Measuring Firewall Performance

Andrew Molitor

amolitor@network.com

ABSTRACT

In this paper, we develop some terminology to describe performance measurements, and develop an experiment to compare packet filtering against a proxy approach. The single limited case of FTP transactions is our primary focus. The results suggest that the maximum rate at which FTP sessions can be set up and torn down is unaffected by packet filtering, but approximately halved by a proxy. The bulk data transfer throughput is a little worse than halved by the packet filter used, and cut to one-fifth by the proxy used. Thus, it appears that proxies to perform substantially worse than packet filters. However, the actual results also suggest that with reasonably available fast hardware, throughputs quite close to wire rate, at least for the tests conducted, would be perfectly possible with either approach.

1. Firewall Performance Measurement

First, some basic definitions. In order to get anywhere, we need a notion of a *transaction.* This is just a single atomic "thing to do", it might be to set up a TCP connection, move one byte of data, authenticate a login, or FTP a single file from host A to host B. *Throughput* is the maximum number of transactions a system can process in a unit time. Things like 'bytes per second' or 'transactions per hour' are statements of throughput. *Latency* is the time required by a system to complete a single transaction, from start to finish.

From these definitions, we develop a couple more useful ideas. *Service time* is the reciprocal of throughput. Think of it as how much actual processing time it takes the system to handle a transaction. The part of the latency that's not the service time is the *queuing time.* So queuing time is, roughly, the amount of time a transaction spends waiting on the wire or in a queue.

Most networking transactions are latency insensitive, within reason. A few milliseconds here and there is fine. What generally matters is throughput. However, for some networking protocols (such as LAT or IPX) latency is most certainly an issue.

2. Types of Transactions

We'll be looking at a couple of different notions of 'transaction.'

- FTP session setup + GET + teardown
- FTP get bulk data, 'transaction' is really the fetch of a single byte

- send a UDP echo request, receive a response

These tests are described in greater detail below. For each, we'll try to understand both the latency and the throughput.

3. Types of Firewalls

There are several firewall types out there to compare:

- Routers/Bridges (packet filtering)
- Stateful packet filters
- Application proxies , with various styles of authentication.

For this paper, a packet filtering bridge and the ftp-gw from the TIS firewall toolkit were compared. Stateful packet filters and 'various styles of authentication' are left to other experimenters.

4. Problems

There are various problems inherent in testing firewalls for performance. The testing described herein has tried to account for many of them, in order to provide a useful comparison, but in doing so has diverged from the real world substantially.

Platform

Many firewalls will run on a variety of different hardware, some slow, some fast. Inevitably, comparisons between real-world configuration of product A and real-world configuration of product B will be rendered useless because either A or B is using hardware that's 6 months ahead of the other. The solution for these experiments was to use identical, and very slow, hardware. This slow hardware has the additional effect of bringing performance into a range where differences can be seen (if everything does wire rate, it's hard to compare).

Noise

It is often hard to eliminate stray noise in timing. This can be caused by other traffic on the networks, various and sundry service protocol requests and so on. The networks used in this series of experiments were constructed entirely of crossover-wired ethernets connected point to point, so the networks involved were noise-free. In order to reduce the extraneous service request traffic, all hosts were named only by IP addresses, and all users were to be found in the password file. In fact, no nameservice or NIS was possible, since there were no nameservers of NIS servers available.

