B

Samba Performance Tuning

This appendix discusses various ways of performance tuning and system sizing with Samba. *Performance tuning* is the art of finding bottlenecks and adjusting to eliminate them. *Sizing* is the practice of eliminating bottlenecks by spending money to avoid having them in the first place. Normally, you won't have to worry about either with Samba. On a completely untuned server, Samba will happily support a small community of users. However, on a properly tuned server, Samba will support at least twice as many users. This chapter is devoted to outlining various performance-tuning and sizing techniques that you can use if you want to stretch your Samba server to the limit.

A Simple Benchmark

How do you know if you're getting reasonable performance? A simple benchmark is to compare Samba with FTP. Table B-1 shows the throughput, in kilobytes per second, of a pair of servers: a medium-size Sun SPARC Ultra and a small Linux Pentium server. Numbers are reported in kilobytes per second (KB/s).

Table B-1. Sample Benchmark Benchmarks

Command	FTP	Untuned Samba	Tuned Samba
Sparc get	1014.5	645.3	866.7
Sparc put	379.8	386.1	329.5
Pentium get	973.27	N/A	725
Pentium put	1014.5	N/A	1100

If you run the same tests on your server, you probably won't see the same numbers. However, you *should* see similar ratios of Samba to FTP, probably in the range of 68 to 80 percent. It's not a good idea to base *all* of Samba's throughput

against FTP. The golden rule to remember is this: if Samba is much slower than FTP, it's time to tune it.

You might think that an equivalent test would be to compare Samba to NFS. In reality, however, it's much less useful to compare their speeds. Depending entirely on whose version of NFS you have and how well it's tuned, Samba can be slower or faster than NFS. We usually find that Samba is faster, but watch out; NFS uses a different algorithm from Samba, so tuning options that are optimal for NFS may be detrimental for Samba. If you run Samba on a well-tuned NFS server, Samba may perform rather badly.

A more popular benchmark is Ziff-Davis' *NetBench,* a simulation of many users on client machines running word processors and accessing data on the SMB server. It's not a prefect measure (each NetBench client does about ten times the work of a normal user on our site), but it is a fair comparison of similar servers. In tests performed by Jeremy Allison in November 1998, Samba 2.0 on a SGI multiprocessor outperformed NT Server 4.0 (Patch Level 2) on an equivalent high-end Compaq. This was confirmed and strengthened by a Sm@rt Reseller test of NT and Linux on identical hardware in February 1999.

In April 1999, the Mindcraft test lab released a report about a test showing that Samba on a four-processor Linux machine was significantly slower than native file serving on the same machine running Windows NT. While the original report was slammed by the Open Source community because it was commissioned by Microsoft and tuned the systems to favor Windows NT, a subsequent test was fairer and generally admitted to reveal some areas where Linux needed to improve its performance, especially on multiprocessors. Little was said about Samba itself. Samba is known to scale well on multiprocessors, and exceeds 440MB/s on a four-processor SGI O200, beating Mindcraft's 310MB/s.

Relative performance will probably change as NT and PC hardware get faster, of course, but Samba is improving as well. For example, Samba 1.9.18 was faster only with more than 35 clients. Samba 2.0, however, is faster regardless of the number of clients. In short, Samba is very competitive with the best networking software in the industry, and is only getting better.

As we went to press, Andrew Tridgell released the alpha-test version suite of benchmarking programs for Samba and SMB networks. Expect even more work on performance from the Samba team in the future.

Samba Tuning

That being said, let's discuss how you can take an already fast networking package and make it even faster.

Benchmarking

Benchmarking is an arcane and somewhat black art, but the level of expertise needed for simple performance tuning is fairly low. Since the Samba server's goal in life is to transfer files, we will examine only throughput, not response time to particular events, under the benchmarking microscope. After all, it's relatively easy to measure file transfer speed, and Samba doesn't suffer too badly from response-time problems that would require more sophisticated techniques.

Our basic strategy for this work will be:

- Find a reasonably-sized file to copy and a program that reports on copy speeds, such as *smbclient*.
- Find a quiet (or typical) time to do the test.
- Pre-run each test a few times to preload buffers.
- Run tests several times and watch for unusual results.
- Record each run in detail.
- Compare the average of the valid runs to expected values.

