Inferno Nettverk A/S

Performance analysis of Dante version 1.4.0

Inferno Technical Report #4

May 17, 2014

Contents

List of Figures

1 Introduction

This report documents the results of a performance analysis performed on version 1.4.0 of the Dante SOCKS server implementation from Inferno Nettverk A/S. The testing was done slightly prior to the release of Dante 1.4.0, with the code being mostly identical to the 1.4.0 release code.

The load analysis was done on a machine running in production at a company using Dante on several machines. The analysis was done on one of these machines over a period of roughly two weeks, with up to $25{,}000$ simultaneous active clients being handled during peak times.

This report first describes the basic testing methodology and test environment, and then provides a general overview of the collected data to provide a context for understanding the load the system was placed under, the amount of data transmitted, the number of clients, etc. The report additionally looks at some interesting data points in more detail, such as the ability of Dante to adapt to abrupt changes in network traffic and the resource requirements of different aspects of Dante. Finally, the report discusses challenges related to doing this type of performance evaluation and considers some possible improvements for future versions of Dante.

2 Executive summary

Dante is a product developed by Inferno Nettverk A/S. It consists of a SOCKS server and a SOCKS client, implementing RFC 1928 and related standards. It is a flexible and scalable product that can be used to provide convenient and secure network connectivity.

The performance of the Dante server running on a highly loaded machine with 16 2.40 GHz *Intel Xeon* CPU cores and 32 GB of RAM was measured over two weeks and the following behavior was observed:

- Sustained send/receive rates of around 0.8 Gigabit/s in each direction and combined rates of around 1.5 Gigabit/s (see Figure 1).

  IP-packets are for most of the two week period received and transmitted at a steady combined rate of around $250{,}000$ packets per second, with around $150{,}000$ of these being TCP segments, and around $100{,}000$ being UDP packets. Roughly the same amount of packets are sent and received.

- Between $15{,}000$ and $25{,}000$ concurrent users were handled without any identified performance problems (see Figure 2).

- Good ability to quickly adapt to sudden load changes (see Figure 3, where on day 10 there was likely an external network outage temporarily disconnecting all clients), by first quickly decreasing the amount of Dante processes as the number of concurrent users falls, and then handling the sudden burst of new users by adding new processes when the traffic returns.

- The machine has enough memory, and also enough CPU capacity most of the time.

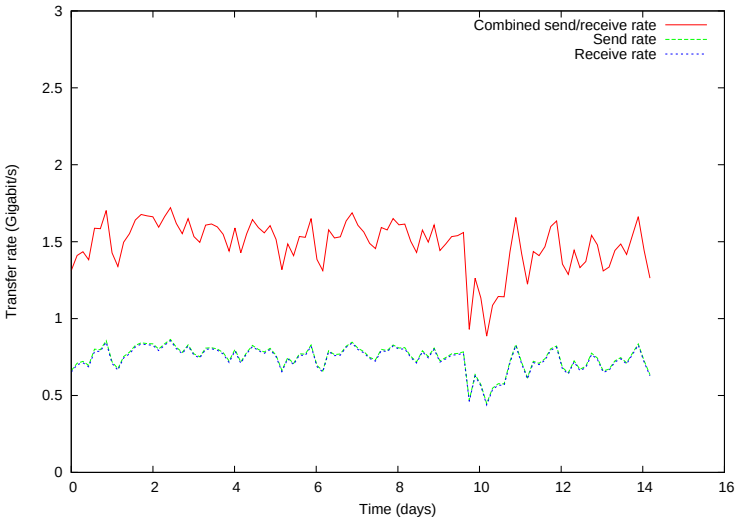

Figure 1: Transfer rates

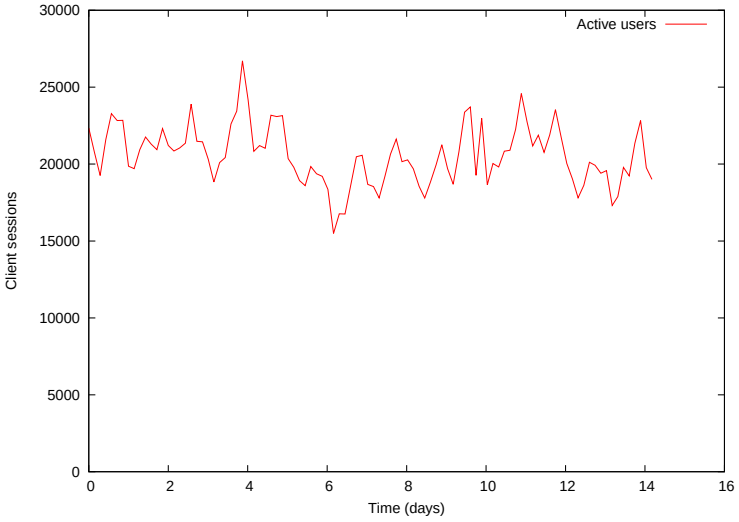

Figure 2: Active users

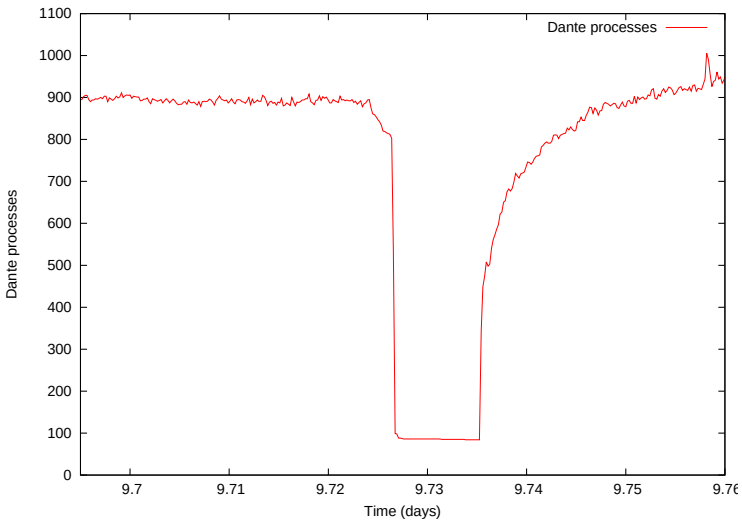

Figure 3: Traffic loss Dante processes

3 Production environment

The production environment consists of the machine the Dante server was running on and the Dante build used on the machine.

3.1 Machine description

The production machine has a 16-core 2.40 GHz *Intel Xeon* CPU and 32 GB of RAM, making it a fairly powerful machine with a high amount of memory by todays standards. It should be well suited for running Dante with a high load.

The machine runs a 64-bit Linux distribution and uses kernel version 3.2.0-4. Both incoming and outgoing traffic passes over the same interface (*eth0*).

3.2 Dante configuration

A non-released snapshot of the Dante source code was used on this machine, with the source code essentially being identical to the subsequent official 1.4.0 Dante release.

Some compile time adjustments were made to the Dante server, using the following configure option:

–without-gssapi GSSAPI support was not needed on this machine and using this option disables the GSSAPI support and reduces the amount of memory required by Dante.

The Dante *-N 4* runtime option was additionally used to improve performance on this highly loaded machine, leading to four Dante *mother* processes being used to receive client requests. The *-n* option was also used to disable TCP keep-alive messages, due to the preference of the site operator.

Only errors were logged by the Dante server, meaning very little was written to the log files during normal usage.

4 Measurement methodology

The performance analysis was performed in a production environment, and as such, it was desirable to monitor the server without any additional server logging or configuration changes, but this made it necessary to limit data collection primarily to information obtainable via system tools such as *ifconfig* and *vmstat*, in addition to the information Dante provides via its *setproctitle()* calls, which includes an overview of the current number of active clients.

This monitoring approach places relatively little extra load on the system, but the values obtained are not always entirely exact. For example, having the Dante server log data traffic would allow exact values for transmitted data and protocols to be obtained, but at the cost of increased system load due to the overhead from logging. Obtaining these values instead from *ifconfig* means that traffic not going to or from the Dante server might inflate the number of transmitted bytes, so this approach is only usable if the vast majority of traffic on the interface is to or from the Dante server. For this machine, running the Dante server was the main purpose, which should make the data obtained via *ifconfig* be fairly accurate, if somewhat limited in detail. The bandwidth measurements that are listed in the results with a ”per second” rate are based on averaging the bandwidth measured by sampling done every 15 seconds, which should be unproblematic due to the measurements spanning more than two weeks.

The CPU usage information obtained via *top* and *ps* is likely to be slightly lower than the actual load on the machine; because only CPU usage for each running process since the last sample point is available. This means that the CPU time spent by processes that terminate will only be measured up to the last sampling point before the process exits. Likewise, processes with a lifetime of less than the sampling interval might not be observed at all, if they are created and exit between two sampling points. Most CPU intensive processes were however long-running on this machine, so this should not have significant consequences for the measured results, except for probably giving slightly lower CPU usage values and some information lost during spikes of activity, as is discussed later.

Data collection was done via a script that ran measurement programs roughly every 15 seconds and stored the output to disk. The resulting text files were later analyzed and interpreted, with the results presented in this report.

There are some practical consequences resulting from the analysis methodology used:

1. The collected data is not obtained in an atomic manner.

2. Data collection is affected by (and to a lesser degree affects) the system load on the machine.

The first point is primarily relevant when data from different system commands are compared. Data is collected at 15 second intervals and grouped based on the 15 second interval it was obtained in. With some commands executed in parallel and some sequentially, there will always be some variability concerning when in the 15 second interval a command was executed. The difference in time should however not exceed 15 seconds (not counting command execution time). There potentially being a difference of up to 15 seconds between values merged into the same data point should be considered when data from different system commands are analyzed together, but with the full data set spanning two weeks, a difference of a few seconds should not have a large impact on the results overall.

One merged data set with data combined from all system commands was created with complete information from the different commands for most data points; the number of data points available for each command varies from 98.62% and 99.24%. Some data points missing data from some commands is partly a result of a limitation in the collection procedure. A fixed delay of 15 seconds in the data collection script resulted in a certain amount of drift between each data point due to time also being spent on command execution in addition to the fixed delay. This has been compensated for in the data analysis, but there are also changes in the time between data points caused by high system load.

When the machine is under very high load, scheduling might result in more than 15 seconds passing between each time data is collected. While this means that less precise information will be available when the system is under high load, the variability in time between each data point also provides an additional indication of when the system has been so highly loaded that normal process scheduling has been significantly affected, giving an indication of points in the data set that are potentially interesting to look at in more detail.

Figure 4 shows the time series of data with the difference in time between each snapshot interval shown. For the majority of the values, the interval values are between 15 and 16 seconds, but there are some higher spikes, especially during the third day,

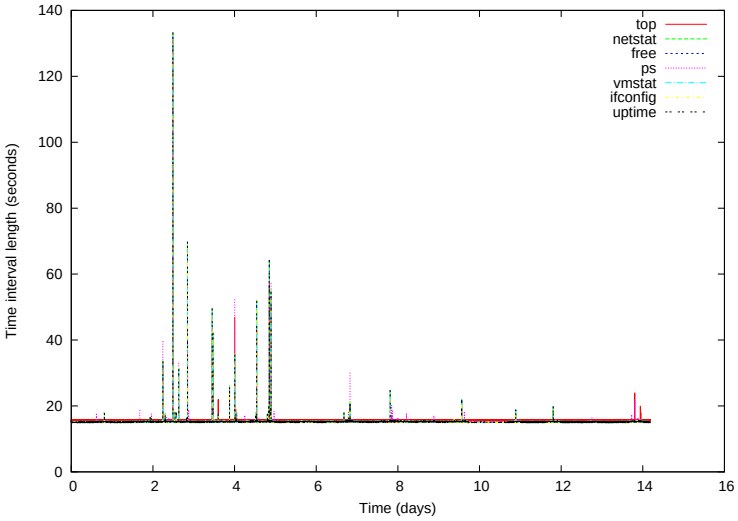

Figure 4: Snapshot interval lengths

with the highest being 133.36 seconds long, almost nine times longer than the normal delay of 15 seconds. This high point is examined in more detail in Section 10.

The upper part of the distribution of the interval lengths can be seen in Figure 5, showing the cumulative distribution function for the snapshot length values. While most of the values do not exceed 20 seconds, there is a tail with around 0.02% of the values being over 20 seconds.

It seems natural to conclude that the high interval values are caused by scheduling problems due to the machine being highly loaded at these times.

For the actual data analysis, the variability in interval times meant that it was necessary to be careful when looking at the difference between data points. The magnitude of changes in values between data points might be misleading unless the time elapsed is also considered, meaning that the rate of change needs to be examined in many cases.

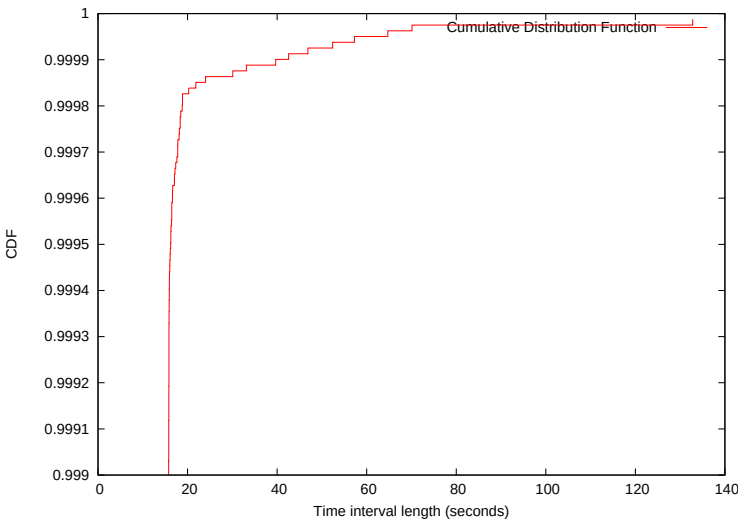

Figure 5: Snapshot interval CDF

5 Analysis methodology and limitation

Along with a general overview of the behavior observed on the machine, we also look at some specific events in more detail and for each event there are two main factors that are considered:

- Any observable changes in the collected data.
- The event or events that resulted in the changes.

Both are interesting for multiple reasons. In cases where unwanted behavior is observed, it is desirable to know the underlying reason or set of events that led to the behavior in order to be able to make adjustments to the behavior in Dante, should there be a better way to respond when similar events occur. In cases where the observed behavior not necessarily is undesired, knowing what it is might still help when trying to analyse the data changes that are observed.

Underlying events can however only be implied from the observed behavior, which means that the quality of the collected data affects how simple it is to attempt to deduce what the underlying event was. When an event results in, for example, high load, the quality of the collected data might be reduced as a result of the collection processes not being able to run as frequently as when the system load is lower.

These limitation make it possible that what occurred on the machine at any given time might be misinterpreted or that data points that would be necessary to determine what happened might be missing, but there should still be sufficient information available to be able to identify cases where data might be missing. Quite a lot of different information is also collected, which provides multiple ways to analyze any given event.

6 Performance and load analysis

We first look at the overall load on the machine.

