. The goal is to distribute individual scientific articles via e-mail to scientists in institutions in developing countries who do not have access to sufficient bandwidth to download material from the Internet in a timely manner. Providing scientists with current literature supports their ongoing research and puts them with their peers in industrialised countries.

loband.org

A useful service offered by AidWorld that simplifies web pages in order to make them download faster over slow Internet connections is available at: *http://www.loband.org/.*

To use the service, simply type the web address of the page you want to visit into a web form and click the "Go" button. Once you start browsing through loband, the simplified page will contain the same text information as the original website and the formatting of the simplified page will be as similar to the original as possible, with colours and images removed.

Images contained in the original page are replaced by links containing an "i" in square brackets, e.g. [i-MainMap] or [i]. In either case, the original image can be viewed by clicking the link.

High Frequency (HF) networks

HF (***High-Frequency***) data communications enable radio transmission of data over a range of hundreds of miles.

HF radio waves reflect off the ionosphere to follow the curvature of the earth. The great advantage of HF is it can go the distance, leaping over obstacles in its path. Where HF wins the wireless game in range, but it loses in data capacity. Typical HF radio modems yield about 2400 bps. Two-way radio is the classic half-duplex medium of communication, so you are either transmitting or receiving, but not both at the same time. This, plus the robust error-checking protocols implemented by the modem hardware itself, means the actual link experience is on the order of 300 bps. It is not possible to use HF data communications for on-line browsing, chat, or video-conferencing. But HF is a workable solution for very remote places when using classic store-and-forward applications like text-based e-mail. One simply needs to pay close attention to the configuration and try to optimise as much as possible.

A basic HF data communications system consists of a computer, a modem, and a transceiver with an antenna. Modems used for HF radio vary in throughput and modulation technique, and are normally selected to match the capabilities of the radio equipment in use. At HF frequencies, Bell 103 modulation is used, at a rate of 300 bit/s.

Two distinct error and flow control schemes may be simultaneously used in HF networks. One is the Transport Control Protocol (TCP) which provides reliable packet delivery with flow and congestion control. The other is a radio link ***Automatic Repeat Request*** (***ARQ***) protocol which provides radio link layer frame error recovery.

The adaptive mechanisms in TCP are not optimum for HF networks, where link error rates are both higher and burstier than in the wired Internet. TCP has been designed under the assumption that packet losses are caused almost exclusively by network congestion, so TCP uses congestion avoidance mechanisms incorporating rate reduction and multiplicative increase of the retransmission timeout.

Application protocols (e.g., HTTP, FTP, and SMTP) may exchange many short commands and responses before each large file transfer (web page, file, or email message). In the course of transferring a single email message, an SMTP client will send at least four short SMTP commands and receive a similar number of short replies from the SMTP server. This high ratio of short to long messages is an important characteristic of Internet applications for the designer of a suitable wireless ARQ protocol.

The clients of a HF wireless ARQ protocol (e.g., IP) call upon the ARQ entity to deliver data reliably over a link. Just as for TCP, reliability is achieved by retransmissions. A typical sequence of events is a link setup handshake, followed by cycles of alternating data transmissions and acknowledgments from the destination. Each such cycle requires that the roles of physical transmitter and receiver be reversed twice. During each link turnaround some delay is introduced. This added cost for link turnarounds is a characteristic of wireless links, especially for HF radio links whose turnaround times range up to tens of seconds. However, the timeouts and other behaviour of a HF wireless ARQ protocol can be matched to known characteristics of a specific wireless channel for more efficient operation.

For more information about HF networks, see:

- NB6Z's Digital Ham Radio page, *http://home.teleport.com/~nb6z/about.htm*
- Introduction to Packet Radio, *http://www.choisser.com/packet/*
- Das Packet Radio Portal (German), *http://www.packetzone.de/*

Modem optimisation

If your only connection to the Internet is a 56k modem, there are a few things you can do to make the best possible use of the available bandwidth. As mentioned on page **199**, using SFQ with PRIO can help ensure that traffic is handled fairly on a congested link. Many of the other techniques in this book (such as using good caching software and doing far-side scrubbing) can help to ensure that the line is only used when it is absolutely needed. If you are using a dedicated modem connection, you will want to evaluate the performance when using various forms of compression.

Hardware compression

Standards such as V.42bis, V.44 and MNP5 are hardware compression techniques that are implemented in the modem itself. Under some circumstances, using hardware compression can significantly improve throughput. Note that the modems on both sides of the link must support the compression used.