After establishing a baseline using this method, we can adjust a single parameter and do the measurements all over again. An empty table for your tests is provided at the end of this chapter.

Things to Tweak

There are literally thousands of Samba setting combinations that you can use in search of that perfect server. Those of us with lives outside of system administration, however, can narrow down the number of options to those listed in this section, which are the most likely to affect overall throughput. They are presented roughly in order of impact.

Log level

This is an obvious one. Increasing the logging level (`log level` or `debug level` configuration options) is a good way to debug a problem, unless you happen to be searching for a performance problem! As mentioned in Chapter 4, *Disk Shares*, Samba produces a ton of debugging messages at level 3 and above, and writing them to disk or syslog is a slow operation. In our *smbclient/ftp* tests, raising the log level from 0 to 3 cut the untuned `get speed` from 645.3 to 622.2KB/s, or roughly 5 percent. Higher log levels were even worse.

Socket options

The next thing to look at are the `socket options` configuration options. These are really host system tuning options, but they're set on a per-connection basis,

and can be reset by Samba on the sockets it employs by adding `socket options = option` to the `[global]` section of your *smb.conf* file. Not all of these options are supported by all vendors; check your vendor's manual pages on *setsockopt*(1) or *socket*(5) for details.

The main options are:

`TCP_NODELAY`
: Have the server send as many packets as necessary to keep delay low. This is used on telnet connections to give good response time, and is used—somewhat counter-intuitively—to get good speed even when doing small requests or when acknowledgments are delayed (as seems to occur with Microsoft TCP/IP). This is worth a 30–50 percent speedup by itself. Incidentally, in Samba 2.0.4, `socket options = TCP_NODELAY` became the default value for that option.

`IPTOS_LOWDELAY`
: This is another option that trades off throughput for lower delay, but which affects routers and other systems, not the server. All the IPTOS options are new; they're not supported by all operating systems and routers. If they are supported, set `IPTOS_LOWDELAY` whenever you set `TCP_NODELAY`.

`SO_SNDBUF and SO_RCVBUF`
: The send and receive buffers can often be the reset to a value higher than that of the operating system. This yields a marginal increase of speed (until it reaches a point of diminishing returns).

`SO_KEEPALIVE`
: This initiates a periodic (four-hour) check to see if the client has disappeared. Expired connections are addressed somewhat better with Samba's `keepalive` and `dead time` options. All three eventually arrange to close dead connections, returning unused memory and process-table entries to the operating system.

There are several other socket options you might look at, (e.g., `SO_SNDLOWAT`), but they vary in availability from vendor to vendor. You probably want to look at *TCP/IP Illustrated* if you're interested in exploring more of these options for performance tuning with Samba.

read raw and write raw

These are important performance configuration options; they enable Samba to use large reads and writes to the network, of up to 64KB in a single SMB request. They also require the largest SMB packet structures, `SMBreadraw` and `SMBwriteraw`, from which the options take their names. Note that this is not the same as a Unix *raw read*. This Unix term usually refers to reading disks without using the files system, quite a different sense from the one described here for Samba.

In the past, some client programs failed if you tried to use `read raw`. As far as we know, no client suffers from this problem any more. Read and write raw default to `yes`, and should be left on unless you find you have one of the buggy clients.

Opportunistic locking

Opportunistic locks, or *oplocks*, allow clients to cache files locally, improving performance on the order of 30 percent. This option is now enabled by default. For read-only files, the `fake oplocks` provides the same functionality without actually doing any caching. If you have files that cannot be cached, *oplocks* can be turned off.

Database files should never be cached, nor should any files that are updated both on the server and the client and whose changes must be immediately visible. For these files, the `veto oplock files` option allows you to specify a list of individual files or a pattern containing wildcards to avoid caching. *oplocks* can be turned off on a share-by-share basis if you have large groups of files you don't want cached on clients. See Chapter 5, *Browsing and Advanced Disk Shares*, for more information on opportunistic locks.