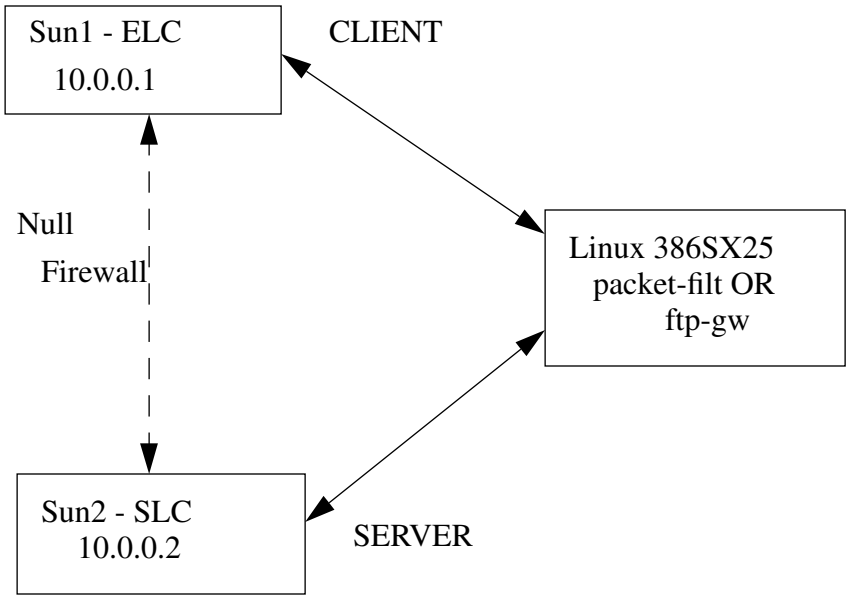
5. Measurements

In general, one wants to measure both latency and throughput.

Latency can be measured relatively simply, by doing a bunch of transactions in a row, single-threaded and synchronously (start the next one when the previous one completes). Divide the total transaction count by the elapsed time.

Throughput can be measured by running many threads of transactions, as fast as possible, for some interval of time. Divide elapsed time by the total number of transactions to compute the Observed Service Time. Note that the real service time may be less than the observed one. It is necessary to try to get as many transactions as possible going simultaneously to try to permit all the queuing time for transactions to be used for service time on other transactions. In practice, there is a 'sweet spot' a certain number of parallel transactions that yields the highest throughput. For the purposes of this paper, a little tinkering was done to locate the sweet spot, and the observed throughput there was taken as the actual throughput figure.

6. The Lab



In order to establish baselines, the tests will also be run with the two suns directly connected. This is the Null Firewall. The underpowered Linux box is a) what was available and b) so much smaller than the Suns that it was easy to overpower it, and thereby get real numbers. That is, it will make the firewall the actual bottleneck, and avoid nasty problems with bottlenecks elsewhere obscuring the truth.

7. Predictions

FTP session transaction throughput for ftp-gw will be substantially less than the Null Firewall, but not measurably degraded for the packet filters. Latency through the packet filters will be slightly higher.

Byte throughput will be substantially the same for the packet filters and ftp-gw, each being substantially worse than the Null Firewall.

8. The Experiment

Where applicable, a UDP echo tool was used to get moderately high-precision round trip times for packets (the overhead of sending and receiving such a packet dominates the times, but the differences in round trip times for different configurations are interesting).

In addition, several tests using FTP transactions were carried out to further characterize performance. Many short sessions are run in sequence, to measure total session setup and teardown times (latency), and many short sessions are run in parallel to measure 'session level' throughput. Finally, several bulk transfer sessions (fetching a 1.7M file from the server to the client) are executed, both in sequence, and then several in parallel, so get the bulk data transfer characteristics (bytes/sec throughput).

To accomplish these goals, a tool called *workload* is used to synthesize the desired workload. See "Workload Performance Tool" on page 6 for details.

The use of FTP in scripts was greatly simplified by using a single line .netrc with the right things in it for the context at hand.

Ping tests

The round trip time of packets with 8 and 1000 bytes of UDP payload (respectively) were measured. In all cases the differences between the average time and the lowest time were quite small, probably well within any error bounds. These tests are conducted with a tool which sends a series of UDP packets to the echo port on the destination host, and displays results, not unlike standard ping. This tool exists to permit a higher resolution timestamp to be used.

FTP session tests

The following input files to *workload* were used:

```
sequential 30 {
        command "ftp-session"
}

parallel 6 {
        sequential 5 {
```

```
                command "ftp-session"
        }
}
```

which will run the script 30 times, in sequence, and start 6 parallel jobs each running the same script 5 times in sequence, respectively. The number 6 for parallelism was chosen by experiment as the number leading to the best throughput in the Null Firewall case (possibly related to the default length of listen queue of 5?). These two inputs measure, respectively, the total latency to set up a session, and the maximum possible throughput, in terms of sessions set up and torn down per unit time. In each case, 30 sessions are built and torn down. In the case of the ftp-gw based firewall, a parallelism of 8 yielded slightly better throughput.

The "ftp-session" script is a trivial shell script to log in to the server, fetch a single small file (the passwd file, in fact) and exit.