6.1 System load

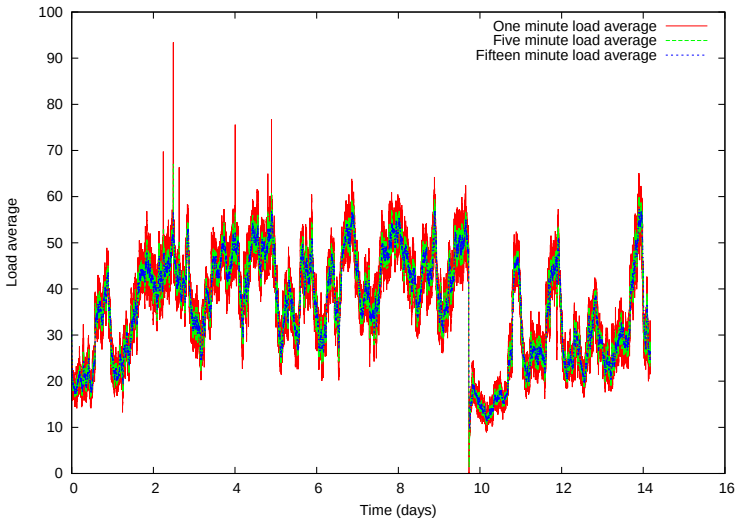

Figure 6: System load average

Figure 6 shows the load average on the machine for the two week period, as reported by *uptime*. Without attempting to interpret the absolute numerical values reported, the load is non-zero for most of the time, and has some somewhat regular changes that are possibly related to the time of day, without the patterns clearly pointing towards any specific timezone.

There is also one spike down at the end of the 10th day, with the load quickly going down towards zero, before gradually increasing again, but to a somewhat lower level than before the spike down. This spike is examined in more detail in Section 8.

6.2 Network usage

Figure 7 shows the transfer rate of data passing through the interfaces on the machine, with traffic for all interfaces (*eth0* and *lo*) summarized. As both traffic between clients and the Dante server, and traffic between Dante and the target servers pass over the same interface, and no information about transmitted data is logged by Dante, it is not possible to provide a more detailed breakdown of how much data is transmitted via Dante, but almost all of the data will have passed via the Dante server, as there is known to be only negligible amounts of non-Dante traffic on the machine most of the time.

The machine receives and sends data at a fairly stable rate of around $0.7-0.8$ Gigabit/s in each direction, for a total combined rate of around 1.5 Gigabit/s passing through

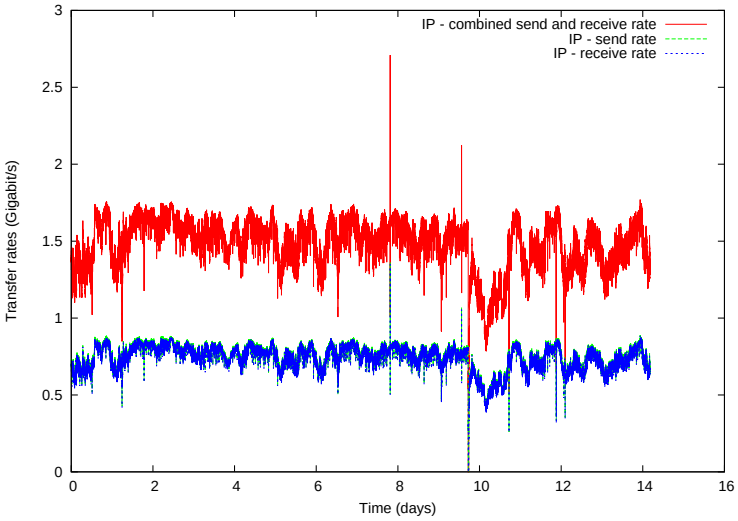

Figure 7: Bandwidth usage

Dante. A spike down towards a rate of zero can be seen also in this plot at the end of day 10. The transfer rate gradually recovers, but with a slightly different traffic pattern.

One point to note with the configuration on this machine is that the use of a single physical NIC[1] for both the internal and external Dante interfaces likely limits the achievable traffic rate. Rather than having the client side on one physical network interface card and the target server side on another card, on this machine both logical interfaces share a single physical NIC. Even so, we measure transfer rates of up to 0.9 Gigabit/s in each direction of the full-duplex network link, so the machine and Dante does manage to push traffic up to near the saturation point for the 1 Gigabit/s link.

The division of the traffic between different protocol types is not available on the byte level via the system tools used to collect data, but *netstat* provides information about package counts for different protocols, which is shown in Figure 8, for TCP, UDP and ICMP, along with the total number of IP-packets.

On a per-packet/segment basis, TCP traffic appears to represents the majority of the traffic, but there is also a significant amount of UDP traffic. There is interestingly enough a gap between the number of received and transmitted TCP segments, with a higher number of received segments. The reason for the difference between the two values is unknown, but might be related to the SOCKS protocol processing, as a SOCKS request is performed between the SOCKS clients and the Dante server, or it might be related to how the kernel on the machine Dante is running on handles traffic before it is transmitted. It would however likely be prudent to see if similar behavior is observable in a more controlled environment, so examining this in more detail might be interesting in future experiments.

IP-packets are for most of the two week period received and transmitted at a com-

[1] Network Interface Card

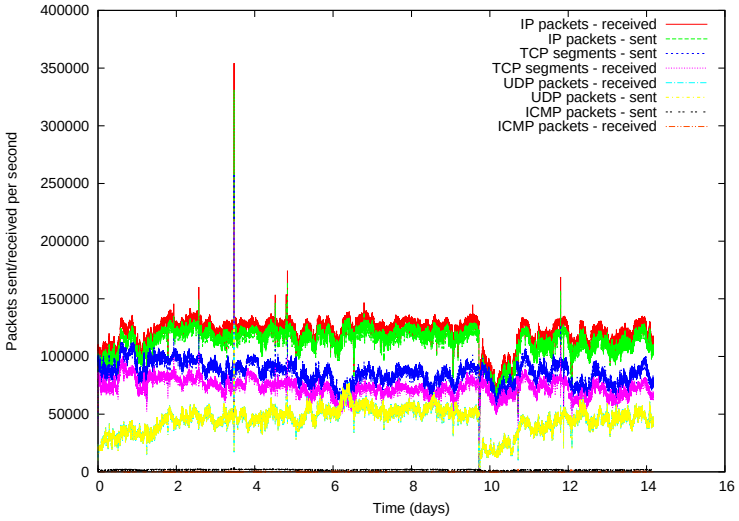

Figure 8: Per-protocol packet transmission overview

bined rate of around $250{,}000$ packets per second, with around $150{,}000$ of these being TCP segments. TCP traffic is fairly stable during the whole period, while the UDP traffic gradually increases from $50{,}000$ per second to $100{,}000$, before dropping towards zero on the 10th day. After this, the UDP traffic starts gradually increasing again.

ICMP traffic is very modest compared to UDP and TCP, with around 130 packets received per second and 1700 packets per second transmitted.

The change in load at the end of the 10th day appears to primarily have a lasting affect on the UDP traffic, with TCP traffic quickly resuming the previous rate.

6.3 CPU usage

Figure 9 shows the CPU usage for the period, based on data from *top*. The Dante value is obtained by summarizing the CPU usage of each Dante process, and the non-Dante value by summarizing the CPU usage of each non-Dante processes.

As can be expected, Dante processes account for most of the CPU usage, totalling around $80 - 90\%$. Idle CPU time lies at around 10%. Non-Dante processes have very little CPU consumption, most of the time less than $3 - 4\%$. Slightly less than 40% of the CPU time is spent in the kernel, with soft interrupts consuming around 5% CPU time. Slightly more than 40% of the CPU time is spent in user space. As the purpose of the Dante server is to transmit data, this is largely as expected; data is transmitted from the network, to kernel memory, to application memory, and then back again in the reverse order. All this data movement relies on both the application and the kernel and CPU time is split fairly evenly between the two.

The spike down at the end of the 10th day can be seen also in the CPU usage, giving a reduction in the CPU usage of Dante, and a corresponding upwards spike in the idle CPU time. The load appears to be slightly lower after the spike down. The

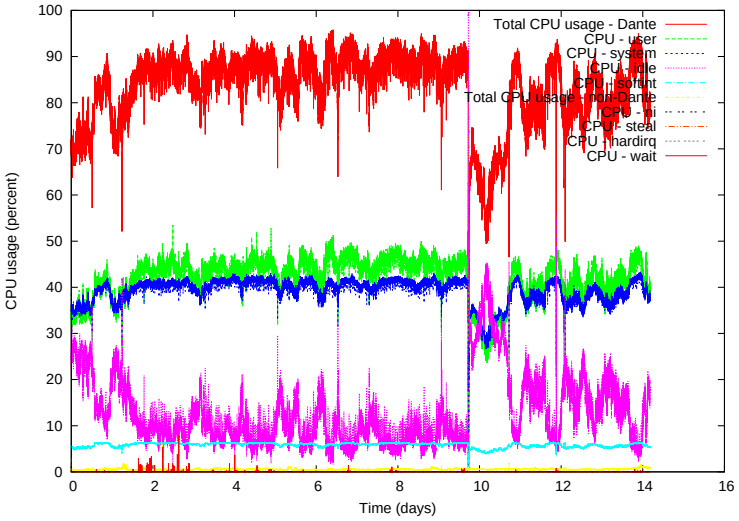

Figure 9: CPU usage

spike down in CPU usage corresponds to the similar spike down in network traffic, seen in Figure 7 and Figure 8, as would be expected if the network traffic was suddenly cut off or significantly reduced.

Apart from the spike down, the load is fairly constant. The CPU idle time generally being at around 10% might indicate that the performance of Dante on this machine is not limited by the CPU.

To look at the CPU usage in Dante in a bit more detail, Figure 10 shows the highest CPU usage reported for a process of a given type at any point, and gives an indication of which Dante activities are responsible for most CPU consumption. The CPU usage values have been calculated based on the CPU time output reported by *ps*. The data has then been smoothened (using the *gnuplot smooth bezier* option) to make it easier to interpret.

The Dante *io* processes, responsible for reading and writing data after SOCKS protocol processing has completed, has both the highest and most variable CPU usage, with CPU usage values of around $30 - 40\%$. The high CPU usage of the *io* processes corresponds well with the high amount of system CPU usage seen in Figure 9; sending and receiving data is the most resource intensive operation of Dante, as would be expected because the main task of the Dante server is to relay data and a high amount of data passes through the machine.

The Dante *mother* and *negotiate* processes have fairly constant and similar CPU usage, at around 15%.

Very low CPU usage is seen for the *request* process type, with usage at around $1 - 2\%$. The *monitor* process is not in use and has no CPU consumption.

While the data in Figure 10 shows the smoothened data from *top*, the raw data can be seen in Figure 11. This plot essentially shows that there are some high spikes in the data, as reported by *top*. There is especially one spike after day 2, where very high

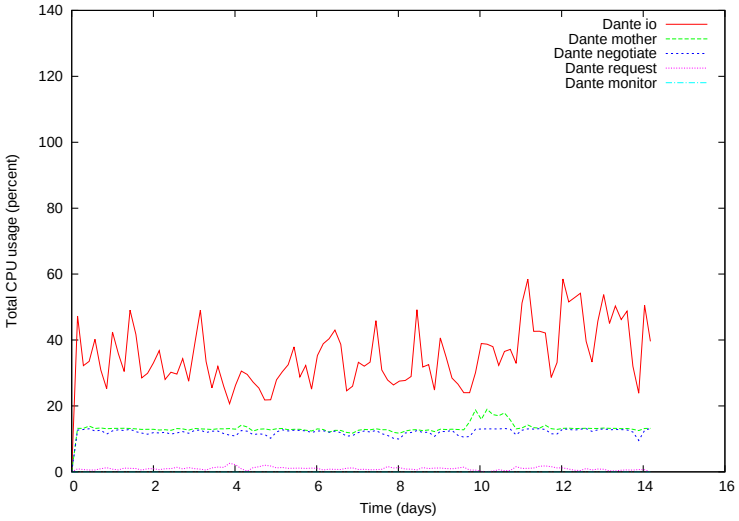

Figure 10: Dante per-process type highest CPU usage (smoothed)

CPU usage is reported. Some of the values reported are above 100%, which is clearly impossible, but as discussed in Section 4, the data is not obtained atomically, which can result in inaccurate data when the machine is highly loaded.

The high CPU usage spike appears to affect multiple Dante process types and the events at this data point is examined in more detail in Section 10.

6.4 Processes

An overview of the processes running on the machine can be seen in Figure 12, with the total number of processes varying for most of the time between 1200 and 1400, with some spikes up towards 1600.

The majority of processes are from Dante, which has between 800 and 1000 processes for most of the time. Non-Dante processes number between 400 and 600, and there appears to be some degree of correlation between both Dante and non-Dante processes, with many of the same changes occurring for both.

A significant spike down to only 84 Dante processes can be seen at the end of day 10 also in this figure, as it would appear that the reduction in network usage corresponds with a reduction in the number of Dante processes.

The composition of the Dante processes is shown in Figure 13, with the majority being *io* processes, varying between 600 and 800 in number. The *io* process count is fairly variable, presumably adapting to load changes as the number of active client sessions increases or decreases.

The number of *request* processes is relatively stable at between 100 and 150, with some spikes higher. The *negotiate* processes also show some spikes but are also fairly stable, with around 10 to 15 processes.

The number of *mother* and *monitor* processes is always constant, at 4 and 1, re-

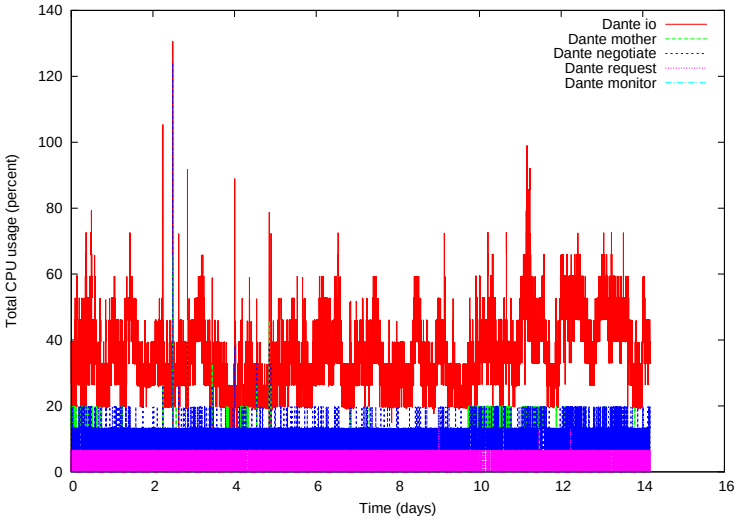

Figure 11: Dante per-process type highest CPU usage

spectively[2].

A spike down at the end of day 10 can be seen also in this plot, and clearly affects the number of *io* processes, which quickly falls towards zero and then recovers. This event will as noted above be examined later, in Section 8.

6.5 Memory usage

The overall memory usage on the machine is shown in Figure 14, and there are some interesting changes that can be seen. The machine has a fairly large amount of memory (32 GB) and even with a large number of processes, there is hardly any swap usage (only around 22 KB most of the time). The changes that occur happen in the amount of data stored in main memory and there are two primary sets of behaviors that can be seen.

The first is a variability similar to the load-dependent changes seen in the network traffic and CPU load plots. The used, active and inactive memory values all show this type of variability. For the active memory, the overall value is fairly stable at around 10 GB. The used and free memory values, on the other hand, show a corresponding overall trend either upwards (used memory), or downwards (free memory). The amount of used memory gradually increases from around 10 GB to 25 GB, while the amount of free memory decreases from around 20 GB to 5 GB. The likely reason for this overall trend can be seen in the *swap cache* and *inactive memory* values, that gradually increase from around 0 GB to almost 20 GB for the swap cache and above 10 GB for the inactive memory. A notable exception to the trend occurs at the beginning of the

[2]Some additional mother processes run at certain times due to a script running on the machine that attempts to restart the Dante server in case it is no longer running, but these processes quickly exit and never receive any client requests since Dante was never terminated during this measurement period.