IP packet size (MTU)

Networks generally set a limit to the size of an individual transmission or packet This is called the Maximum Segment Size, or if the packet header size is included, the Maximum Transport Unit (MTU). This MTU is not set by Samba, but Samba needs to use a `max xmit` (write size) bigger than the MTU, or throughput will be reduced. This is discussed in further detail in the following note. The MTU is normally preset to 1500 bytes on an Ethernet and 4098 bytes on FDDI. In general, having it too low cuts throughput, and having it too high causes a sudden performance dropoff due to fragmentation and retransmissions.

If you are communicating over a router, some systems will assume the router is a serial link (e.g., a T1) and set the MTU to more or less 536 bytes. Windows 95 makes this mistake, which causes nearby clients to perform well, but clients on the other side of the router to be noticeably slower. If the client makes the opposite error and uses a large MTU on a link which demands a small one, the packets will be broken up into fragments. This slows transfers slightly, and any networking errors will cause multiple fragments to be retransmitted, which slows Samba significantly. Fortunately, you can modify the Windows MTU size to prevent either error. To understand this in more detail, see "The Windows 95 Networking Frequently Asked Questions (FAQ)" at *http://www.stanford.edu/~llurch/win95netbugs/faq.html*, which explains how to override the Windows MTU and Window Size.

The TCP receive window

TCP/IP works by breaking down data into small packets that can be transmitted from one machine to another. When each packet is transmitted, it contains a checksum that allows the receiver to check the packet data for potential errors in transmission. Theoretically, when a packet is received and verified, an acknowledgment packet should be sent back to the sender that essentially says, "Everything arrived intact: please continue."

In order to keep things moving, however, TCP accepts a range (window) of packets that allows a sender to keep transmitting without having to wait for an acknowledgment of every single packet. (It can then bundle a group of acknowledgments and transmit them back to the sender at the same time.) In other words, this receive window is the number of bytes that the sender can transmit before it has to stop and wait for a receiver's acknowledgment. Like the MTU, it is automatically set based on the type of connection. Having the window too small causes a lot of unnecessary waiting for acknowledgment messages. Various operating systems set moderate buffer sizes on a per-socket basis to keep one program from hogging all the memory.

The buffer sizes are assigned in bytes, such as `SO_SNDBUF=8192` in the `socket options` line. Thus, an example `socket options` configuration option is:

```
socket options = SO_SNDBUF=8192
```

Normally, one tries to set these socket options higher than the default: 4098 in SunOS 4.1.3 and SVR4, and 8192–16384 in AIX, Solaris, and BSD. 16384 has been suggested as a good starting point: in a non-Samba test mentioned in Stevens' book, it yielded a 40 percent improvement. You'll need to experiment, because performance will fall off again if you set the sizes too high. This is illustrated in Figure B-1, a test done on a particular Linux system.

Setting the socket options `O_SNDBUF` and `SO_RCVBUF` to less than the default is inadvisable. Setting them higher improves performance, up to a network-specific limit. However, once you exceed that limit, performance will abruptly level off.

max xmit

In Samba, the option that is directly related with the MTU and window size is `max xmit`. This option sets the largest block of data Samba will try to write at any one time. It's sometimes known as the *write size*, although that is not the name of the Samba configuration option.

Because the percentage of each block required for overhead falls as the blocks get larger, max xmit is conventionally set as large as possible. It defaults to the