FTP bulk transfer tests

For this test, the *workload* tool was again used. The two input files were:

```
sequential 10 {
        command "ftp-bulk"
}

parallel 3 {
        sequential 3 {
                command "ftp-bulk"
        }
}
```

to do 10 bulk transfers (of vmunix to /dev/null) in a row, and then to do 3 parallel jobs, each doing 3 such transfers in a row. In the first case, 10 sessions occur, in the second, 9. The file transferred in approximately 1.7 megabytes in length.

9. Results and Analysis

Null Firewall Results

Ping (8 bytes)	3.1msec (average)
Ping (1000 bytes)	5.3ms (average)
Sequential FTP sessions	41.3sec
Parallel FTP sessions	13.2sec

Sequential FTP bulk transfer	29.8sec
Parallel FTP bulk transfer	20.6sec

It is worth noting at this time that the FTP session latency works out to about 1.38 seconds (41.3seconds / 30sessions), and the throughput of sessions is 2.3 sessions/second (30 sessions/13.2seconds).

Based on this, the sequential transfer in 10 sessions spent 13.8 seconds on session overhead, and 16 seconds on data transfer, while the parallel set of 9 sessions spends only 3 session overheads (due to 3x parallelism, more or less) for 16.46 seconds of transfer. The Null Firewall can push 17 megabytes of data in about 16 seconds, for just about wire rate, discounting the various session overheads.

Packet Filtering Firewall

See "Packet Filters Used" on page 9 for the actual text of the packet filters used and some information about the packet filtering software itself. Apart from verifying that the filters did in fact block the right packets and permit the right packets, the same tests were run.

Ping (8 bytes)	5.3msec (average)
Ping (1000 bytes)	11.8msec (average)
Sequential FTP sessions	41.3sec
Parallel FTP sessions	11.4sec
Sequential FTP bulk transfer	57.5sec
Parallel FTP bulk transfer	44.2sec

Several things are worthy of note here.

First, the packet filter did add per-packet latency, 1.1msec for the little packets, and 3.25msec for the bigger ones. The larger packets take approximately 1.7msec just to receive from the wire to the NIC and to transmit from the NIC onto the wire at 10Mbps (there's an additional hop with the firewall in place). The smaller packets transmit/receive overhead is very low, lost in the noise. Thus, the packet filter is taking 0.55msec longer to process a large packet, in the sense of service time. Note that we are assuming that the time spent actually moving a packet between the wire and the NIC is free, it is not charged to service time, since the CPU is presumably free while the NIC does this work.

Guessing, for the moment, that the 'per byte' costs of the smaller pings are negligible, we guess it takes essentially all of that 1.1msec to do the 'per packet' processing. The 0.55 milliseconds to process the additional 992 bytes of data in the large packets suggests a 'per byte' overhead of about half a microsecond per byte. In short, we're guessing that the service time for a packet is 1.1msec, plus 0.55 microseconds for

each byte in the packet. This analysis is all very well, but the resulting estimates are very sensitive to experimental error, minor changes in the input data change the results substantially. The FTP throughput tests show that, at least under load, the packet service time is closer to 2.5msec.

The FTP numbers are perhaps more useful. The session rates are virtually identical to the Null Firewall, TCP session setup and teardown seems to be completely dominated by the end-host overheads in this case. Indeed, the parallel version seems to have done slightly better, which is probably experimental noise, but might be due to the presence of an ethernet bridge (the firewall) improving some arcane ethernet phenomenon.

Finally the FTP bulk transfers, after the same computations as the Null Firewall case, shows about 40-44 seconds spent on data transfer. The parallel form appears a little better, but again this is most likely to be experimental noise. The actual throughput in bytes/second is, then, 17 megabytes in 42 seconds, or a little over 400 kilobytes/second. Since (by observing counters on the firewall) we observe that about 3 packets travel per kilobyte transferred (2 x 1500 bytes data and 1 ACK per 3000 bytes of data), we get a packet rate of about 424 packets/second.