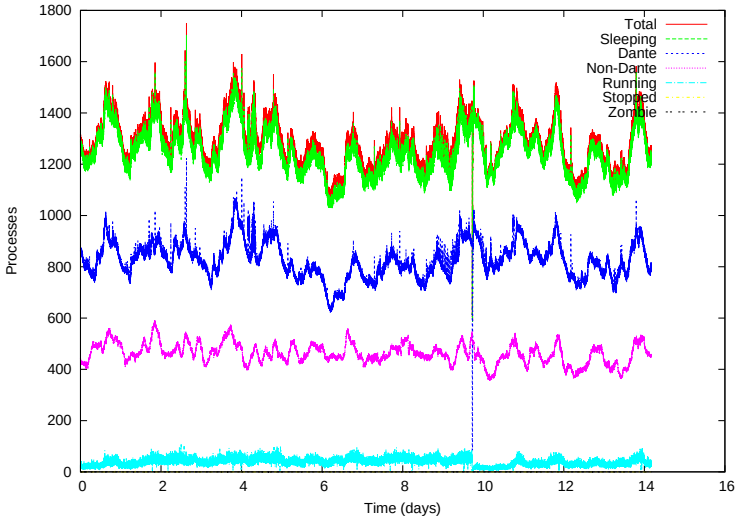

Figure 12: All processes

third day, when the amount of inactive memory suddenly falls to zero, and this point is examined in more detail later, in Section 7.

As with the other plots, a downwards spike occurs at the end of the 10th day, and is likely caused by the reduction in Dante processes at this point, as seen in Figure 13. That the swap cache and inactive memory values appear unaffected by this indicates that these values are not directly related to the Dante processes.

6.6 Clients and connections

The Dante *io*, *negotiate* and *request* processes call *setproctitle()* when necessary to update the process title with information on the current number of active client slots, making it obtainable via *ps*. This client data is shown in Figure 15, along with the *connections established* value reported by *netstat*, which shows the current number of established TCP connections on the machine.

All values show what are likely load-related changes, but are still fairly stable, with gradual increases and decreases. The number of active *io* process clients varies between around $17{,}500$ and $25{,}000$, with $10{,}000$ in the *in progress* state. Around 475 clients are in the negotiate state and 30 in the request state.

The spike towards zero at the end of day 10 can be seen also in this plot, and is followed by a spike up in the number of clients in the *in progress* state, indicating that many new client connections were received in a short time.

After the spike down there is also a small reduction in the gap between the number of *io* clients and the total number of established TCP connections. Any active TCP SOCKS session consists of two TCP connections (one between Dante and the SOCKS client, and one between Dante and the target server), meaning that if all clients shown in the plot represented TCP sessions, the total number of established connections should

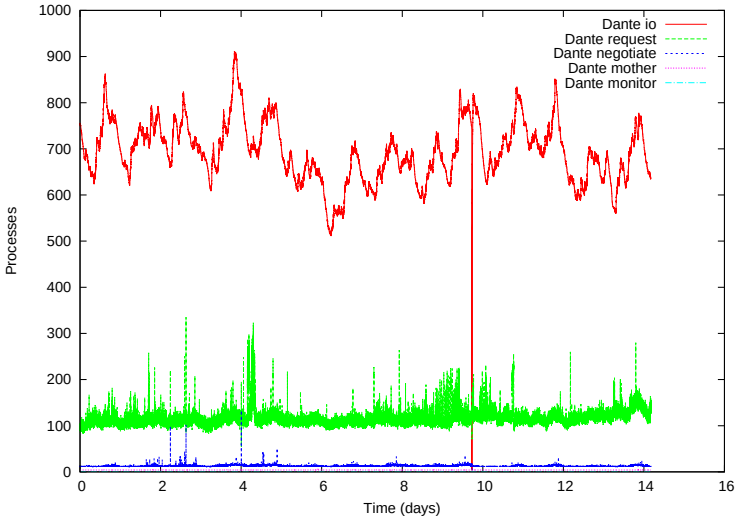

Figure 13: Dante processes

be at least twice the number of active *io* clients, which is clearly not the case here. Part of the reason for this will be due to some clients using UDP instead of TCP, which needs a TCP connection between the SOCKS client and Dante, but not between Dante and the target server.

The extent to which the established connections values consist of UDP clients can be seen at data points just after the spike down. As seen in Figure 8, the UDP traffic takes a long time to recover after the spike down, while the TCP traffic quickly returns to almost the same rate. The reduction in difference between the *io* client values and the established connections values is likely related to this. While it cannot be said for certain whether the UDP clients reconnect or not after the spike down, or if they simply do not transmit as much data, it seems like there are fewer UDP clients, and that the remaining clients primarily are TCP clients. In which case, the total number of established connections is somewhat low.

Most likely this is caused by TCP connections that have not fully terminated, but are no longer in the ESTABLISHED state, or are not yet fully established, but we unfortunately have no information on the state of the TCP connections and cannot verify this. However, if this is the case, it would appear that a significant portion of the *io* processes handle sessions that are not currently active, but rather are in the process of being shut down or established.

6.7 Disk usage

Figure 16 shows the rates at which disk sectors are read and written from and to the disk on the machine. The average write rate varies between roughly 10 and 400 sectors/s, while the read rate is near zero, with the exception of a few spikes above 10 sectors/s. Assuming a sectors size of 512 bytes, this corresponds to an average write rate of up to

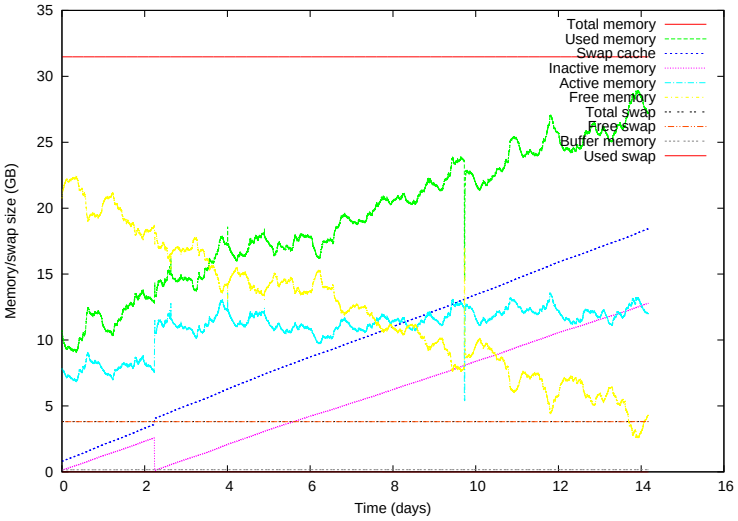

Figure 14: Memory usage

200 KB/s.

With this little disk activity, the handling of disk I/O will not have competed with network I/O to any significant degree, and the data collection script is likely the main source of data being written to the disk.

6.8 Summary

The examined machine is under a fairly high load, with $0.7 - 0.8$ Gigabit/s of traffic passing through the one network interface on the machine, in both directions. The majority of packets are from TCP traffic, but there is also a significant amount of UDP traffic. The traffic is primarily generated by the around $20{,}000$ active clients that send and receive data through the Dante server. The load results in around 90% of total CPU usage from the Dante processes, of which there are around $1200 - 1400$. Memory consumption is not a problem on the machine, which has 32 GB of memory, and no swapping occurs.

This machine would appear to have been well dimensioned for the load and generally uses most of the CPU and network capacity, without appearing overloaded, at least for most of the two week period that was examined.

Some periods of interesting behavior that break with the overall pattern were observed, including significant reductions in network traffic, sudden spikes in CPU load and changes in memory usage. Some of these events are examined in later sections.

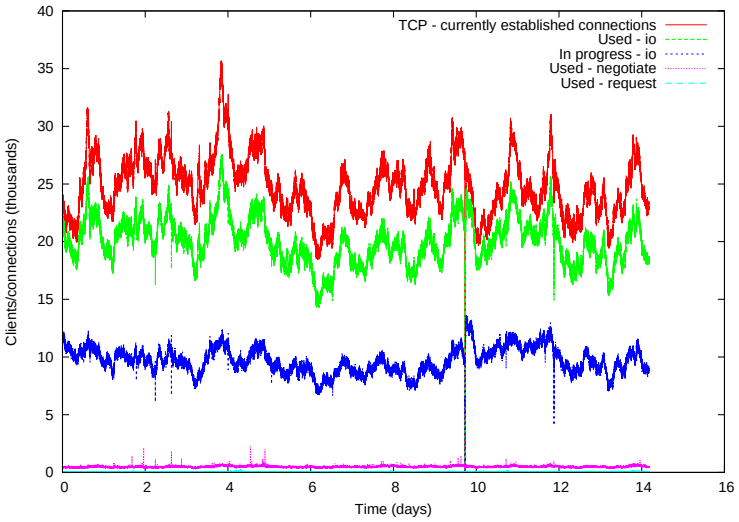

Figure 15: Client overview

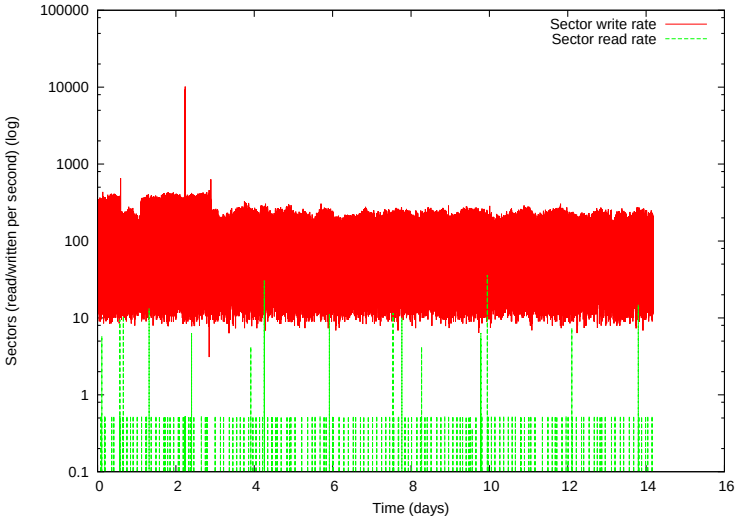

Figure 16: Disk I/O rate

7 Memory changes

Figure 14 shows a gradual increase in the *Swap cache* and *Inactive memory* for the entire time data is collected on the machine, with the inactive memory having one spike down towards zero after two days, shown in more detail in Figure 17.

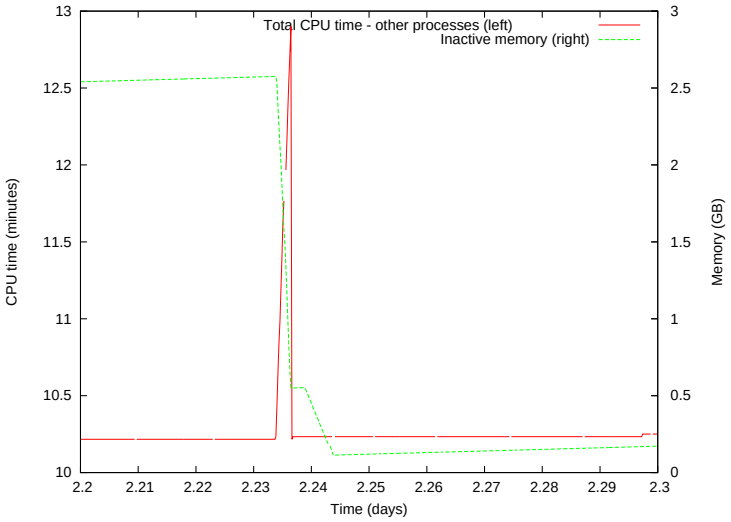

Figure 17: Process CPU usage and inactive memory usage

The spike down in inactive memory corresponds to a short spike up in CPU usage. The CPU time line corresponds to the CPU time of non-Dante processes, with some constantly running processes removed, and turned out to have been caused by a combination of *gzip* and *tar*. At this time a copy had been made of the log data accumulated for these measurements so far and this action resulted in the change in inactive memory.

The data collection script regularly writes the output from *ps*, *top* and various other commands to disk, and the Linux kernel appears to retain a copy of this data in main memory. When a copy was made of the collected data after two days, the log data was presumably removed from the inactive list maintained by the kernel and the newly created compressed archive was added to the swap cache.

After this point, the inactive memory grows constantly again, along with the swap cache, which is not affected by the data being accessed. Because there is no shortage of memory in the machine, both values have room to grow for the entire period the data is collected.

Figure 18 shows the inactive memory/swap cache data along with the number of aggregated write requests, as reported by *vmstat*. The values grow at roughly the same rate and the total size of the swap cache at the end of the two weeks is around 18 GB, which corresponds well with the uncompressed measurement data, which has a size of around 17.2 GB.

The gradual increase being a result of the data collection process also explains why the inactive memory and swap cache are almost empty at the start of the measurement

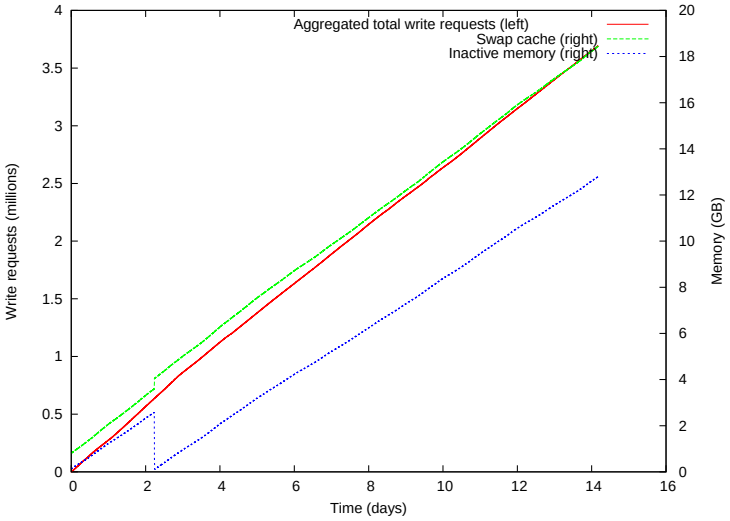

Figure 18: Disk write requests and memory usage

period and then gradually increase the entire time; the values start increasing at the same time as the measurement script is started. In conclusion, the increase in inactive memory/swap cache is not an indication of a memory leak or similar in Dante.

A more detailed look at the memory usage can be seen in Figure 19. The data in this figure is based on summarizing the memory values for each Dante process, which gives higher total values than what was actually used (due to shared memory not being accounted for), but the relative changes in the values are still interesting. As can be seen, there are no indications of gradual memory increases here either. Because the data was collected with a very high load over two weeks, it would be expected that any memory leaks would be visible in this period, unless the memory loss is very small, but no indications of memory leaks can be observed.