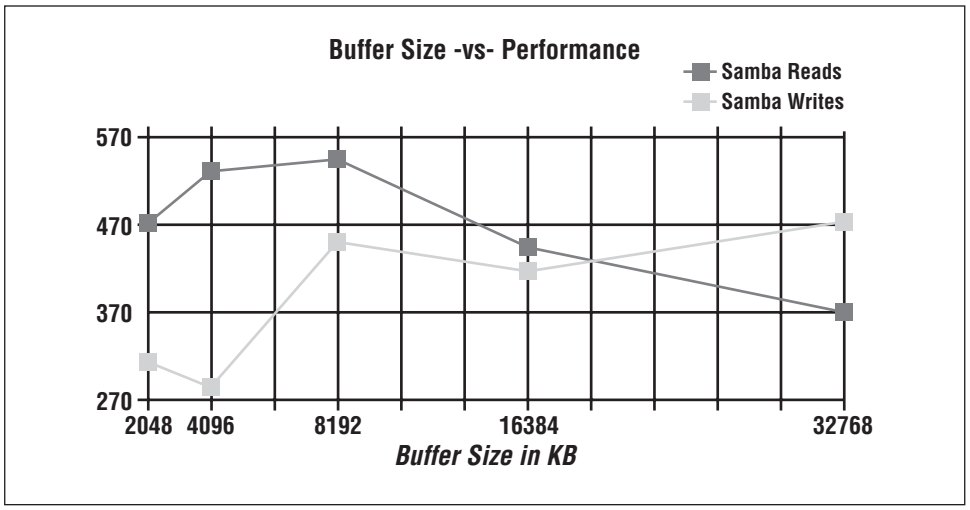

Figure B-1. SO_SNDBUF size and performance

protocol's upper limit, which is 64 kilobytes. The smallest value that doesn't cause significant slowdowns is 2048. If it is set low enough, it will limit the largest packet size that Samba will be able to negotiate. This can be used to simulate a small MTU if you need to test an unreliable network connection. However, such a test should not be used in production for reducing the effective MTU.

read size

If `max xmit` is commonly called the write size, you'd expect `read size` to be the maximum amount of data that Samba would want to read from the client via the network. Actually, it's not. In fact, it's an option to trigger *write ahead*. This means that if Samba gets behind reading from the disk and writing to the network (or vice versa) by the specified amount, it will start overlapping network writes with disk reads (or vice versa).

The read size doesn't have a big performance effect on Unix, unless you set its value quite small. At that point, it causes a detectable slowdown. For this reason, it defaults to 2048 and can't be set lower than 1024.

read prediction

Besides being counterintuitive, this option is also obsolete. It enables Samba to read ahead on files opened read only by the clients. The option is disabled in Samba 2.0 (and late 1.9) Because it interferes with opportunistic locking.

Other Samba Options

The following Samba options will affect performance if they're set incorrectly, much like the debug level. They're mentioned here so you will know what to look out for:

`hide files`
: Providing a pattern to identify files hidden by the Windows client `hide files` will result in any file matching the pattern being passed to the client with the DOS hidden attribute set. It requires a pattern match per file when listing directories, and slows the server noticeably.

`lpq cache time`
: If your `lpq` (printer queue contents) command takes a long time to complete, you should increase `lpq cache time` to a value higher than the actual time required for `lpq` to execute, so as to keep Samba from starting a new query when one's already running. The default is 10 seconds, which is reasonable.

`strict locking`
: Setting the `strict locking` option causes Samba to check for locks on every access, not just when asked to by the client. The option is primarily a bug-avoidance feature, and can prevent ill-behaved DOS and Windows applications from corrupting shared files. However, it is slow and should typically be avoided.

`strict sync`
: Setting `strict sync` will cause Samba to write each packet to disk and wait for the write to complete whenever the client sets the sync bit in a packet. Windows 98 Explorer sets the bit in all packets transmitted, so if you turn this on, anyone with Windows 98 will think Samba servers are horribly slow.

`sync always`
: Setting `sync always` causes Samba to flush every write to disk. This is good if your server crashes constantly, but the performance costs are immense. SMB servers normally use oplocks and automatic reconnection to avoid the ill effects of crashes, so setting this option is not normally necessary.

`wide links`
: Turning off `wide links` prevents Samba from following symbolic links in one file share to files that are not in the share. It is turned on by default, since following links in Unix is not a security problem. Turning it off requires extra processing on every file open. If you do turn off wide links, be sure to turn on `getwd cache` to cache some of the required data.

There is also a `follow symlinks` option that can be turned off to prevent following any symbolic links at all. However, this option does not pose a performance problem.

`getwd cache`
: This option caches the path to the current directory, avoiding long tree-walks to discover it. It's a nice performance improvement on a printer server or if you've turned off `wide links`.