- **V.42bis** is an adaptive data compression developed by British Telecom. The v.42bis algorithm continually estimates the compressibility of the data being sent, and whenever compression is not possible, it switches to a "transparent mode" and sends the data without compression. Files that have already been compressed (such as file archives or MP3 files) do not compress well, and would be sent without further processing. To enable V.42bis, use this modem string: **`AT%C2`**

- **V.44** is a data compression standard incorporated into the v.92 dial-up modem standard. V.44 can provide significantly better compression than V.42bis. V.44 was developed by Hughes Electronics to reduce bandwidth consumption on its DirecPC and VSAT products. Hughes released the V.44 algorithm in 1999, and it has been widely implemented as an alternative to V.42bis. To enable V.44, use this modem string: **`AT+DS44=3,0`**

- **MNP5** is short for Microcom Network Protocol 5. While it can provide better compression than V.42bis, it can only do so on compressible files. If you transfer data that is already compressed, MNP5 may actually decrease your throughput as it attempts to compress the data further. Many modems support the use of both MNP5 and V.42bis, and will select the appropriate compression algorithm depending on the data being sent. To enable MNP5 and V.42bis, the modem string is **`AT%C3`**. To enable just MNP5, use **`AT%C1`**.

Software compression

Enabling PPP compression can further increase throughput. Since PPP is implemented in a host computer (and not modem hardware), there are more aggressive compression algorithms available that can increase throughput by consuming slightly more CPU and RAM. The three most popular PPP compression algorithms are ***BSD Compression***, ***Van Jacobson***, and ***deflate***.

BSD Compression

The BSD Compression protocol (***bsdcomp***) is outlined in RFC1977. The algorithm is the same that is used in the ubiquitous UNIX compress command. It is freely and widely distributed and has the following features. It supports Automatic optimisation and disabling of compression when doing so would be ineffective. Since it has been widely used for many years, it is stable and well supported by virtually all PPP implementations.

To enable PPP in pppd 2.4.3, add the following directive to your command line (or options file): **`compress 15,15`**

VAN Jacobson (VJ)

Van Jacobson (VJ) header compression, detailed in RFC1144, is specifically designed to improve performance over serial links.

VJ compression reduces the 40 byte TCP/IP packet header to 3-4 bytes by tracking the state of TCP session on both sides of the link. Only the differences in header changes are sent in future transmissions. Doing this drastically improves performance and saves on average of 36 bytes per packet. VJ compression is included in most versions of PPP. To enable Van Jacobson compression, add this to your PPP configuration: **`vj-max-slots 16`**

Deflate

The deflate compression algorithm (RFC1979) is based on the Lempel-Ziv LZ77 compression algorithm. The GNU project and the Portable Network Graphics working group have done extensive work to support the patent-free status of the deflate algorithm. Most PPP implementations support the deflate compression algorithm.

To enable deflate compression, use this PPP option: **`deflate 15,15`**

Bandwidth accounting

Bandwidth accounting is the process of logging bandwidth usage on a per-IP or per-user basis. This can make it very easy to derive statistics about how much bandwidth individual users and machines are consuming. By using accounting techniques, you can generate hard statistics that can be used to enforce quotas or limits.

Squid bandwidth accounting

If you use authentication on your Squid cache, it can be desirable to perform bandwidth reporting or accounting broken down per user. If you intend to introduce quotas, or if you want to charge your users for access, you will have to keep records of each user's utilisation. There are many ways to do this by applying some simple scripts. This example is a quick recipe for basic bandwidth accounting. Feel free to adapt it to fit your own organisation.

To begin, you will obviously need a Squid server configured for user authentication (page **186**). You will also need a server with AMP (Apache, Mysql, PHP, and Perl) installed. If your network is relatively small, these can be installed on the same box as the Squid server. For busier environments, you should install these components on a separate machine. You will also need to install Perl (with the Perl DBI interface) on the Squid server. If you wish to implement this without the user front end, you only need Perl and a MySQL database.

First, download the example scripts from *http://bwmo.net/*. On the MySQL server, create the database tables by following the instructions provided in the **`mysql-setup.txt`** file included in the archive.

Next, copy the **`dolog.pl`** script to the MySQL server. This script will parse the log entries, and enter a summary of the user and site information into the database. Edit it to reflect the user name and password needed to connect to the MySQL database. Schedule a cron job to run the script every half hour or so.