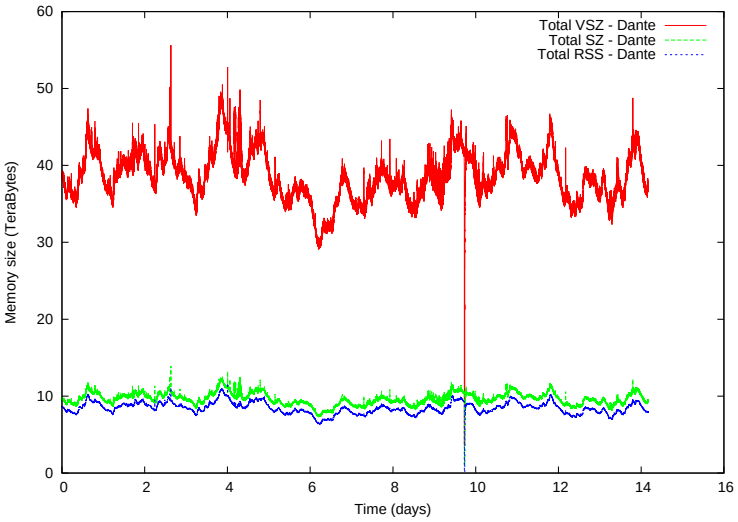

Figure 19: Dante memory usage

8 Traffic loss spike

A significant spike down in traffic at the end of day 10 can be observed in most of the data presented in Section 6, such as in Figure 8. This event is examined in more detail in this section.

8.1 Network traffic

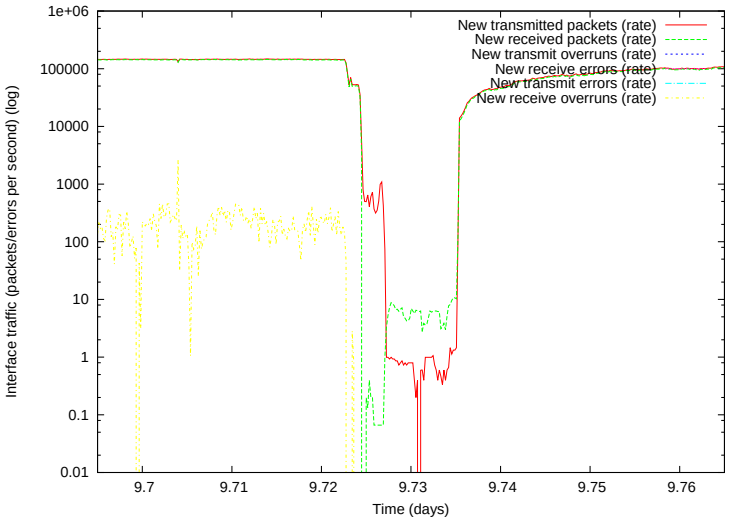

Figure 20: Spike network traffic changes (*eth0*)

A closeup of this spike can be seen in Figure 20, which shows a logarithmic overview of the number of packets transmitted per second. Each 0.01 increment on the x-axis corresponds to roughly 15 minutes, giving a total of roughly 90 minutes for the whole period shown. At a point after 9.72, the number of received and transmitted packets starts to fall, with the number of received packets falling fastest. Packets are still sent, at a lower rate, for some minutes, before the send rate also falls to almost zero. After roughly fifteen minutes with very little traffic, the traffic starts recovering with a spike upwards in traffic. The interface saw frequent receive overruns before the spike down but not immediately after the traffic starts increasing. Possibly due to the load on the machine being lower afterwards for the period shown.

A closeup of the point where the traffic starts going up again is shown in Figure 21. The received packet rate is slightly higher than the rate for transmitted packets, but neither can be seen in the collected data to start rising first.

Figure 22 shows various TCP related network statistics, including the number of connection resets sent and received, and the rate of new connections. All values appear to fall off at the same time; there is no indication that connections being blocked (resulting in a sharp increase in received connection resets) is the reason for the drop in traffic. After the spike down, the number of incoming/outgoing connections resumes

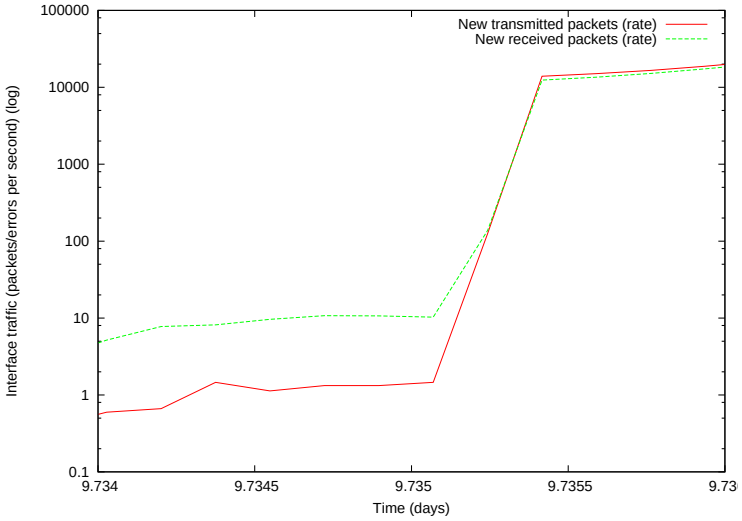

Figure 21: Spike network traffic changes (*eth0*), recovery

at the same rate as before the spike, but the number of connection resets sent by the machine is notably lower.

The number of established connections, as reported by *netstat*, can also be seen in Figure 23 to largely recover. From around 27,000 connections it falls to 191 at the lowest point and then recovers to around 26,000 again.

The data for all protocols is shown together in Figure 24. The number of transmitted and received TCP segments recovers to almost the level seen before the spike down at the end of the shown time period, but there is still less traffic transmitted in total. The difference is caused by UDP, which appears to recover much slower. The reason for this is not known, but might be application related; perhaps the used UDP applications need more time to recover.

By looking only at the traffic data, there is no clear indication as to the reason for the sudden loss of traffic. Possible explanations would include network problems or some problem with the Dante server that suddenly resulted in many Dante processes either ending client sessions or terminating. A more detailed analysis provides more information below.

8.2 Underlying event

The information collected about Dante shows how it behaves in this period, with Figure 25 showing an overview of the processes on the machine, grouped into *Dante* and *Other* (non-Dante) processes. Several observations can be made from this plot, firstly that the total number of Dante processes falls significantly, while there is no significant change in the number of non-Dante processes on the machine. After falling from around 900 to 84 at the lowest, the number of Dante processes then quickly increases to around 950. However, in regards to trying to determine what caused the drop, the most

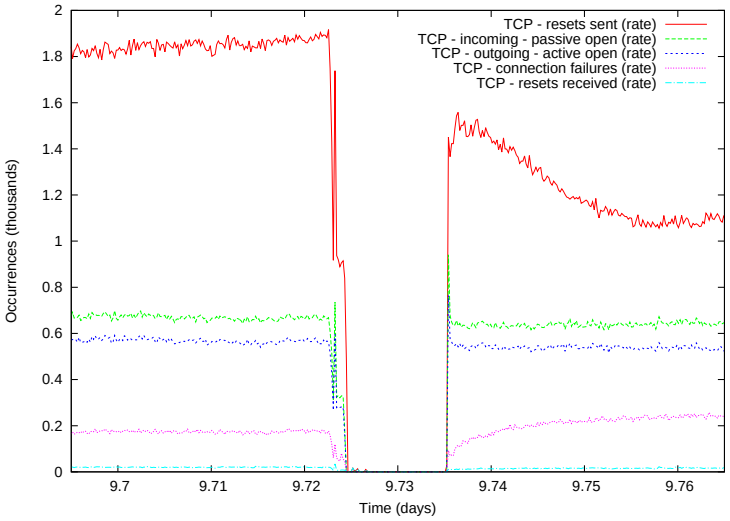

Figure 22: Spike TCP network statistics

important observation is the change in *Running/Sleeping* processes at the point before the drop starts. There is a constant gap between the *Total* number of processes and the number of *Sleeping* processes, showing that the processes on the machine (including Dante processes) are active. This gap then closes before the number of processes start falling, first gradually, then quickly. The Dante processes appear to have been inactive before they start terminating, which would be consistent with a network problem or change externally to the machine.

Figure 26 shows the interface packet data and process overview in the same figure, and this observation is confirmed here. The traffic on the interface starts falling off first, followed by an increase in the number of sleeping processes and then a reduction in the total number of processes. When the traffic returns, the number of processes quickly rises along with the traffic.

While it is possible that the traffic stops because all Dante processes suddenly stop sending traffic, this seems unlikely because the machine would still receive new incoming connections from new clients as this is handled by the kernel and not Dante, but Figures 22 and 23 show no indication of this. It seems very likely that the spike down is caused by network traffic no longer reaching the machine.

This situation also provides a good opportunity to examine how Dante reacts to this type of event; going suddenly from handling a large amount of clients and traffic to all client sessions ending.

8.3 Dante reaction

A more detailed overview of the different Dante process types can be found in Figure 27. Most of the Dante processes are *io* processes and these are also the ones most affected by the loss of traffic. The number of *io* processes falls from over 750 to 4, then

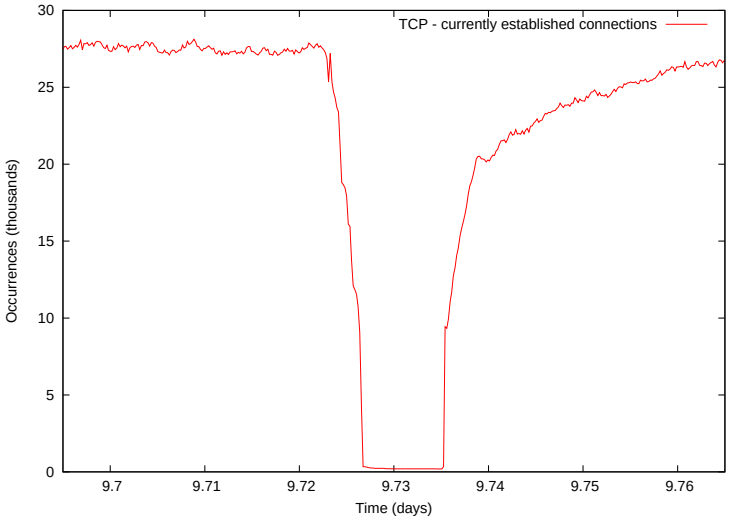

Figure 23: Spike established TCP connections

increases to more than 800. In other words, slightly more than it started with. This is slightly surprising because the previous plots have shown that the total traffic is lower after the spike than it was before it, but Dante will need some time to fine-tune the number of different processes it needs to handle the traffic load.

The number of *request* processes lies at around 115 before the loss of traffic, then falls to 71, before returning to around $110 - 130$ processes.

The *request* processes primarily handle connection setup and checking of whether the ACL rules permit connecting to the target server. That the number of *request* processes is not falling further might indicate improper, or at least suboptimal, cleanup of *request* processes in Dante. This should be investigated further so the exact cause can be determined.

The lower range of the figure is shown in Figure 28, and shows the other process types, that have relatively few processes in total. Since the number of mother and monitor processes is constant, only the number of *negotiate* processes is interesting. It falls from around 15 to 4 and then bounces back to around the same number, as can be expected.

From this it would appear that with the exception of the request process, the Dante server is able to quickly adapt resource usage (in the form of processes) to the current load, both when the load drops and when it increases. As the data is collected at 15 second intervals, there might have been spikes between the data points that were not observed, but from the data available, it would appear that both the termination and addition of child processes by Dante is fairly quick.

The Dante *io* and *negotiate* processes handle multiple clients, each client corresponding to one used *slot* in one process. The number of used slots is shown in Figure 29. For the *io* processes, the number of clients falls from above 20,000 to zero. All used slot types fall to zero, except the number of *request* process slots, which has 7 as

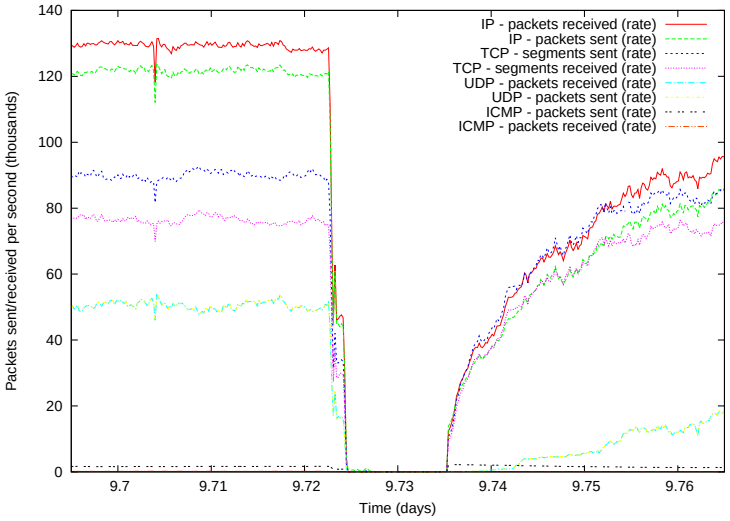

Figure 24: Spike all protocols network statistics

the lowest value. What the *request* process is waiting for and why it cannot close these 7 slots is unknown at the moment.

The number of clients quickly recover, with the number of used *io* and *in progress* slots actually going to a higher level than before the spike down. The higher number of used slots explains the higher number of *io* processes seen in Figure 27, but not why the number of active clients appears to be higher than would be expected by looking at number of established connections or the amount of transmitted traffic. There is no available information about whether a client uses UDP or TCP so it is not possible to see if the composition of clients has changed in some way, but it would appear that the amount of data transmitted per client is lower than before the spike.

Figure 30 confirms this, showing the average transfer rate (based on the sum of read and written bytes for all interfaces) per active (not *in progress*) *io* process slot. The rate is around 140 Kilobit/s before the spike, then falls down to around zero, before spiking up and gradually increasing to around 120 Kilobit/s, i.e., slightly lower than before the spike down. Possibly this is thus partly a client application issue, with clients initially sending and receiving less data.

8.4 Resource usage

The changes in the number of Dante processes and the network load also makes it possible to see the effects on resource usage on the system. Figure 31 shows the memory usage, as reported by *vmstat*. The amount of used and active memory falls (and the amount of free memory rises correspondingly) at the start of the spike down, and then gradually increases afterwards. This is as would be expected due to the termination of Dante processes, and later, the creation of new processes as the network traffic returns.

Figure 32 shows the rate of interrupts and CPU context switches, as reported by

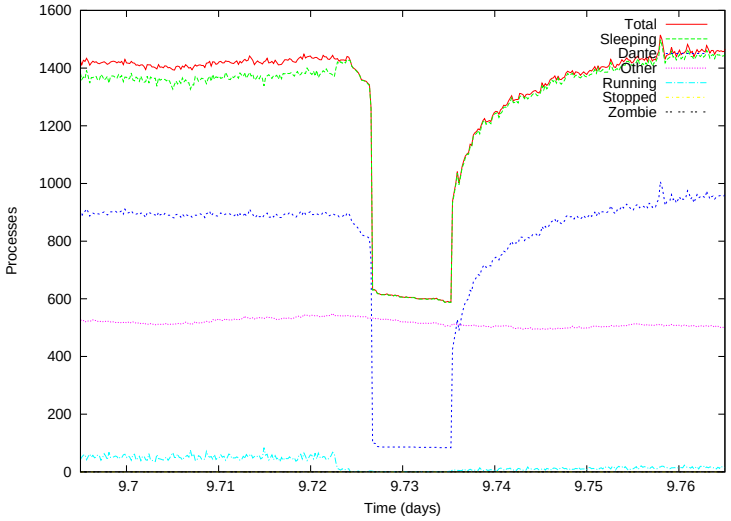

Figure 25: Spike process overview

vmstat. The number of context switches returns to around the same level as before the spike in the shown time interval, while the number of interrupts reaches a higher level. It is difficult to say for certain, but it would appear that the number of interrupts follows the number of used Dante *io* slots, while the number of CPU context switches more closely follow the number of Dante processes, while also showing some of the variations that can be seen in the packet rates. As multiple processes handle the sending and reception of client data it would seem logical that a higher number of clients (and processes) would result in an increased number of context switches, and that there is also a relation between I/O and context switches.