Our Recommendations

Here's an *smb.conf* file that incorporates the recommended performance enhancements so far. Comments have been added on the right side.

```
[global]
    log level = 1                       # Default is 0
    socket options = TCP_NODELAY IPTOS_LOWDELAY
    read raw = yes                      # Default
    write raw = yes                     # Default
    oplocks = yes                       # Default
    max xmit = 65535                    # Default
    dead time = 15                      # Default is 0
    getwd cache = yes
    lpq cache = 30
[okplace]
    veto oplock files = this/that/theotherfile
[badplace]
    oplocks = no
```

Sizing Samba Servers

Sizing is a way to prevent bottlenecks before they occur. The preferred way to do this is to know how many requests per second or how many kilobytes per second the clients will need, and ensure that all the components of the server provide at least that many.

The Bottlenecks

The three primary bottlenecks you should worry about are CPU, disk I/O, and the network. For most machines, CPUs are rarely a bottleneck. A single Sun SPARC 10 CPU can start (and complete) between 700 and 800 I/O operations a second, giving approximately 5,600 to 6,400KB/s of throughput when the data averages around 8KBs (a common buffer size). A single Intel Pentium 133 can do less only because of somewhat slower cache and bus interfaces, not due to lack of CPU power. Purpose-designed Pentium servers, like some Compaq servers, will be able to start 700 operations per CPUs, on up to four CPUs.

Too little memory, on the other hand, can easily be a bottleneck; each Samba process will use between 600 and 800KB on Intel Linux, and more on RISC CPUs. Having less will cause an increase in virtual memory paging and therefore a performance hit. On Solaris, where it has been measured, *smbd* will use 2.6 MB for program and shared libraries, plus 768KB for each connected client. *nmbd* occupies 2.1 MB, plus 496KB extra for its (single) auxiliary process.

Hard disks will always bottleneck at a specific number of I/O operations per second: for example, each 7200 RPM SCSI disk is capable of performing 70 operations per second, for a throughput of 560KB/s; a 4800 RPM disk will perform fewer than 50, for a throughput of 360KB/s. A single IDE disk will do still fewer. If the disks are independent, or striped together in a RAID 1 configuration, they will each peak out at 400 to 560KB/s and will scale linearly as you add more. Note that this is true only of RAID 1. RAID levels other than 1 (striping) add extra overhead.

Ethernets (and other networks) are obvious bottleneck: a 10 Mb/s (mega*bits*/second) Ethernet will handle around 1100KB/s (kilo*bytes*/s) using 1500-byte packets A 100 Mb/s Fast Ethernet will bottleneck below 65,000KB/s with the same packet size. FDDI, at 155 Mb/s will top out at approximately 6,250KB/s, but gives good service at even 100 percent load and transmits much larger packets (4KB).

ATM should be much better, but as of the writing of this book it was too new to live up to its potential; it seems to deliver around 7,125 Mb/s using 9KB packets.

Of course, there can be other bottlenecks: more than one IDE disk per controller is not good, as are more than three 3600 SCSI-I disks per slow/narrow controller, or more than three 7200 SCSI-II disks per SCSI-II fast/wide controller. RAID 5 is also slow, as it requires twice as many writes as independent disks or RAID 1.

After the second set of Ethernets and the second disk controller, start worrying about bus bandwidth, especially if you are using ISA/EISA buses.

Reducing Bottlenecks

From the information above we can work out a model that will tell us the maximum capability of a given machine. The data is mostly taken from Brian Wong's *Configuration and Capacity Planning for Solaris Servers*,* so there is a slight Sun bias to our examples.

A word of warning: this is not a complete model. Don't assume that this model will predict every bottleneck or even be within 10 percent in its estimates. A model to predict performance instead of one to warn you of bottlenecks would be

* See Wong. Brian L, *Configuration and Capacity Planning for Solaris Servers*, Englewood Cliffs, NJ (Sun/Prentice-Hall), 1997, ISBN 0-13-349952-9.

much more complex and would contain rules like "not more than three disks per SCSI chain". (A good book on real models is Raj Jain's *The Art of Computer Systems Performance Analysis.*[*]) With that warning, we present the system in Figure B-2.