On the Squid server, you will need to run the script **`squidlog.pl`**. You should insert the correct values for the MySQL server network address, user name, and password, as well as the correct path to the Squid log file. This script should be put in a cron job to run at some suitable interval (for example, every 15 or 30 minutes).

Finally, on the web server, make sure Apache is configured to use PHP with the LDAP extension installed. Extract and place the contents of the tar file into your web server's document directory. Edit **`config.php`** to include the correct values for the LDAP authentication source, and then visit the directory in your web browser.

You should now be able to log in as a user and view your browsing history. Even if you do not use the web application to give your users access to the database, you will still have a database that you can use for your own needs. This provides a perfect base to implement quotas, generate a report of the top 100 users and web sites visited, and so on. You can use any SQL tool you like (such as phpMyAdmin) to view the data, and can make simple scripts to provide reports and even take action when certain thresholds are crossed. When used in conjunction with a redirector (page **184**), you can automatically notify users when their bandwidth consumption reaches a specified limit.

Bandwidth accounting with BWM tools

As mentioned earlier, BWM Tools is a full-featured firewall, shaping, logging, and graphing system. It can be integrated with RRDtool to generate live graphs, including bandwidth utilisation and statistics.

Setting up of BWM Tool flows for reporting purposes is very simple. To log data every 5 minutes, add this to your configuration file:

```
<traffic>
    <flow name="smtp_inbound" max-rate="16384" report-timeout="300">
        inbound;
    </flow>
    <flow name="smtp_outbound" max-rate="8192" report-timeout="300">
        outbound;
    </flow>
</traffic>
```

If you want to log directly to an RRD file, you can set it up like this:

```
<traffic>
    <flow name="smtp_inbound" max-rate="16384" report-timeout="300"
report-format="rrd">
```

```
        inbound;
    </flow>
    <flow name="smtp_outbound" max-rate="8192" report-timeout="300"
report-format="rrd">
        outbound;
    </flow>
</traffic>
```

You can now use RRDtool (page **83**) or an integrated package (such as Cacti, page **84** or Zabbix, page **89**) to display graphs based on the data files generated above.

Linux interface bandwidth accounting with RRDtool

Using a simple Perl script with RRDtool integration, you can monitor interface bandwidth usage and generate live graphs by polling an interface directly on a Linux server.

You will need this script:

http://www.linuxrulz.org/nkukard/scripts/bandwidth-monitor

Edit the script and change **`eth0`** to match the interface you wish to monitor. You create the RRD files by running the script with the **`--create-rrd`** switch:

```
bandwidth-monitor --create-rrd
```

This will create the file **`/var/log/bandwidth.rrd`**. To start the bandwidth monitor, run this command:

```
bandwidth-monitor --monitor
```

The script will keep running and will log traffic statistics every five minutes. To view live statistic graphs, download this cgi and put it into your web server's **`cgi-bin`** directory:

http://www.linuxrulz.org/nkukard/scripts/stats.cgi

Now visit that script in your web browser to see the flow statistics for the interface being monitored.

VSAT optimisation

(Note: This portion was adapted from the VSAT Buyer's Guide, and is used with permission. This guide is also released under a Creative Commons license, and is full of valuable information that is useful when comparing satellite communications systems. For the full guide, see *http://ictinafrica.com/vsat/*).

VSAT is short for Very Small Aperture Terminal, and is often the only viable connectivity option for very remote locations. There are a whole host of technical considerations you will need to make when buying a VSAT. Most of them involve making trade-offs among the technical characteristics that give you what you want and what you can afford. The common considerations you may be forced to make are:

- Whether to use inclined orbit satellites
- Whether to use C or Ku band
- Whether to use shared or dedicated bandwidth

The technology you choose will necessarily need to balance operating costs with performance. Here are some points to keep in mind when choosing a VSAT provider.

Use of inclined orbit satellite

The price of bandwidth on inclined orbit satellites is usually much lower since these satellites are nearing their end of life. The downside is that it requires a dish with tracking capabilities that can be very expensive. The high capital costs associated with the expensive antenna can be offset by lower operating costs, but only if you are purchasing large amounts of bandwidth. You should therefore make sure that you carefully consider both your capital and operating costs over the period you intend to operate the VSAT. Of course, remember to ascertain the exact remaining life of the satellite, when making this consideration. If you decide to opt for inclined orbit capacity, caution is advised as the service can be down for a while in the event that you are running mission critical applications.