The general CPU usage is shown in Figure 33. Before the spike, around 90% of the CPU time is used by Dante processes, but this falls to zero when the spike down occurs. It then gradually increases up towards 70% as network traffic is again being forwarded. The user and system CPU usage start at around 40% and 45%, fall to less than one percent, and then start increasing along with the increase in Dante CPU usage.

This behavior is as would be expected; there are no indications in the data of sudden spikes or CPU usage when there is no traffic. There is no apparent change in the ratio for user and system CPU time before and after the spike, and the idle CPU time goes to around 99% during the spike down, indicating that there is very little activity on the machine when there is no network traffic.

Figure 34 shows the load average for the period, and the difference between the load average before and after the spike is quite interesting. Before the spike it lies at around 50, and the *one minute* load average also falls fairly quickly down to zero, but for the period shown, it rises to a much lower value than would be expected based on the amount of traffic being forwarded and the amount of processes being active. The final value in the shown interval is below 20, and less than half the value from before the interval. It does however match the amount of UDP traffic quite well, which as can

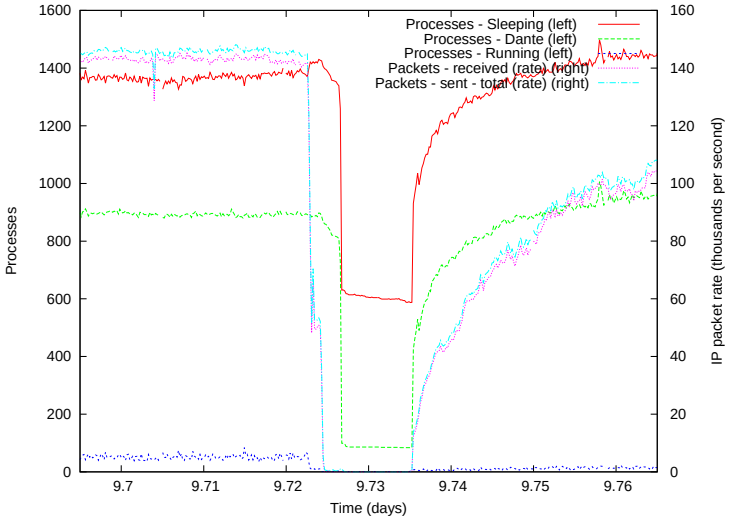

Figure 26: Spike Dante processes and interface traffic

be seen in Figure 24 also recovers to less than half of the initial value.

This indicates that the overhead from UDP traffic might be larger than the overhead from TCP, and this relationship is examined in more detail in Section 9.

8.5 Additional observations

Finally, the spike down provides an opportunity to take a closer look at the effects of system load on the variation in time between each data point in the data collected for these measurements. Figure 35 shows time between each data point for several of the commands that are executed during the data collection. Before the spike down, the time between each timestamp clearly differs for most of the values but stays in the range between 15.2 and 15.4 seconds for most of the values. During the period of very little CPU activity, the interval times fall to below 15.1 seconds and there is very little variation between each data point. When the machine again starts forwarding traffic, the variation increases, and for *ps*, the time between data points clearly also increases, likely due to the gradually increasing number of processes. The overall difference is however not that large; the biggest difference is in how variable the interval durations are.

As noted in Section 4, a fixed delay of 15 seconds obtained with *sleep* is used in the data collection script. Ideally the interval should be 15 seconds in all cases (when possible), so the data collection scripts might likely benefit from being updated to consider the time already passed before sleeping to give more exact intervals.

8.6 Summary

The spike down in the traffic data appears to be caused by an external event that results in no traffic from the Internet reaching the machine.

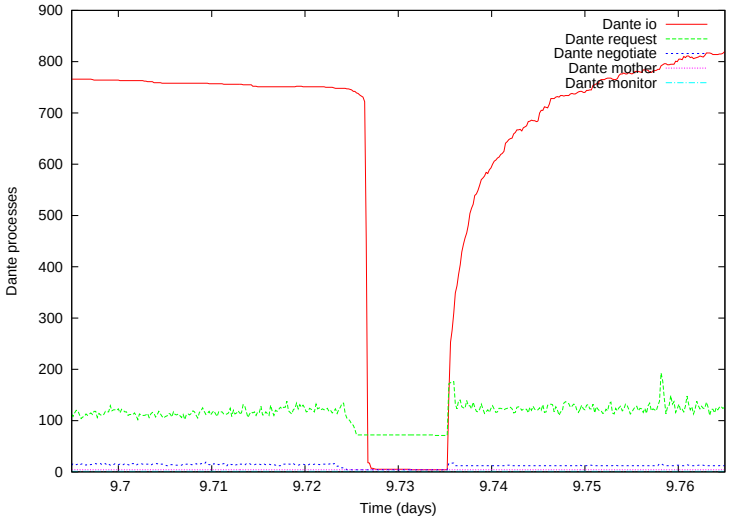

Figure 27: Spike Dante process types

Dante appears to handle the sudden change in traffic load quite well, both when no traffic is being received and when it suddenly returns. Dante processes handling *io* are terminated on the spike down and the resources consumed by Dante are significantly reduced. The opposite happens when traffic returns and Dante is able to quickly handle adding new clients.

Only the *request* processes appeared to not fully react to the change.

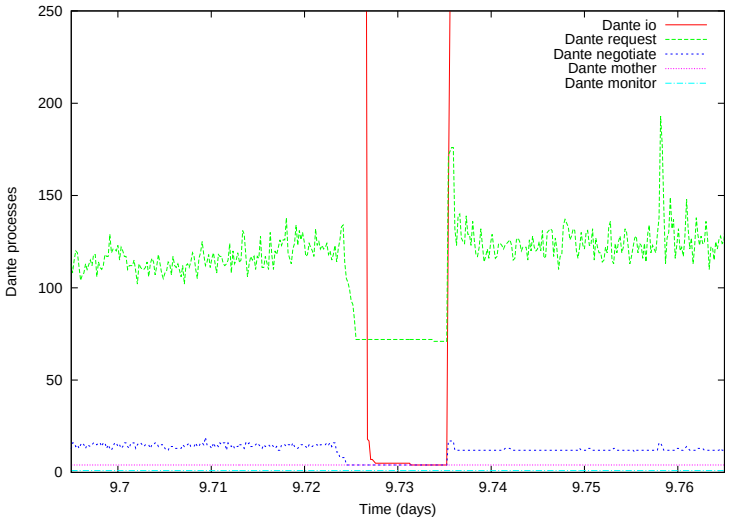

Figure 28: Spike Dante process types, lower range

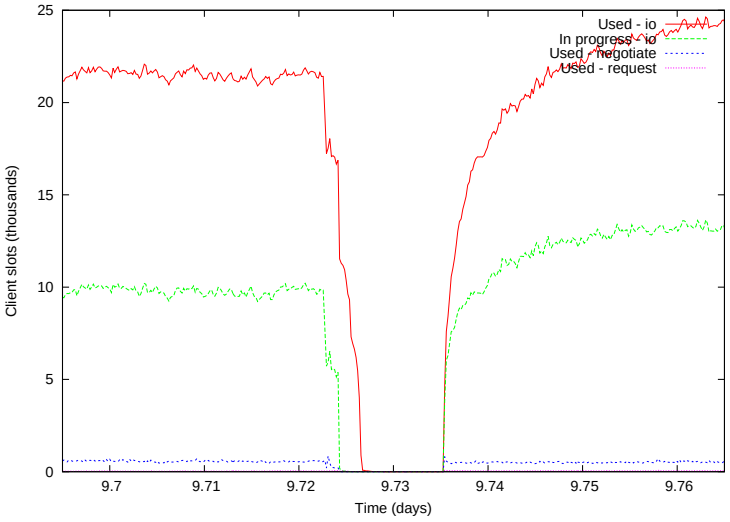

Figure 29: Spike Dante client slots

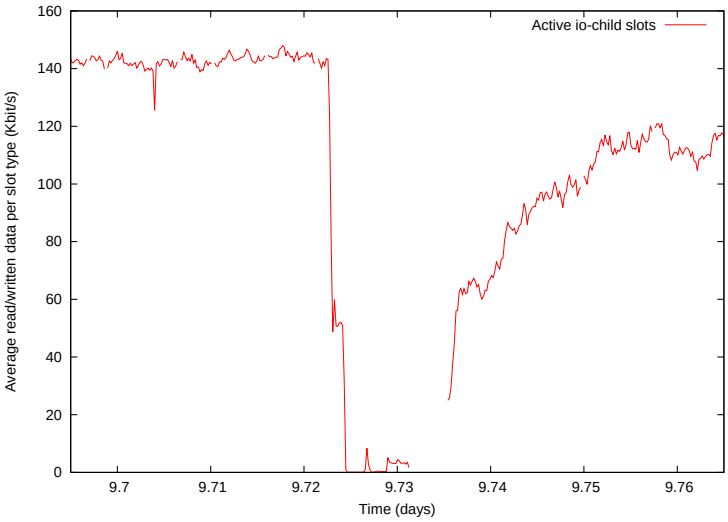

Figure 30: Spike average transfer rate per active *io* process slot

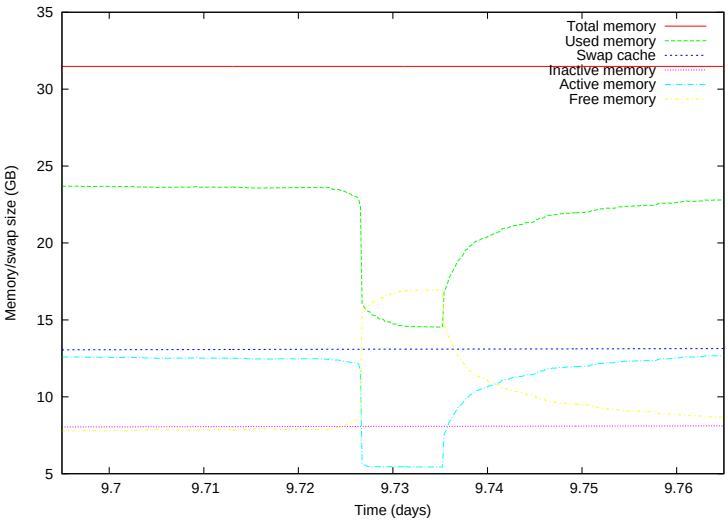

Figure 31: Spike memory usage

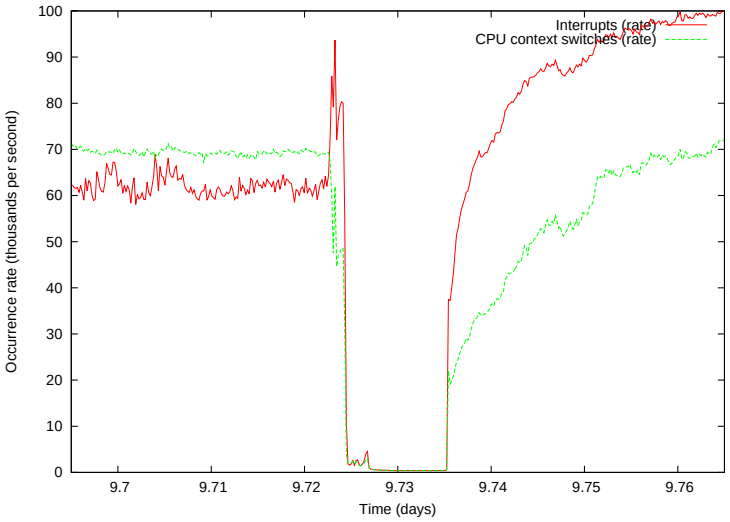

Figure 32: Spike interrupt and CPU context switch rates

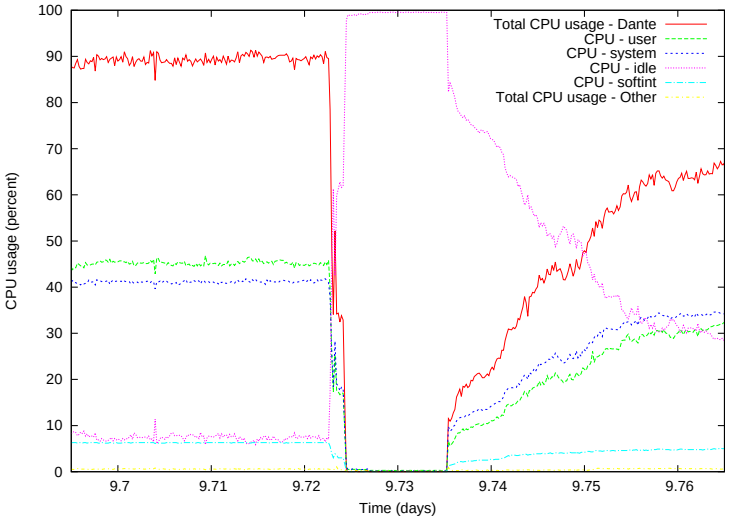

Figure 33: Spike CPU usage

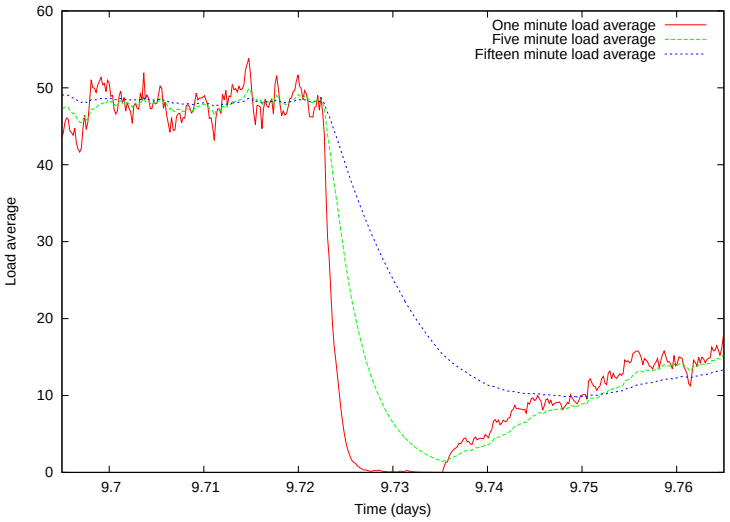

Figure 34: Spike load average

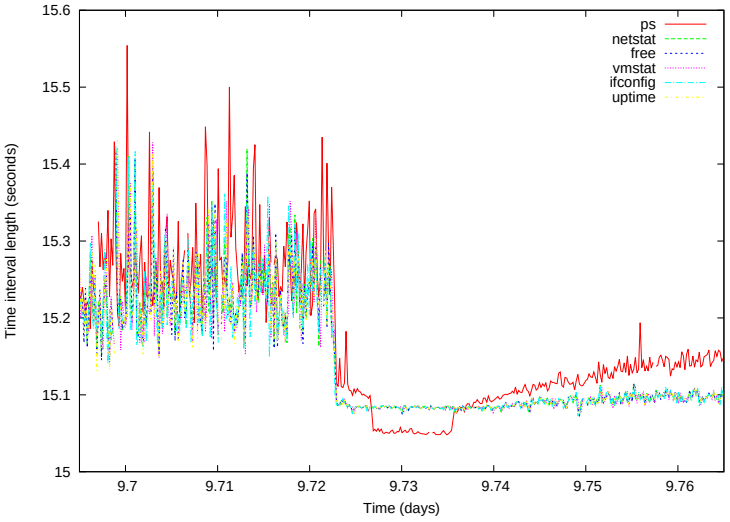

Figure 35: Spike data snapshot interval lengths

9 UDP traffic, interface errors and CPU usage

An observation made in Figure 34 of Section 8.4 is that the load average appears to be more closely related to the UDP traffic rate than is the case for TCP. This section examines this observation in more detail and attempts to determine if it is possible to say if the performance on the machine is limited by the CPU.