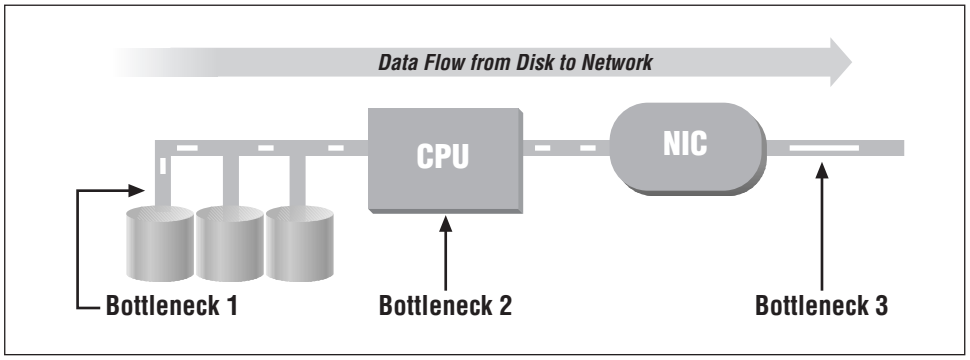

Figure B-2. Data flow through a Samba server, with possible bottlenecks

The flow of data should be obvious. For example, on a read, data flows from the disk, across the bus, through or past the CPU, and to the network interface card (NIC). It is then broken up into packets and sent across the network. Our strategy here is to follow the data through the system and see what bottlenecks will choke it off. Believe it or not, it's rather easy to make a set of tables that list the maximum performance of common disks, CPUs, and network cards on a system. So that's exactly what we're going to do.

Let's take a concrete example: a Linux Pentium 133 MHz machine with a single 7200 RPM data disk, a PCI bus, and a 10-Mb/s Ethernet card. This is a perfectly reasonable server. We start with Table B-2, which describes the hard drive—the first potential bottleneck in the system.

Table B-2. Disk Throughput

Disk RPM	I/O Operations/second	KB/second
7200	70	560
4800	60	480
3600	40	320

Disk throughput is the number of kilobytes of data that a disk can transfer per second. It is computed from the number of 8KB I/O operations per second a disk can perform, which in turn is strongly influenced by disk RPM and bit density. In

[*] See Jain. Raj, *The Art of Computer Systems Performance Analysis*, New York, NY (John Wiley and Sons), 1991, ISBN 0-47-150336-3.

effect, the question is: how much data can pass below the drive heads in one second? With a single 7200 RPM disk, the example server will give us 70 I/O operations per second at roughly 560KB/s.

The second possible bottleneck is the CPU. The data doesn't actually flow through the CPU on any modern machines, so we have to compute throughput somewhat indirectly.

The CPU has to issue I/O requests and handle the interrupts coming back, then transfer the data across the bus to the network card. From much past experimentation, we know that the overhead that dominates the processing is consistently in the filesystem code, so we can ignore the other software being run. We compute the throughput by just multiplying the (measured) number of file I/O operations per second that a CPU can process by the same 8K average request size. This gives us the results shown in Table B-3.

Table B-3. CPU Throughput

CPU	I/O Operations/second	KB/second
Intel Pentium 133	700	5,600
Dual Pentium 133	1,200	9,600
Sun SPARC II	660	5,280
Sun SPARC 10	750	6,000
Sun Ultra 200	2,650	21,200

Now we put the disk and the CPU together: in the Linux example, we have a single 7200 RPM disk, which can give us 560KB/s, and a CPU capable of starting 700 I/O operations, which could give us 5600KB/s. So far, as you would expect, our bottleneck is clearly going to be the hard disk.

The last potential bottleneck is the network. If the network speed is below 100 Mb/s, the bottleneck will be the network speed. After that, the design of the network card is more likely to slow us down. Table B-4 shows us the average throughput of many types of data networks. Although network speed is conventionally measured in bits per second, Table B-4 lists bytes per second to make comparison with the disk and CPU (Table B-2 and Table B-3) easier.

Table B-4. Network Throughput

Network Type	KB/second
ISDN	16
T1	197
Ethernet 10m	1,113
Token ring	1,500

Table B-4. Network Throughput (continued)

Network Type	KB/second
FDDI	6,250
Ethernet 100m	6,500[a]
ATM 155	7,125[a]

[a] These will increase. For example, Crays, Sun Ultras, and DEC/Compaq Alphas already have bettered these figures.