This suggests that the packet service time is something like 2.4 milliseconds (1000/424). This is in stark contrast to our guesses about service times in the analysis of the UDP pings. However, if we guess that one of the ethernet transmit and receive times are not free, that is, that it must be included in the packet service time, the numbers start to look much more reasonable. A short perusal of the NE2000 drivers suggests that, at least sometimes, the CPU does indeed busywait while the card clocks an earlier frame onto the wire. This suggests a real service time for a 1500 byte packet of 1.1msec packet overhead, plus 1.2msec to clock the 12000 bits onto the wire at 10Mbps, for a total of 2.3msec.

The lesson to learn here is that you cannot easily extrapolate from little workloads to bigger ones. While the detailed analyses may have worked out, more or less, they still contain many guesses. The calculations may work out only because multiple gross errors have more or less cancelled one another out.

ftp-gw Firewall

The ftp-gw daemon was modified slightly for some minor performance tuning. Buffer sizes were made adjustable to get the best bulk data throughput possible. See “ftp-gw” on page 10 for the details.

Ping (8 bytes)	N/A
Ping (1000 bytes)	N/A
Sequential FTP sessions	77.8
Parallel FTP sessions	17.9(but see note)
Sequential FTP bulk transfer	112.8
Parallel FTP bulk transfer	88.5

NOTE: The parallel FTP sessions were slightly faster at 8 parallel sessions, each of 4 sequential jobs than at the 6 by 5 configuration used for the previous tests. This configuration yielded a throughput of 1.78 sessions per second. This is probably some artifact of the Linux TCP/IP stack, which gets exercised heavily here.

The FTP session latency here is 2.6 seconds, just about double the latency in the previous tests. This is not surprising, since ftp-gw essentially doubles the amount of session establishment work by mirroring everything out the other side. The FTP session throughput is 1.78 sessions per second, a little better than three-quarters of the two earlier results.

The FTP bulk throughput results are a little surprising. Even subtracting off 26 seconds of session overhead, the sequential version still spends 87 seconds transferring data, about the same time the parallel version does in total. Presumably there are details of the TCP implementations in question that make the parallel version just plain work better. Perhaps the kernel can receive many more packets between context switches? It is certainly nice to see performance improve under load, as opposed to degrade. The absolute byte rate seems to be about 200 kilobytes per second, less than half that of the packet filtering gateway, and 1/5 wire rate.

10. Conclusions

Neither packet filtering nor proxying does a great deal of harm to the ability to start up an FTP session through a the firewall. If the major load should happen to be this (perhaps moving lots of email about by FTPing each message in a separate session), the proxy will collapse first, given identical CPU power. However, the observed throughput of the ftp-gw configuration is substantially less than the Null Firewall, suggesting that 1.78 sessions per second is an upper bound. In contrast, the packet filtering firewall performed approximately as well as the Null Firewall, so may well be able to achieve better throughputs with a client and server (or several clients and servers) capable of generating a higher aggregate session rate.

Both packet filtering and proxying do interfere noticeably with bulk data throughput, the proxy substantially moreso. This last is somewhat surprising to the author, as he predicted that the two would perform roughly equally. Since both performed substantially worse than the Null Firewall, it is reasonable to suppose that the observed bulk transfer rates are fairly close to the maximum rates the firewalls are capable of.

Both approaches can, realistically, perform perfectly well in lower speed (common) environments. Even the little PC achieved something close to a full (half-duplex) T1's worth of bandwidth with the slower firewalling technology. With a sufficiently powerful machine, either approach could saturate an ethernet, at least in this simplified test configuration.

11. Future Directions

There are several fairly obvious tests to run in future. These are outlined, along with predicted outcomes.

• Multiple clients, multiple servers. The FTP session rates are fairly clearly dominated by host costs in the Null and Packet Filter case. In order to get a real understanding about just how much real work a firewall can do in this dimension, multiple hosts are needed. The proxy firewall will probably not perform much better, since it is an end-host with all the end-host costs, but the other scenarios should show marked improvement.

• Bidirectional tests. All the bulk traffic was unidirectional in this experiment, which may affect the performance in some way. This author would not care to venture a prediction here.

• ttcp tests. This is a performance analysis tool that allows a variety of TCP parameters to be adjusted, and provides detailed output. This would be valuable for higher-resolution information that the FTP bulk transfer tests herein can only approximate. Probably the results will be similar to the FTP bulk transfers, but verifying this would be valuable.

• Other protocols. The FTP session rates are not tremendously interesting in the real world, but a similar test for http probably would be of great interest. Probably the results would be similar to those for FTP but, again, verification would be valuable.