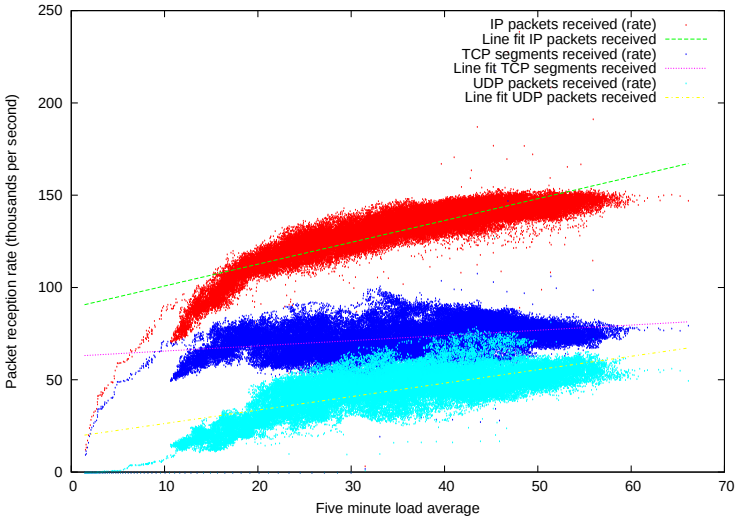

Figure 36: Packet rate and load average relationship

A scatter plot with the relationship between the five minute load average and the packet reception rates is shown in Figure 36. The data is based on the entire measurement period. The five minute load average mostly lies between 10 and 60. The highest packet/segment rate is seen for TCP, with a rate of around 75,000 packets per second for a large part of the time. The rate for UDP is lower for most of the values, at around 50,000 packets per second. Looking at the total number of received IP-packets, and the least squares fitted regression lines, it would seem that there is a relationship between the five minute load average and the packet rate. According to the Linux manual page for the *getloadavg()* function, the load average as returned by the function corresponds to *the number of processes in the system run queue averaged over various periods of time*. As Dante uses multiple processes for handling I/O, it would be expected that large increases in traffic would be a result of increases in the number of active clients and, as a result, an increase in the number of processes that potentially are waiting to be executed as there is only a limited number of CPUs in the machine.

By comparing the values for the UDP and TCP traffic, several observations can be made. Apart from the UDP rate generally being lower, the UDP rate also appears to be more closely correlated to the load average; higher UDP packet rates generally correspond to higher load averages. The regression line for the UDP traffic also has a steeper angle than the line for TCP traffic. It would seem that higher rates of UDP traffic might impose a higher load on the machine than TCP traffic. This is also to be

expected, as processing SOCKS UDP traffic incurs a higher overhead than processing SOCKS TCP traffic. The latter can be forwarded in both directions verbatim (unless encryption is enabled), while SOCKS UDP traffic to and from the SOCKS client needs to be encapsulated and decapsulated, due to the extra SOCKS UDP header that is added or removed.

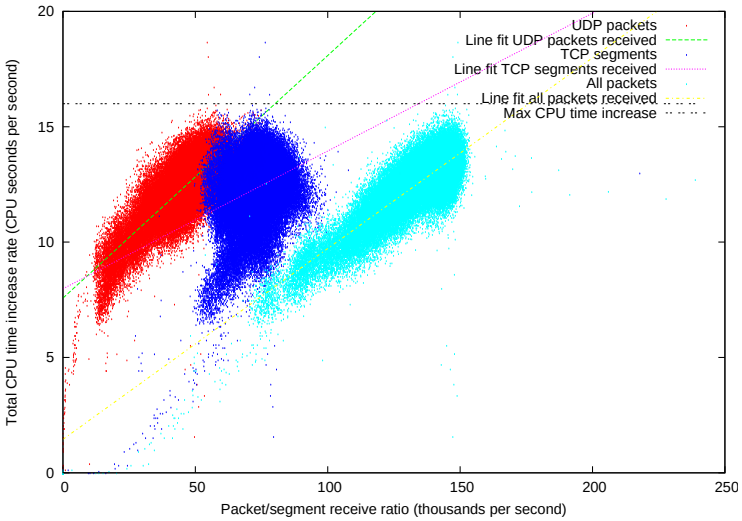

Figure 37: Packet rate and Dante *io* process CPU usage

The relationship between the packet rate and the rate of CPU time increase for all Dante *io* processes is shown in Figure 37. With 16 CPU cores, the highest possible CPU time increase per second is 16 CPU seconds (corresponding to Dante *io* processes running on each core for 100% of the time). The scatter plot shows the data for UDP, TCP and all protocol types, in addition to the least squares fitted regression line for each data set.

The data in this figure also appears to show a correlation between higher UDP packet rates and higher *io* process CPU usage. Based on the regression lines, this relationship appears to hold true for all three data sets, but for TCP to a much lesser degree than for UDP. System overhead likely means that using 100% of the available CPU time is not possible, so around $60{,}000$ UDP packets per second would appear to be the point at which the ability of this machine to transmit UDP packets at a higher rate will be limited by the available CPU resources, assuming an equivalent amount of TCP traffic is present.

Interface overruns and UDP send and receive errors are another indication of the machine possibly being overloaded. Figure 38 shows a scatter plot of the interface receive overruns and UDP errors (there are no interface transmit overrun errors in the data set). There are very few UDP send errors, but both interface receive overruns and UDP receive errors occur, especially when the Dante *io* processes use most of the available CPU time. The regression lines also indicate that higher *io* process CPU usage occurs at the same time as higher error rates.

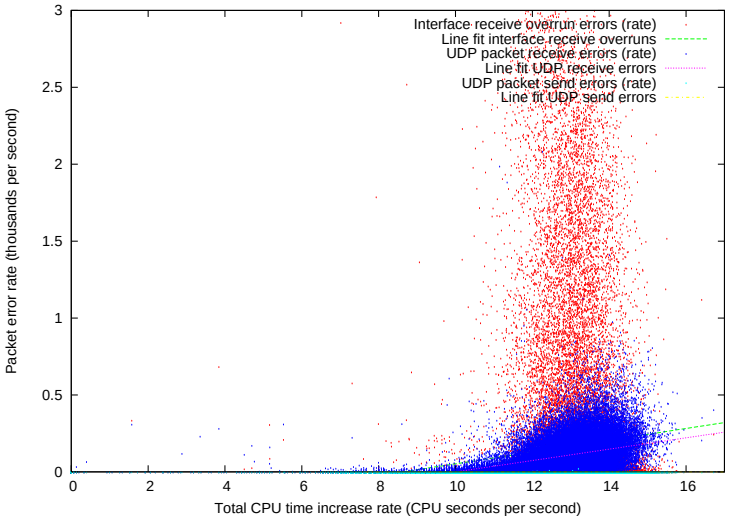

Figure 38: Packet error and Dante *io* process CPU relationship

The same error data is shown along with the interface data reception rates in Figure 39 and basically shows the same relationship; higher reception rates correspond to higher error rates. That overruns are more likely to occur when more traffic is being received is fairly logical, but the presence of errors would seem to be an indication of the machine being overloaded, as it is not always able to handle the data it receives. The lost packets would likely also result in lower performance as seen by the users due to the need for retransmissions.

To take a closer look at how frequent the errors are, Figure 40 shows a heatmap of the interface *receive rate* and the interface overrun error rate, and most of the time there are no overruns. Most of the values in the figure are found in the range between 0.74 and 0.81 Gigabit/s. Many of the overruns are also found in this traffic range.

A CDF plot of the errors in the range is shown in Figure 41 and roughly 73% of the time the receive rate is in this range there are no receive overruns. In other words, 27% of the time in this range there is at least 1 receive overrun per second, which seems quite high.

The variation in time between timestamps for data obtained by the collection script used to build the data set analyzed in this document appears to be a good indicator of the load on the system, as seen in Figure 35. Higher loads leads to less regular scheduling and increased time between each snapshot. Figure 42 shows the relationship between the interface receive overruns and the UDP packet reception errors and the time intervals for one of the executed data collection commands.

For the interface receive errors there is a slight tendency towards higher intervals, but it is more obvious for the UDP receive errors. Higher rates of UDP reception errors appear to occur along with increases in timestamp interval lengths. This relationship does not appear strange; if the machine is busy, it will likely affect both the ability of the kernel to keep up with data reception rates and handle CPU scheduling.

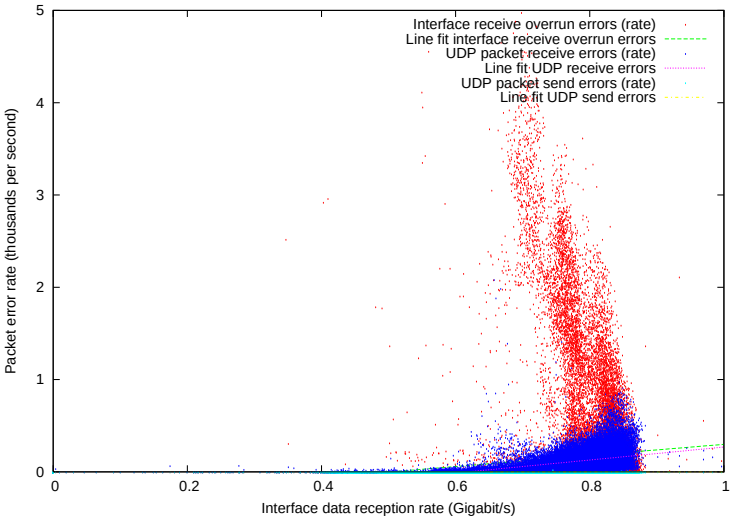

Figure 39: Packet error and interface rate relationship

The relationship between the packet error rate and the system CPU usage is shown in Figure 43. The interface receive errors clearly start to rise quickly after 40% system CPU usage is reached. The UDP receive error rate also starts rising, though less fast.

For the user CPU usage and packet error rates, the relationship is shown in Figure 44. Also here, similar behavior can be seen. For high user application CPU usage, the interface receive overruns start increasing, but the correlation does not appear to be as strong as for the system CPU, which is logical. The kernel is responsible for getting data from the NIC so when it is highly loaded, it will be less able to handle all incoming packets, but even if a user application is busy, this will not necessarily affect the ability of the kernel to handle network traffic. However, for a proxy like Dante, high user CPU times will lead to high system CPU times due to data forwarding being the main activity, so high rates of interface overruns will likely occur when the user CPU time is high, even though the two evidently are not as closely related as is the system CPU time and interface overruns.

However, for the UDP receive errors, the correlation between high user CPU time and higher rates of packet receive errors appears to hold. The figure clearly shows that error rates start rising along with the CPU usage after it reaches 40%.

As we have shown above, there appears to be a relationship between the UDP packet rates and the amount of CPU usage. Unfortunately, we do not have any information about byte transfer rates for UDP, but it would appear that with high rates of UDP traffic the machine will become unable to handle the received traffic due to the overhead of handling UDP.

It is difficult to say if the CPU is what is limiting performance on this machine, as there might be other external factors that contribute to limiting the achievable throughput (the machine does for example frequently receive ICMP *source-quench* messages when it transmits at higher rates, as shown in Figure 45, but we do not know if the

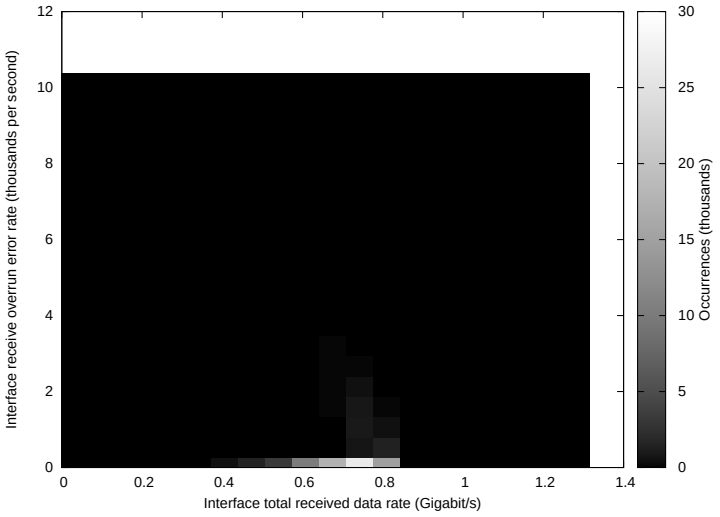

Figure 40: Interface error and receive rate heatmap

kernel acts on these to limit TCP rates, and what, if any, effect this has on throughput).

However, even if the machine is not limited by the CPU at present, there does not appear to be much room for forwarding UDP traffic at higher rates, without being limited by the available CPU resources. Optimizing the overhead of forwarding UDP in Dante, to the extent that this is possible, would likely help here. It is also possible that a different number of clients per process would help, but more systematic testing will be needed to determine this.

The use of *splice()*, or a similar copy elimination API, would likely also have an effect by reducing the system CPU time. Doing this might be difficult for UDP, but having it for TCP would still provide a benefit for the system overall by freeing up CPU capacity.