C band, Ku band, and Ka band

One of the big decisions you are likely to encounter when buying a VSAT is whether to use C band or Ku band. In order to enable you to arrive at an informed decision, we have briefly presented the advantages and disadvantages of each band.

Advantages of using C band

- **C band is less affected by rain.** If you happen to live in a high rain-fall area such as the tropics and your VSAT applications are “mission critical," in other words, you want your system available all the time, you can opt for C band over Ku band. However, this does not exclude the use of Ku band systems in the tropics especially for home TV and Internet access since these are not mission critical applications and the consumers can live with a few hours of intermittent service.

- **C band systems have been around longer than Ku band systems and thus rely on proven technology.** However, Ku band systems seem to be overtaking C band systems as the preferred technology in the home consumer business in the last few years. Note that Ku band dishes are more likely to be smaller and therefore cheaper for any given application, because of Ku band's higher frequencies. You should also bear in mind that Ku band bandwidth prices are higher than C band prices and therefore any savings on capital costs could be offset by higher operating costs.

- **C band satellite beams have large foot prints.** The global beam for C band covers almost a third of the earth's surface. If you are looking for single satellite hop operation (e.g. for real time applications such as VoIP or video conferencing) to connect locations far apart from one another, you may be forced to choose the wider coverage C band beams. However, the latest satellites launched have large Ku band beams covering entire continents. You should also note that two beams on the satellites can be connected through a method called "cross strapping" thus allowing two or more locations covered by two separate beams to be connected in a single hop.

Disadvantages of C band

- **C band requires the use of larger dishes.** They can be quite cumbersome to install and are more expensive to acquire and transport.
- **C band systems share the same frequency bands as allocated to some terrestrial microwave systems.** As such, care must be taken when positioning C band antennas in areas where terrestrial microwave systems exist (for example TV or radio stations). For this reason, C band satellite transponder power is deliberately limited during the satellite's design and manufacture according to sharing criteria laid down by the ITU, leading to a requirement for larger dishes on the ground.

Advantages of Ku band

- **Ku band systems require smaller dishes.** Because of their higher satellite transponder power and higher frequencies, they can use smaller, cheaper antennas on the ground and therefore reduce startup and transport costs.
- **The smaller Ku Band dishes can be easily installed on almost any surface.** For example, they can be installed on the ground, mounted on roofs, or bolted to the side of buildings. This is an important consideration for areas with limited space.

Disadvantages of Ku band

- **Ku band systems are more affected by rainfall.** Because of their higher operating frequencies, they are usually considered unsuitable for mission critical applications in the tropics, unless specific measures are taken to provide for the added rain attenuation. For example, this can be overcome by using larger dishes. This drawback has also been slightly offset by the higher power satellites being manufactured today. As noted above, Ku band systems are gaining popularity even in the tropics for home use where users can survive a few hours of intermittent service a month.
- **Ku band satellite systems usually have smaller beams covering a small surface of the earth.** Therefore if you intend to cover two locations a large distance apart, within a single hop or with a broadcast system, you may need to consider C band systems.
- **Ku band bandwidth is more expensive that C band bandwidth.** As noted above, the savings in capital cost you gain using Ku band's smaller antennas may be negated by the higher operating costs imposed by high bandwidth prices.

Advantages of Ka band

- **Ka band dishes can be much smaller than Ku band dishes.** This is due to the even higher Ka band frequencies and higher satellite power. The smaller dishes translate to lower start up costs for equipment.

Disadvantages of Ka band

- **The higher frequencies of Ka band are significantly more vulnerable to signal quality problems caused by rainfall.** Therefore, Ka band VSATs are usually unsuitable for mission critical or high availability systems in the tropical and sub-tropical regions without the provision of measures to combat adverse weather conditions.
- **Ka-band systems will almost always require tracking antennas.** This adds to complexity and startup costs.
- **Ka band bandwidth is more expensive than C band or Ku band bandwidth.**
- **The Ka band is currently unavailable over Africa.**

Shared vs. dedicated bandwidth

It is critical for you to decide whether you will accept shared or dedicated bandwidth. Shared bandwidth refers to bandwidth that is shared with other customers of your service provider. Dedicated bandwidth is "committed" solely to you. Shared bandwidth is obviously cheaper than dedicated bandwidth because you are also sharing the cost of the bandwidth among other users. Unfortunately, some service providers pass off shared bandwidth as dedicated bandwidth and charge you rates equivalent to those for dedicated bandwidth. You therefore have to be clear about what you are buying.