• Tests with synthetic workloads derived from real workloads. Work is proceeding on tools to generate specifications for the *workload* tool from actual log files. Such specifications can then be used to drive various firewall configurations, and can be easily modified to simulate statistically similar but higher or lower absolute loads.

12.Acknowledgments

Many thanks to Marcus Ranum and Jody Patilla for reviewing preliminary versions of this paper and providing valuable feedback, and to Marcus for the firewall toolkit.

Appendix A Workload Performance Tool

The workload generator tool creates a synthetic workload of user commands to be executed. The workload might be ‘do this 1000 times in a row’ or ‘do 100 of these all at the same time’ or more complex constructions. These constructions are specified in a formal input language, defined as follows.

The version of this tool used for these tests can be found in the archives of the firewalls-performance mailing list, on ftp.greatcircle.com, in /pub/firewalls-performance/digest/v01.n011.Z.

The language read by workload allows for comments, any text following an octothorpe (# character) on a line is discarded. The remainder is parsed, ignoring newlines and other whitespace (except as needed to separate keywords from one another), into a job description.

The notion of a job is defined recursively. The simplest job is:

command “<string>”

The string is split into tokens, at white space, and the resulting vector is used as arguments to execvp(3). No filename globbing or metacharacter handling is done. There are also complex job specifiers for doing a set of jobs in sequence, in parallel, or in a (deterministic) pseudo random order.

A set of jobs can be carried out in sequence by using the *sequential* keyword. The first form carries out each of the specified jobs, in the order specified. Each job must conclude before the next one is started. The second form does the same thing a specified number of times, as specified by the repeat count.

sequential {
<1 or more job specifiers>
}

sequential <repeat count> {
<1 or more job specifiers>
}

The *parallel* keyword has exactly the same syntax as the sequential keyword, but the semantics are different. Rather than waiting for each job to conclude, all are started in the same order as with the sequential form, but as fast as possible. The parallel construct then waits for all of the jobs to complete before it is itself complete.

The last job type is random. This has only one form:

random <repeat count> <seed> <low> <high> {
<1 or more job specifiers>
}

This is intermediate between the sequential and the parallel constructs. The repeat count is not optional, and there are three additional numeric arguments which determine the pseudo-random intervals between job starts. The jobs are started as in the two other constructs, except that the interval between job starts is determined by successive calls to random(3). The output of random(3) is used to select a uniformly distributed number between <low> and <high> to start the next job in sequence. The seed value is used to as the argument to srandom(3), to provide repeatable results.

When all the jobs have been started, workload waits for all to complete before the job is complete.

Some examples:

sequential 1 { command “ls -l” }

will cause the command ‘ls -l’ to be executed once.

parallel 5 { command “ls -l” }

will launch 5 copies of ‘ls -l’ at once, and wait for them all to terminate.

sequential 3 { parallel 5 { command “ls -l” } }

will run 5 copies of 'ls -l' in parallel, wait for them to all terminate, and then do it again 2 more times. Finally:

```
sequential 3 {
        random 5 42 1 3 {
                command "test-script"
        }
}
```

Will start 5 copies of 'test-script' at pseudo-random intervals between 1 and 3 seconds (srandom() used to seed the standard library random number generator with 42), and then wait for them all to terminate. Then it'll do it twice more, for a total of 3 sequential instances of the pseudo-random job.

Naturally, job specifications can be nested more deeply (indeed, as deeply as you like).

Appendix B ftp-gw

This is the standard ftp-gw from the TIS firewall toolkit (version 1.3). Early results were slow enough to prompt modifying the server to accept a 'shovel' parameter in the configuration file, which specified how large of a block size to use in reading and writing on data connections. This made it easy to find the 'sweet spot' in the space of possible buffer sizes. Not surprisingly, the bigger the buffer, the better the throughput. At a buffer size of about 16K, the performance seemed to be very close to maximum, so the tests were run at this setting. The overall performance improvement was (for large bulk transfers) from about 160Kbytes/sec with the default 2K buffer, to about 200Kbytes/sec with the 16K buffer.