In the running example, we have a bottleneck at 560KB/s due to the disk. Table B-4 shows us that a standard 10 megabit per second Ethernet (1,113KB/s) is far faster than the disk. Therefore, the hard disk is still the limiting factor. (This scenario, by the way, is very common.) Just by looking at the tables, we can predict that small servers won't have CPU problems, and that large ones with multiple CPUs will support striping and multiple Ethernets long before they start running out of CPU power. This, in fact, is exactly what happens.

Practical Examples

An example from *Configuration and Capacity Planning for Solaris Servers* (Wong) shows that a dual-processor SPARCstation 20/712 with four Ethernets and six 2.1 GB disks will spend all its time waiting for the disks to return some data. If it was loaded with disks (Brian Wong suggests as many as 34 of them), it would still be held below 1,200KB/s by the Ethernet cards. To get the performance the machine is capable of, we would need to configure multiple Ethernets, 100 Mbps Fast Ethernet, or 155 Mbps FDDI.

The progression you'd work through to get that conclusion looks something like Table B-5.

Table B-5. Tuning a Medium-Sized Server

Machine	Disk Throughput	CPU Throughput	Network Throughput	Actual Throughput
Dual SPARC 10, 1 disk	560	6000	1,113	560
Add 5 more disks	3,360	6000	1,113	1,113
Add 3 more Ethernets	3,360	16000	4,452	3,360
Change to using a 20-disk array	11,200	6000	4,452	4,452
Use dual 100 Mbps ether	11,200	6000	13,000	11,200

Initially, the bottleneck is the disk with only 560 MB/s of throughput available. Our solution is to add five more disks. This gives us more throughput on the disks than on the Ethernet, so then the Ethernet becomes the problem. Consequently, as

we continue to expand, we go back and forth several times between these two. As you add disks, CPUs, and network cards, the bottleneck moves. Essentially, the strategy is to add more equipment to try to avoid each bottleneck until you reach your target performance, or (unfortunately) you either can't add any more or run out of money.

Our experience bears out this kind of calculation; a large SPARC 10 file server that one author maintained was quite capable of saturating an Ethernet plus about a third of an FDDI ring when using two processors. It did nearly as well with a single processor, albeit with a fast operating system and judicious over-optimization.

The same process applies to other brands of purpose-designed servers. We found the same rules applied to DECstation 2100s as to the newest Alphas or Compaqs, old MIPS 3350s and new SGI O2s. In general, a machine offering multi-CPU server configurations will have enough bus bandwidth and CPU power to reliably bottleneck on hard disk I/O when doing file service. As one would hope, considering the cost!

How Many Clients can Samba Handle?

Well, that depends entirely on how much data each user consumes. A small server with three SCSI-1 disks, which can serve about 960KB/s of data, will support between 36 and 80 clients in an ordinary office environment where they are typically loading, and saving equal-sized spreadsheets or word processing documents (36 clients×2.3 transfers/second×12k file 1 MB/s).

On the same server in a development environment with programmers running a fairly heavy edit-compile-test cycle, one can easily see requests for 1 MB/s, limiting the server to 25 or fewer clients. To take this a bit further, an imaging system whose clients each require 10 MB/s will perform poorly no matter how big a server is if they're all on a 10 MB/s Ethernet. And so on.

If you don't know how much data an average user consumes, you can size your Samba servers by patterning them after existing NFS, Netware, or LAN Manager servers. You should be especially careful that the new servers have as many disks and disk controllers as the ones you've copied. This technique is appropriately called "punt and hope."

If you know how many clients an existing server can support, you're in *much* better shape. You can analyze the server to see what its maximum capacity is and use that to estimate how much data they must be demanding. For example, if serving home directories to 30 PCs from a PC server with two IDE disks is just too slow, and 25 clients is about right, then you can safely assume you're bottlenecked on

Ethernet I/O (approximately 375KB) rather than disk I/O (up to 640KB). If so, you can then conclude that the clients are demanding 15 (that is, 375/25)KB/s on average.