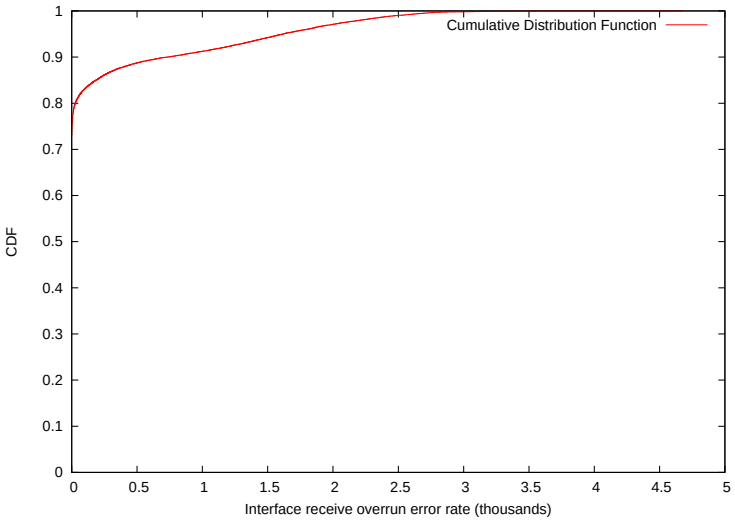

Figure 41: Interface error and receive rate subset CDF

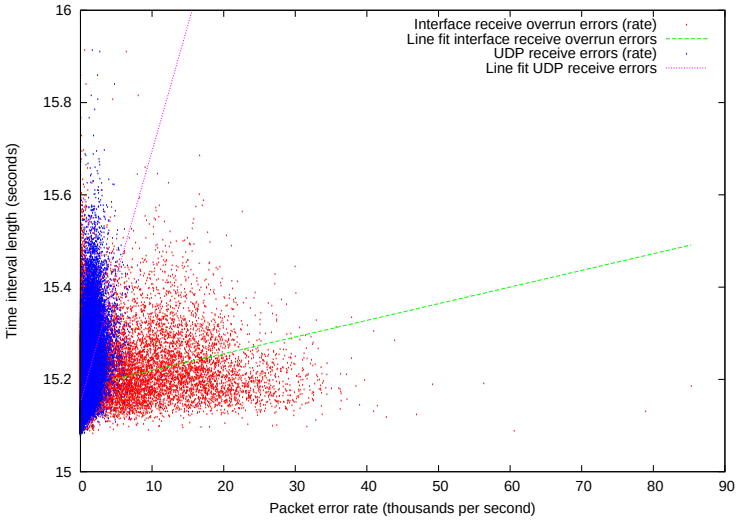

Figure 42: Error rates and snapshot interval lengths

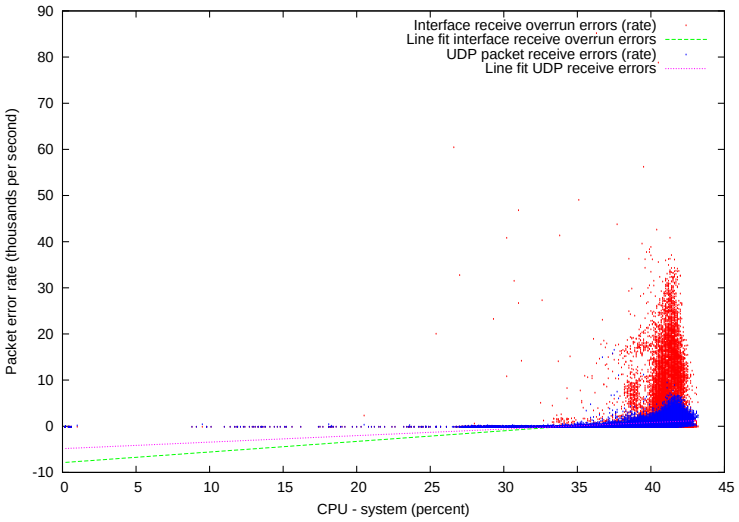

Figure 43: System CPU usage and packet errors

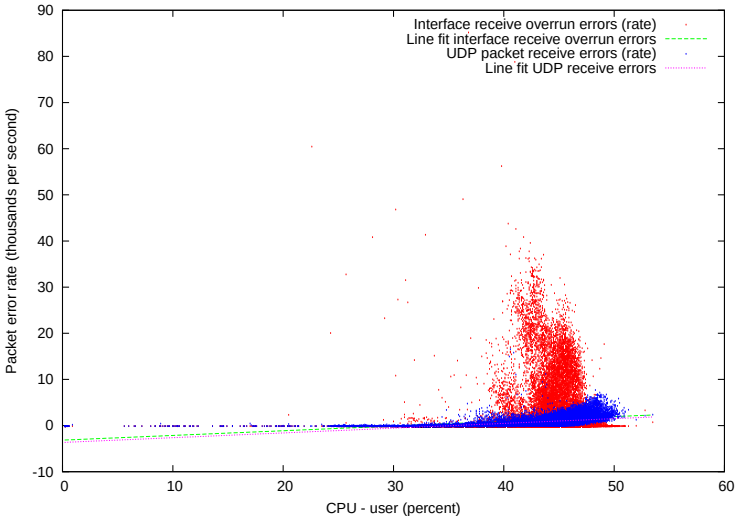

Figure 44: User CPU usage and packet errors

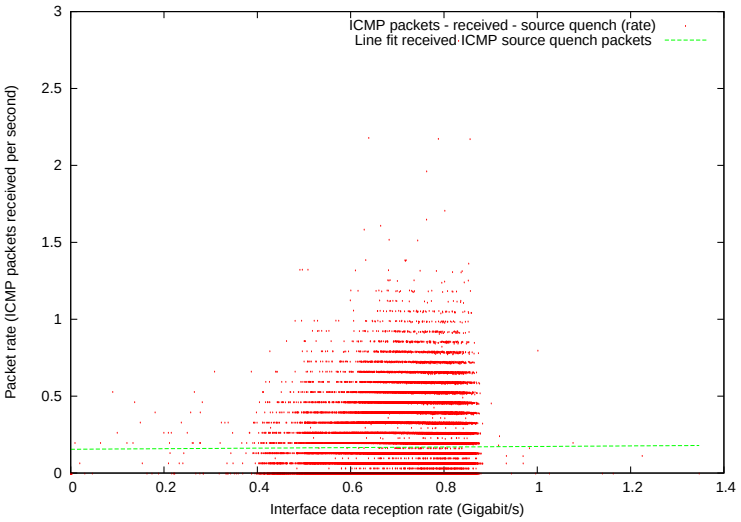

Figure 45: ICMP *source quench* receive rate and interface rate relationship

10 High timestamp delay

In this section, we analyze one of the points in the data set where there is a long delay between the collected data points in the measurement data, indicating that the machine was so busy that data collection could not be done on time.

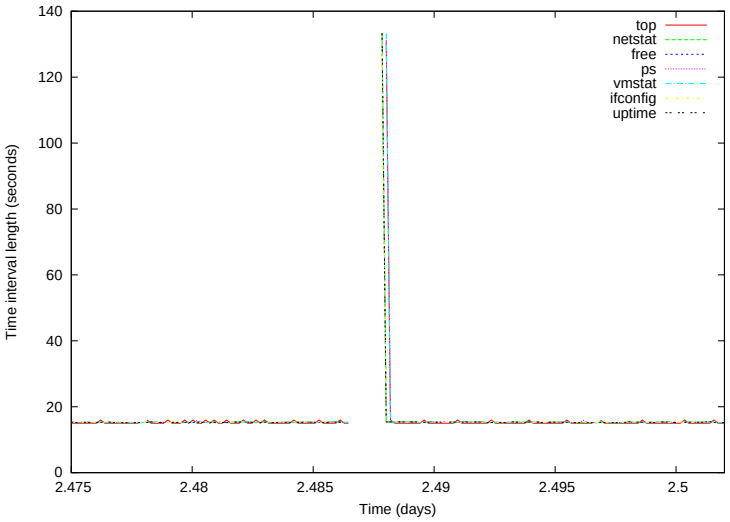

Figure 46: Time skip snapshot interval lengths

Section 4 noted that the highest delay between the data points collected during the measurement was 133.36 seconds, and this data point is shown in Figure 46. All commands show a long gap for this data point. With 15 seconds being the sample rate, nine data points should have been collected during this period, and not only a single data point.

There are also indications that the information collected in the last data point before the gap is not entirely correct. Figure 47 shows the total CPU time of the Dante mother processes. These processes never terminate, hence the CPU time values obtained for these processes after any sampling gap will include the CPU time used by the processes during the gap. The CPU time consumed by these processes has also shown itself to be relatively stable on this machine with the given traffic load.

Instead, this figure shows that the last data point before the gap has a significant step up in CPU usage, while the CPU time after the gap shows the total CPU time to continue at the same rate as it did before; it does not continue from a higher point, which would have indicated that the step up represented a significant increase in CPU usage at this point. It is more likely an indication of the data collection taking a long time, meaning that a significant amount of time passed between the data point timestamp being stored and the *ps* command completing. As the *ps* command likely does not collect information atomically, the collected data might have been gathered over many seconds, or it might have been simply delayed and represent a point in time several seconds after the recorded timestamp. It is not possible to know what actually occurred,

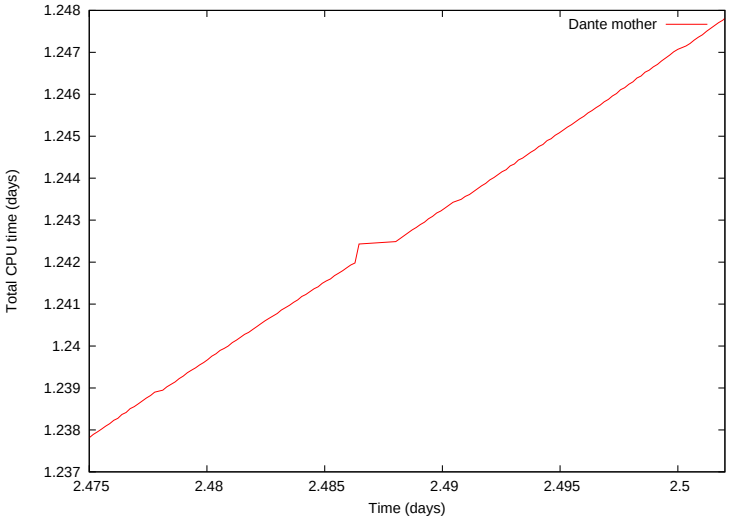

Figure 47: Time skip Dante mother process total CPU time

but the data collected in the last data point will need to be interpreted with caution due to the likely inaccuracy of the data.

The data collected from *top*, shown in Figure 48, also indicates that a larger amount of data was lost compared to the surrounding data points. The figure shows the total amount of CPU time for Dante and other applications, and the values for user/system/idle CPU time. It also shows the sum of all application values returned by *top*. Ignoring the fact that there might be some loss of accuracy due to the low precision in the percentages returned by top, it would be expected that if *top* was able to completely account for all CPU usage by all applications, the final summarized value would be 100%. However, as *top* also runs at regular intervals (here every 15 seconds), *top* will presumably not have information on CPU time for short lived processes that terminate before the data is collected by *top*. In other words, the lower the sum value in the plot is, the more CPU time was likely spent by short-lived and other terminated processes that were not recorded. In the figure, the CPU time sum value lies at around 97% most of the time, meaning that around 3% of the CPU time is likely spent by short-lived or terminated processes. For the first data point after the gap, the sum value falls towards 85%, meaning that around 15% of the CPU time was possibly spent by terminated processes that were not observed. The reduction might also be caused by *top* running at a time when the machine was highly loaded, leading to the collected data being inaccurate.

Looking at the other CPU usage values, there are some other inconsistencies that also stand out. For the two data points before and after the gap, the user CPU time increases and the idle time falls. The system value also falls. From this it would be expected that a higher amount of CPU time would be spent by applications, but the accumulated time for Dante falls, and for other processes if remains at a very low value. Assuming that the user/system values reported by *top* are correct, this is again an indication of information about applications being underreported. While it is possible

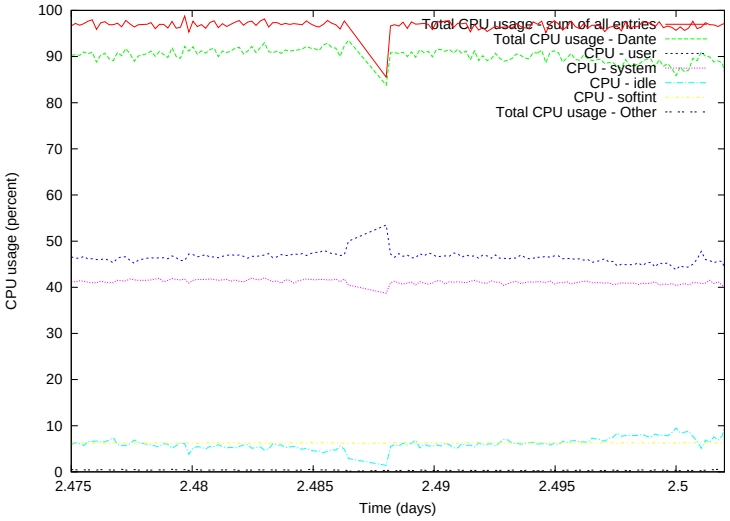

Figure 48: Time skip summarized process CPU usage

that this was caused by non-Dante processes, it seems more likely that short-lived and terminated Dante processes were responsible for the missing CPU time.

Looking at a more general value calculated by the system, the load average is shown in Figure 49, and there is a high spike up for the gap period. The machine was evidently highly loaded during this period.

In summary, the gap in the collected data points appears to have occurred at a time the machine was highly loaded, resulting in the data collection script hanging for almost two minutes. As very little data was collected during the period the machine was highly loaded, and the information that was collected might not be entirely correct, it becomes difficult to attempt to determine what actually occurred during this time, and the reason why it occurred. Transient information such as process information collected by *ps* and *top* appears to be least reliable, while more persistent sources of information such as the aggregated counters obtainable via *ifconfig* and *netstat* should be more reliable, as long as the possibility that the data in the last data point before the gap corresponds to a slightly later point in time is kept in mind.

To get a general idea of the events that occurred during the gap with missing data points, we first look at some of the persistent values that describe system behavior.

The total number of read/written sectors is shown in Figure 50. There are no read operations reported, and no apparent spike in write operations. Instead, there appears to have been fewer write operations made during the gap than before and after it; the values continue to increase at the same rate after the gap. It does not look like disk I/O was the cause of the gap, but the high load more likely resulted in delays in write operations from the data collection script as the data set also shows an increase in write delays for this period (not shown).

Looking at the amount of transmitted bytes on the machine, shown in Figure 51, there is no indication that there was a spike up in network traffic. It looks like there was

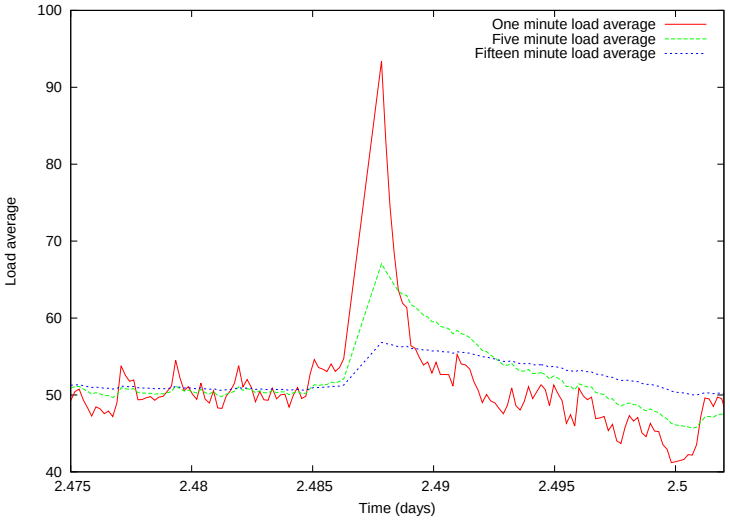

Figure 49: Time skip system load average

no change in the amount of transmitted data during the gap; data appears to have been transmitted at the same rate as before and after the gap. The same appears to be the case for the number of transmitted packets and established/received connections (not shown).

From the above values there is no apparent reason for the high load; it is necessary to examine also the transient data in more detail.

This data must, as noted above, be interpreted with some care, but it does also provide some information about events that occurred during the gap.

Figure 52 shows an overview of the Dante/non-Dante processes, as reported by *ps*. Nothing can as expected be said about the period without data points, but it can be seen that there are no big spikes up or other significant changes either before or after the gap.

The division between different Dante process types is shown in Figure 53. For the Dante *io* processes, there is no apparent change of significance, while both *request* and *negotiate* type processes appear to show an increase both before and after the gap. This might, as noted above, be due to inaccuracy in these data points, but it could also be an indication of higher numbers of these process types having existed during the gap.

A summary of the observed (via *ps*) terminated and added Dante *negotiate* processes can be seen in Figure 54. For the gap period there is a step up in the number of both terminated and new processes. As visualized in Figure 48 and noted above, this number will likely be underreported, meaning that a higher number of Dante processes was likely created and terminated during the gap period than for most of the surrounding time. There is however also an additional small step up at the beginning of the plot, so these process spikes might be something that occurs from time to time.

The *request* processes (not shown) do not show similar behavior, but increases in these processes might be very short-lived and not be reflected in the data.