Shared bandwidth is desirable when you will not be using all the bandwidth all the time. If your primary applications will be email and web surfing and you do not have many users, e.g. a community telecentre, then shared bandwidth may well work for you. However, if you have a large volume of users accessing the system throughout the day, or if you intend to run real time applications such as telephony or video conferencing, then you will need dedicated bandwidth.

There is one instance when you should consider shared capacity even when you have heavy users and real time applications. This is the situation in which you own the entire network. You would essentially be buying a chunk of dedicated bandwidth and then sharing its capacity among your network. The reasoning behind this is that if all VSATs are part of the same network, with the same profile of user, then you are likely to have instances when capacity would be unused. For instance, not all the VSATs in your network may be making voice calls or participating in video conferencing all the time. This method is especially suited to global organisations with offices in different time zones.

There are three key metrics you will need to consider when purchasing shared bandwidth:

The Contention Ratio

Contention is a term that comes from terrestrial Internet systems such as ***Digital Subscriber Link*** (***DSL***) and refers to sharing. The contention ratio is the number of users sharing the bandwidth. Obviously, the more users sharing the bandwidth, the less bandwidth you get if they are all online. For instance, if you are sharing bandwidth with a capacity of 1 Mbps among 20 customers (contention ratio of 20:1), then your maximum connection speed when all the customers are using the bandwidth is 50 kbps, equivalent to a dial up modem connection. If, however, the contention ratio is 50:1, or 50 customers sharing the connection, then your maximum speed when all customers are using the system is 20 kbps. As you can imagine, how much of the 1 Mbps promised by the service provider you actually get depends on the contention ratio. Contention is also called “over booking” or “over selling” capacity.

Committed Information Rate (CIR)

Even with shared bandwidth capacity, your service provider may guarantee you certain minimum capacity at all times. This guaranteed capacity is the CIR. In our example above using a contention ratio of 20:1, this CIR would be 50 kbps, even though you are quoted a bandwidth capacity of 1 Mbps.

Bursting capacity

Bursting refers to the ability of a VSAT system to utilise capacity above and beyond its normal allocation. Bursting is only possible when you purchase shared bandwidth. If your service provider has implemented bursting, a portion or all of the shared bandwidth capacity will be pooled. For instance, several portions of 1 Mbps may be pooled together. When other customers are not using their capacity, you may be able to ***burst***, or use more than your allocated capacity. Note that bursting also only occurs when there is ‘free’ or available capacity in the pool. The amount of additional or burst capacity to which any VSAT station sharing the pool is entitle to is limited to a set maximum, usually less than the total pool capacity to ensure that there is always capacity available for other VSAT stations.

In summary, when purchasing shared capacity, you should ask your service provider to specify the contention ratio, your CIR, and how much bursting capacity you can get.

TCP/IP factors over a satellite connection

A VSAT is often referred to as a ***long fat pipe network***. This term refers to factors that affect TCP/IP performance on any network that has relatively large bandwidth, but high latency. Most Internet connections in Africa and other parts of the developing world are via VSAT. Therefore, even if a university gets its connection via an ISP, this section might apply if the ISP's connection is via VSAT. The high latency in satellite networks is due to the long distance to the satellite and the constant speed of light. This distance adds about 520 ms to a packet’s round-trip time (RTT), compared to a typical RTT between Europe and the USA of about 140 ms.