Supporting a new lab of 75 clients will mean you'll need 1,125KB/s, spread over multiple (preferably three) Ethernets, and a server with at least three 7200 RPM disks and a CPU capable of keeping up. These requirements can be met by a Pentium 133 or above with the bus architecture to drive them all at full speed (e.g., PCI).

A custom-built PC server or a multiprocessor-capable workstation like a Sun Sparc, a DEC/Compaq Alpha, an SGI, or the like, would scale up easier, as would a machine with fast Ethernet, plus a switching hub to drive the client machines on individual 10 MB/s Ethernets.

How to guess

If you have no idea at all what you need, the best thing is to try to guess based on someone else's experience. Each individual client machine can average from less than 1 I/O per second (normal PC or Mac used for sales/accounting) to as much as 4 (fast workstation using large applications). A fast workstation running a compiler can happily average 3-4 MB/s in data transfer requests, and an imaging system can demand even more.

Our recommendation? Spy on someone with a similar configuration and try to estimate their bandwidth requirements from their bottlenecks and the volume of the screams from their users. We also recommend Brian Wong's *Configuration and Capacity Planning for Solaris Servers*. While he uses Sun Solaris foremost in his examples, his bottlenecks are disks and network cards, which are common among all the major vendors. His tables for FTP servers also come very close to what we calculated for Samba servers, and make a good starting point.

Measurement Forms

Table B-6 and Table B-7 are empty tables that you can use for copying and recording data. The bottleneck calculation in the previous example can be done in a spreadsheet, or manually with Table B-8. If Samba is as good as or better than FTP, and if there aren't any individual test runs that are much different from the average, you have a well-configured system. If loopback isn't much faster than anything else, you have a problem with your TCP/IP software. If both FTP and Samba are slow, you probably have a problem with your networking: a faulty Ethernet card will produce this, as will accidentally setting an Ethernet card to half-duplex when it's not connected to a half-duplex hub. Remember that CPU and disk speeds are commonly measured in bytes, network speeds in bits.

We've included columns for both bytes and bits in the tables. In the last column, we compare results to 10 Mb/s because that's the speed of a traditional Ethernet.

Table B-6. Ethernet Interface to Same Host: FTP

Run No	Size in Bytes	Time (sec)	Bytes/sec	Bits/sec	% of 10 Mb/s
1					
2					
3					
4					
5					
Average:					
Deviation:					

Table B-7. Ethernet Interface to Same Host: FTP

Run No	Size in Bytes	Time, sec	Bytes/sec	Bits/sec	% of 10 Mb/s
1					
2					
3					
4					
5					
Average:					
Deviation:					

Table B-8. Bottleneck Calculation Table

CPU	CPU Throughput	Number of Disks	Disk Throughput	Number of Networks	Network Throughput	Total Throughput

In Table B-8:

- CPU throughput = (KB/second from Figure 6-5)×(number of CPUs)
- Disk throughput = (KB/second from Figure 6-4)×(number of disks)
- Network throughput = (KB/second from Figure 6-6)×(number of networks)
- Total throughput = min (Disk, CPU, and Network throughput)

A typical test, in this case for an FTP `get`, would be entered as in Table B-9:

Table B-9. Ethernet Interface to Same Host: FTP

Run No	Size in Bytes	Time, sec	Bytes/sec	Bits/sec	% of 10 Mb/s
1	1812898	2.3	761580		
2		2.3	767820		
3		2.4	747420		
4		2.3	760020		
5		2.3	772700		
Average:		2.32	777310	6218480	62
Deviation:		0.04			

The Sparc example we used earlier would look like Table B-10.

Table B-10. Sparc 20 Example, Redux

CPU	CPU Throughput	Number of Disks	Disk Throughput	Number of Networks	Network Throughput	Total Throughput
2	6,000	1	560	1 10base2	1,113	560
2	6,000	6	3,360	1	1,113	1,113
2	6,000	6	3,360	4 10base2	4,452	3,360
2	6,000	20	11,200	4	4,452	4,452
2	6,000	20	11,200	2 100base2	13,000	11,200