At the suggestion of the toolkit author, the peername() function was also modified. This function normally attempts several nameservice lookups, looking up the name belonging to the address and the address belonging to that name, cross-checking the DNS databases. This is somewhat standard practice, and somewhat expensive. The resulting speedups in session establishment were measurable, but not dramatic. Do recall, however, that the lab network had no nameservice, so while peername() may have been attempting to do nameservice, it was not succeeding, which probably sped these checks up (though rendering them somewhat pointless).

A further suggestion to modify the receive and send buffer sizes of the TCP sockets used by ftp-gw was not taken, as kernel sources suggested that these default to the largest possible sizes.

There is not much else to say here, except that it is wonderful and that it built seamlessly, right out of the box. The design of the configuration system made the modifications described essentially trivial. This author was highly impressed by the entire firewall toolkit.

The entire firewall toolkit is available via anonymous FTP from ftp.tis.com, in /pub/firewalls/toolkit.

The hardware and operating system used to run the server is described in the next section.

Appendix C Packet Filters Used

The packet filtering software is an in-kernel bridge, built into Linux 1.1.95. It uses a fairly sophisticated compiler to produce native code to do the packet filtering, and inserts that into a simple ethernet bridge module. The input filtering language is very similar to Network Systems' Packet Control Facility, though the output is substantially different. The packet filtering software used in these tests is unavailable outside of Network Systems at this time.

The hardware is a 386SX25 PC, equipped with 2 NE2000 compatible ethernet cards. These cards do not do DMA, so the machine spends a fair bit of its time copying packets in and out of the device buffers. Better cards would probably help out, to some extent. Other experiments have suggested strongly that this device is limited by the device driver overhead, and general kernel overhead, the packet filtering is essentially free.

The actual packet filters used are as follows. They provide a pretty fair set of rules, this author believes.

```
# Apply on 10.0.0.2's interface, denies any iffy-looking packets.
filter nospoof_one
        ip_sa in (10.0.0.1) fail;
        ipfrag in (1..8) fail;
end

# Apply on 10.0.0.1's interface, same thing.
filter nospoof_two
        ip_sa in (10.0.0.2) fail;
        ipfrag in (1..8) fail;
end

# Log any TCP packets with a SYN, FIN or RST bit set, so we can count sessions.
# apply on 'first' filter point.
filter audit
        tcpflags/0x02 = 0x02 log 10.0.0.1;
        tcpflags/0x01 = 0x01 log 10.0.0.1;
        tcpflags/0x04 = 0x04 log 10.0.0.1;
end

# Client To Server filtering rules. The logic is convoluted, but it does work.
filter c_to_s
        # Permit UDP echo requests
        ip_protocol in (17) udp_dp in (7) succeed;

        # If it's not originating on a 'safe' user port, dump
        tcp_sp in (0..1023, 6000..6010) fail;

        # If it's not going to ports 20 or 21, dump it
        tcp_dp in (0..19,22..65535) fail;

        # if it's trying to do connection setup to data port, dump
```

```
        tcpflags/0x12 = 0x02 tcp_dp in (20) fail;

        # if it's trying to do connect response to control port, dump
        tcpflags/0x12 = 0x12 tcp_dp in (21) fail;

        # let it go on. Some other filter point might choose to heave it, but we're ok.
end

# Server To Client filtering rules
filter s_to_c
        # Permit UDP echo replies
        ip_protocol in (17) udp_sp in (7) succeed;

        # if it's not going to a 'safe' user port, dump
        tcp_dp in (0..1023, 6000..6010) fail;

        # If it's not coming from ports 20 or 21, dump it
        tcp_sp in (0..19,22..65535) fail;

        # if it's trying to do connection setup from control port, dump
        tcpflags/0x12 = 0x02 tcp_sp in (21) fail;

        # if it's trying to do connect response from data port, dump
        tcpflags/0x12 = 0x12 tcp_sp in (20) fail;

        # let it go on. Some other filter point might choose to heave it, but we're ok.
end

#
# Now apply the rules on the various right places.
#
apply audit on first;
apply nospoof_one on eth1 incoming;
apply nospoof_two on eth2 incoming;

apply s_to_c on 10.0.0.2/255.255.255.255 to 10.0.0.1/255.255.255.255;
apply c_to_s on 10.0.0.1/255.255.255.255 to 10.0.0.2/255.255.255.255;
```