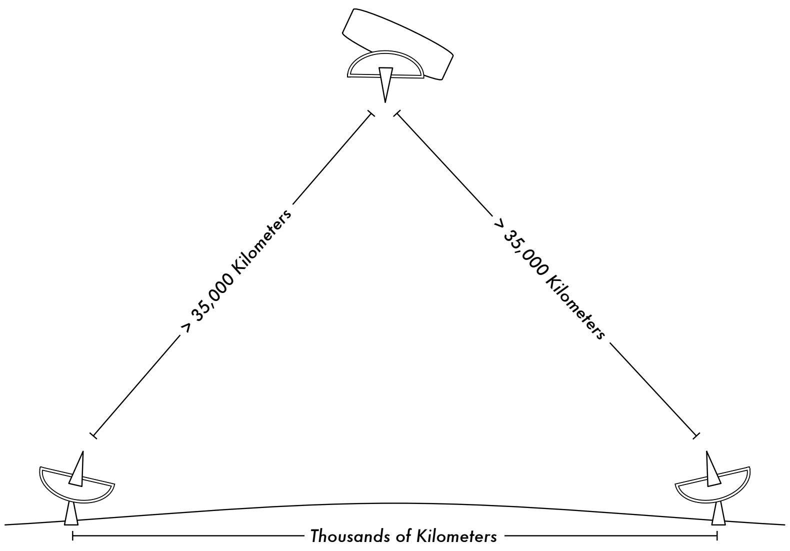

Figure 6.8: Due to the speed of light and long distances involved, a single ping packet can take more than 520 ms to be acknowledged over a VSAT link.

The factors that most significantly impact TCP/IP performance are **long RTT**, **large bandwidth delay product**, and **transmission errors**.

Generally speaking, operating systems that support modern TCP/IP implementations should be used in a satellite network. These implementations support the RFC1323 extensions:

- The ***window scale*** option for supporting large TCP window sizes (larger than 64KB).
- ***Selective acknowledgment*** (***SACK***) to enable faster recovery from transmission errors.
- Timestamps for calculating appropriate RTT and retransmission timeout values for the link in use.

Long round-trip time (RTT)

Satellite links have an average RTT of around 520ms to the first hop. TCP uses the slow-start mechanism at the start of a connection to find the appropriate TCP/IP parameters for that connection. Time spent in the slow-start stage is proportional to the RTT, and for a satellite link it means that TCP stays in slow-start mode for a longer time than would otherwise be the case. This drastically decreases the throughput of short-duration TCP connections. This is can be seen in the way that a small website might take surprisingly long to load, but when a large file is transferred acceptable data rates are achieved after awhile.

Furthermore, when packets are lost, TCP enters the congestion-control phase, and owing to the higher RTT, remains in this phase for a longer time, thus reducing the throughput of both short- and long-duration TCP connections.

Large bandwidth-delay product

The amount of data in transit on a link at any point of time is the product of bandwidth and the RTT. Because of the high latency of the satellite link, the bandwidth-delay product is large. TCP/IP allows the remote host to send a certain amount of data in advance without acknowledgment. An acknowledgment is usually required for all incoming data on a TCP/IP connection. However, the remote host is always allowed to send a certain amount of data without acknowledgment, which is important to achieve a good transfer rate on large bandwidth-delay product connections. This amount of data is called the ***TCP window size***. The window size is usually 64KB in modern TCP/IP implementations.

On satellite networks, the value of the bandwidth-delay product is important. To utilise the link fully, the window size of the connection should be equal to the bandwidth-delay product. If the largest window size allowed is 64KB, the maximum theoretical throughput achievable via satellite is (window size) / RTT, or 64KB / 520 ms. This gives a maximum data rate of 123KB/s, which is 984 Kbps, regardless of the fact that the capacity of the link may be much greater.

Each TCP segment header contains a field called ***advertised window***, which specifies how many additional bytes of data the receiver is prepared to accept. The advertised window is the receiver's current available buffer size. The sender is not allowed to send more bytes than the advertised window. To maximise performance, the sender should set its send buffer size and the receiver should set its receive buffer size to no less than the bandwidth-delay product. This buffer size has a maximum value of 64KB in most modern TCP/IP implementations.

To overcome the problem of TCP/IP stacks from operating systems that don't increase the window size beyond 64KB, a technique known as ***TCP acknowledgment spoofing*** can be used (see Performance Enhancing Proxy, below).

Transmission errors

In older TCP/IP implementations, packet loss is always considered to have been caused by congestion (as opposed to link errors). When this happens, TCP performs congestion avoidance, requiring three duplicate ACKs or slow start in the case of a timeout. Because of the long RTT value, once this congestion-control phase is started, TCP/IP on satellite links will take a longer time to return to the previous throughput level. Therefore errors on a satellite link have a more serious effect on the performance of TCP than do errors on low latency links. To overcome this limitation, mechanisms such as ***Selective Acknowledgment*** (***SACK***) have been developed. SACK specifies exactly those packets that have been received, allowing the sender to retransmit only those segments that are missing because of link errors.

The Microsoft Windows 2000 TCP/IP Implementation Details White Paper states

> "*Windows 2000 introduces support for an important performance feature known as Selective Acknowledgment (SACK). SACK is especially important for connections using large TCP window sizes.*"

SACK has been a standard feature in Linux and BSD kernels for quite some time. Be sure that your Internet router and your ISP's remote side both support SACK.

Implications for universities

If a site has a 512 Kbps connection to the Internet, the default TCP/IP settings are likely sufficient, because a 64 KB window size can fill up to 984 Kbps. But if the university has more than 984 Kbps, it might in some cases not get the full bandwidth of the available link due to the "long fat pipe network" factors discussed above. What these factors really imply is that they prevent a single machine from filling the entire bandwidth. This is not a bad thing during the day, because many people are using the bandwidth. But if, for example, there are large scheduled downloads at night, the administrator might want those downloads to make use of the full bandwidth, and the "long fat pipe network" factors might be an obstacle. This may also become critical if a significant amount of your network traffic routes through a single tunnel or VPN connection to the other end of the VSAT link.

Administrators might consider taking steps to ensure that the full bandwidth can be achieved by tuning their TCP/IP settings. If a university has implemented a network where all traffic has to go through the proxy (enforced by network layout), then the only machines that make connections to the Internet will be the proxy and mail servers.

For more information, see *http://www.psc.edu/networking/perf_tune.html* .

Performance-enhancing proxy (PEP)

The idea of a Performance-enhancing proxy is described in RFC3135 (see *http://www.ietf.org/rfc/rfc3135*), and would be a proxy server with a large disk cache that has RFC1323 extensions, among other features. A laptop has a TCP session with the PEP at the ISP. That PEP, and the one at the satellite provider, communicate using a different TCP session or even their own proprietary protocol. The PEP at the satellite provider gets the files from the web server. In this way, the TCP session is split, and thus the link characteristics that affect protocol performance (long fat pipe factors) are overcome (by TCP acknowledgment spoofing, for example). Additionally, the PEP makes use of proxying and pre-fetching to accelerate web access further.

Such a system can be built from scratch using Squid, for example, or purchased "off the shelf" from a number of vendors. Here are some useful links to information about building your own performance enhancing proxy.

- "Enabling High Performance Data Transfers," Pittsburgh Supercomputing Center: *http://www.psc.edu/networking/projects/tcptune/*
- RFC3135, "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations."
- ***PEPsal***, a Performance Enhancing Proxy implementation released under the GPL: *http://sourceforge.net/projects/pepsal/*

Resources

- Adzapper: *http://adzapper.sourceforge.net/*
- Authentication in Squid, *http://www.squid-cache.org/Doc/FAQ/FAQ-23.html*
- Cache heirarchies with Squid, *http://squid-docs.sourceforge.net/latest/html/c2075.html*
- DansGuard: *http://dansguardian.org/*
- dnsmasq caching DNS and DHCP server, *http://thekelleys.org.uk/dnsmasq/doc.html*
- Enhancing International World Wide Web Access in Mozambique Through the Use of Mirroring and Caching Proxies, *http://www.isoc.org/inet97/ans97/cloet.htm*
- Fluff file distribution utility, *http://www.bristol.ac.uk/fluff/*
- Lawrence Berkeley National Laboratory's TCP Tuning Guide, *http://dsd.lbl.gov/TCP-tuning/background.html*
- Linux Advanced Routing & Traffic Control HOWTO, *http://lartc.org/lartc.html*
- Microsoft Internet Security and Acceleration Server, *http://www.microsoft.com/isaserver/*
- Microsoft ISA Server Firewall and Cache resource site, *http://www.isaserver.org/*

- Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations, *http://www.ietf.org/rfc/rfc3135*
- PF: The OpenBSD Packet Filter FAQ, *http://www.openbsd.org/faq/pf/*
- Pittsburgh Supercomputing Center's guide to Enabling High Performance Data Transfers, *http://www.psc.edu/networking/perf_tune.html*
- SquidGuard: *http://www.squidguard.org/*
- Squid web proxy cache, *http://squid-cache.org/*
- TCP Tuning and Network Troubleshooting by Brian Tierney, *http://www.onlamp.com/pub/a/onlamp/2005/11/17/tcp_tuning.html*
- ViSolve Squid configuration manual, *http://www.visolve.com/squid/squid24s1/contents.php*
- The VSAT Buyer's guide, *http://ictinafrica.com/vsat/*
- Wessels, Duane. *Squid: The Definitive Guide*. O'Reilly Media (2004). *http://squidbook.org/*