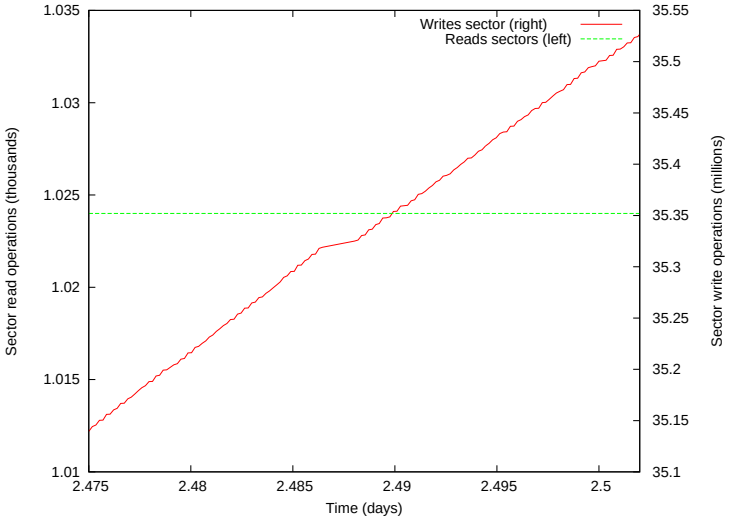

Figure 50: Time skip disk I/O

The CPU time of the *negotiate* processes, shown in Figure 55, provides some indication of the behavior during the gap period. Two types of CPU time are shown. The aggregated total CPU time is calculated based on the known increases in CPU consumption for *negotiate* processes over the entire measurement period, while the current CPU usage corresponds to the sum of CPU usage of the currently running *negotiate* processes. The figure shows two types of behaviors; a distinct step up in the known CPU time consumption of *negotiate* processes, meaning that these processes did more work than for the surrounding time in this period. The current total shows a reduction in the CPU time during the gap, meaning that there has been a certain amount of churn in the *negotiate* processes; most or all of the *negotiate* processes with a non-zero CPU time running before the gap are no longer running after the gap.

As there does not appear to have been any significant changes for the other process types, the behavior of the *negotiate* processes provides a possible explanation for the behavior during the gap. Unless the behavior was the result of a bug, a possible cause would be a spike up in short lived requests to the Dante server, that terminate during, or shortly after, SOCKS processing. The server is already fairly highly loaded and a burst of short requests would likely lead to new processes being created. An increase in the number of processes and the number of short-lived requests would also lead to increased overhead from Dante, as Dante currently removes idle child processes after a certain amount of requests or child process run-time has passed.

If there is a significant number of requests, it would be expected that the running *negotiate* processes would never be entirely idle with no clients, but it is clear from the observed behavior that there has been both a significant amount of CPU usage from *negotiate* processes and a replacement of the running *negotiate* processes.

This could be the result of the client behavior, with many negotiate processes being terminated after a period with many requests ends, or if could be the result of how

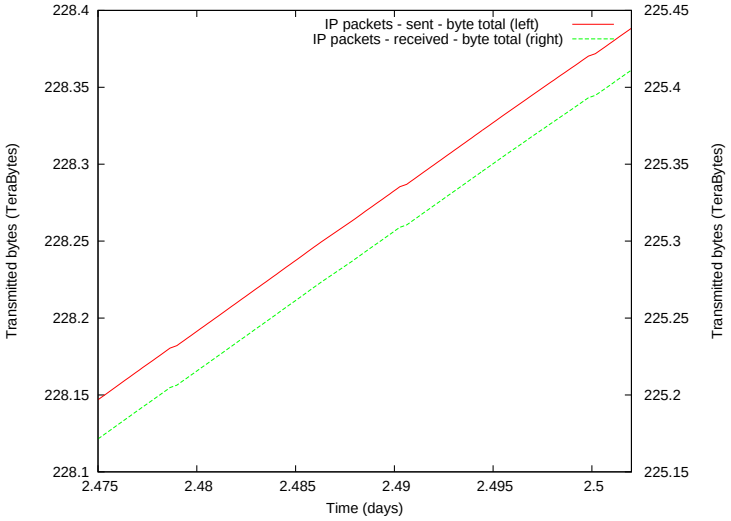

Figure 51: Time skip network I/O

requests are divided between the running *negotiate* processes, with many clients maybe being handled by some processes and few or no requests being handled by others.

Currently, two mechanisms are used in Dante to determine if processes should be terminated: process runtime and the total number of handled client requests. If, rather than using available client slots in existing child processes, the behavior of Dante with the client load that occurred in this time period resulted in processes without any clients being terminated based on the process runtime or the number of requests handled, and then new processes being immediately created to handle new clients, it might partly explain the high load on the machine, as the behavior of Dante would amplify the already increased load in these situations by increasing the overhead from Dante further due to processes constantly being created and terminated, instead of being reused.

However, there is no information available on the number of processes actually created by Dante during this period, and no information about the number of requests handled by the different process types, so this is only speculation based on the information available. The only thing that can be said for certain is that the machine was overloaded during this period and that there appears to have been some churn in *negotiate processes*.

Assuming that process replacement is part of the problem, it could be that changing the way in which child processes termination is handled would improve the behavior of Dante in similar situations of high load.

There are generally two reasons for dynamically terminating running processes:

- Adjusting to the current load.
- Reducing resource consumption due to leaks in *libc* or other third-party libraries external to the Dante code itself.

With new processes forked to handle increases in load, it is necessary to terminate

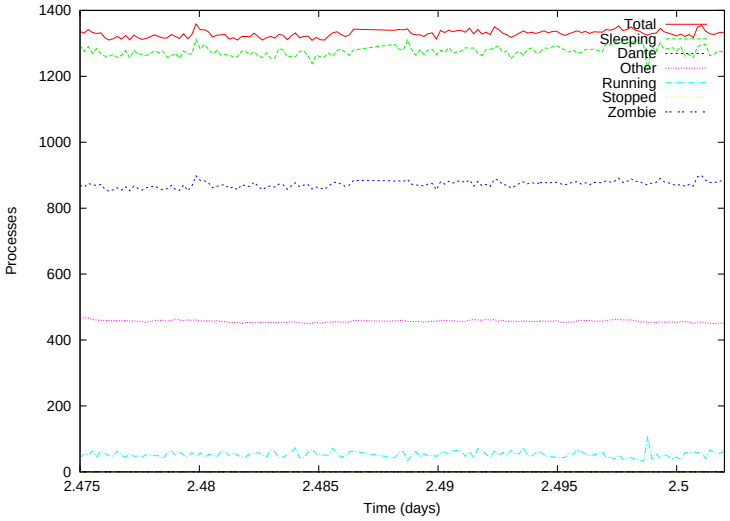

Figure 52: Time skip process counts

processes when the load decreases, and a decrease in load should be recognizable by a corresponding increase in the number of free client slots. A sustained period with a high number of free slots, and without any decrease in the number of free slots, would possibly be a situation in which it would be prudent to decrease the number of processes as it indicates that the load is not increasing.

Handling the second point should ideally not be necessary on most platforms, but there are platforms/platform versions where long running processes might experience resource leaks. For the *negotiate* and *request* process types, all requests will be limited in time and the amount of transmitted data so the number of requests is the most relevant factor that will affect the likelihood of resource leaks. For the *io* processes there is no limit on the duration of client sessions, or on the amount of data that can be transmitted, so finding a good way to determine when to terminate a process might be more difficult.

Regardless of how this limit is set, to avoid unnecessary overhead from process termination and creation on platforms where resource leaks are not a problem, the limit used to determine if a process should be terminated, for purposes other than adjusting to reductions in system load, should likely be set quite high by default. It would then be possible for users on platforms with resource leakage problems to reduce the limit as appropriate. For other users, client processes could then run much longer, unless terminated due to the need to adapt to the current load, in order to reduce unnecessary process churn.

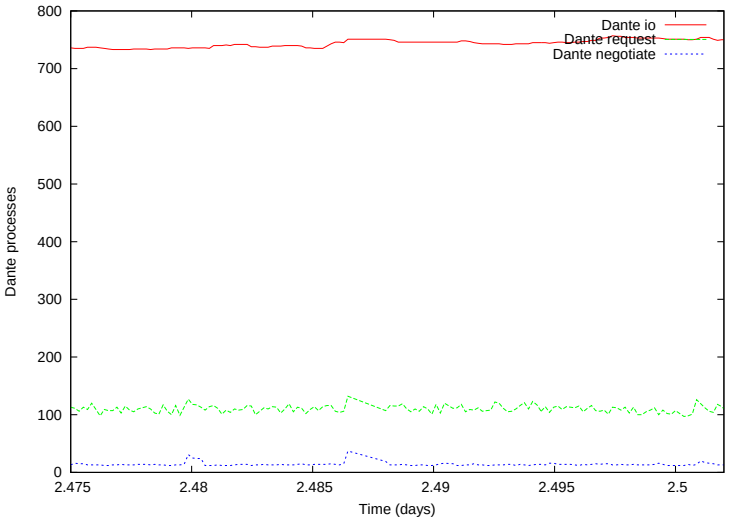

Figure 53: Time skip Dante process counts

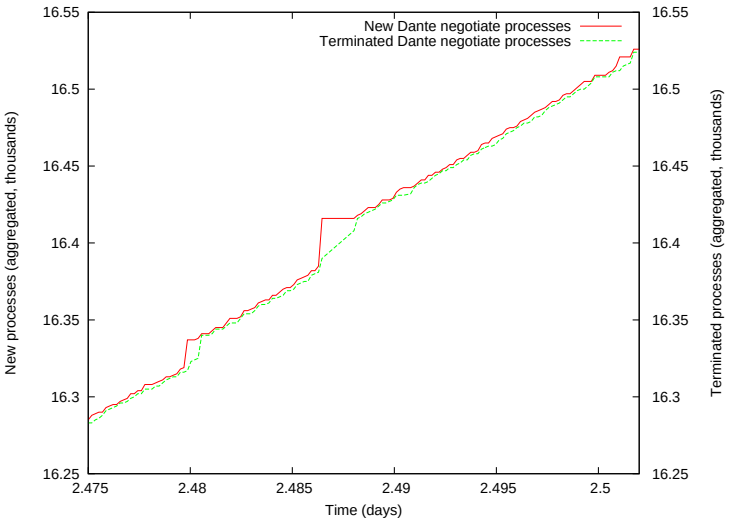

Figure 54: Time skip observed terminated and new Dante *negotiate* processes

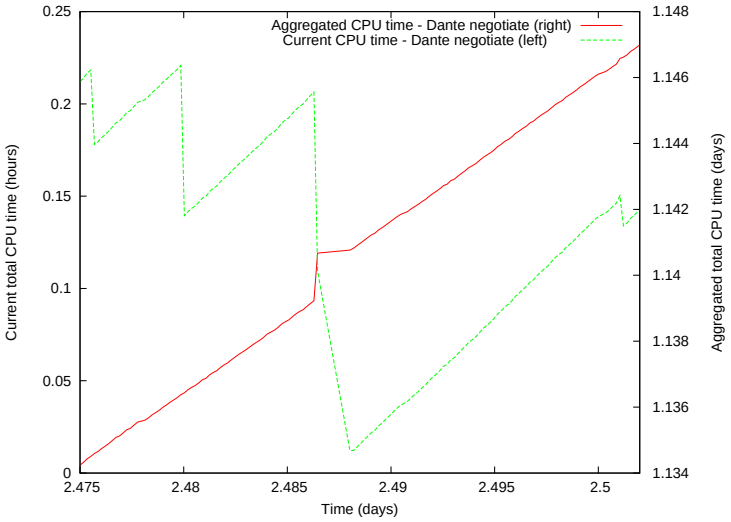

Figure 55: Time skip current and aggregated Dante *negotiate* process CPU time

11 Dante behavior

This section looks at some aspects of the behavior of Dante and how it can be observed on the machine.

11.1 Process churn

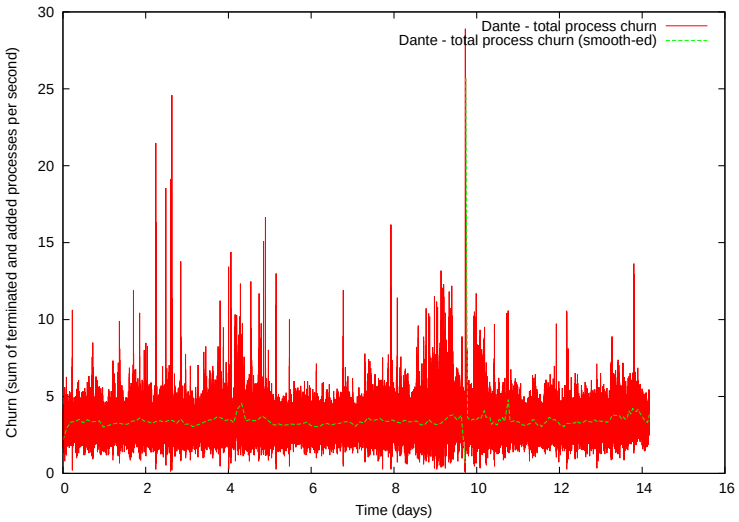

Figure 56: Observed Dante process churn

Dante makes use of multiple processes for handling client requests and Figure 56 shows the rate of process churn, being the sum of the number of terminated and created processes on average per second (during each snapshot interval). Note that this is the observed churn and there might have been short-lived processes that do not show up in the collected data.

The figure shows a churn rate of around four processes per second, which is relatively modest considering the high number of total processes. There are some observed short spikes higher, upwards to 30 processes per second, but a rate of around four processes should not be a problem with regards to performance overhead.

One motivation for terminating child processes is as noted above to avoid resource leaks due to bugs in system libraries. A limitation here is that Dante does not move active clients between processes (or allow termination of active *io* client sessions after a specified time), so *io* child processes cannot be terminated as long as they have at least one active client. This means that the *io* processes can essentially run forever if at least one of its clients never terminates its connection to the SOCKS server (and it is not terminated in another way).

The distribution of the actually observed process lifetimes is shown in Figure 57 and 99% of the observed processes have a lifetime of less than two minutes, meaning that they are relatively short-lived.

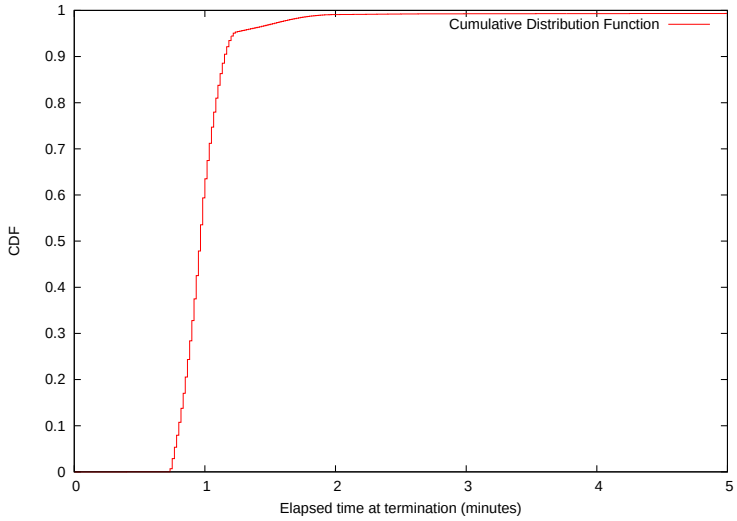

Figure 57: Observed Dante process lifetime CDF

There is however a long tail of longer-lived processes, shown in Figure 58. The longest-lived processes have a lifetime of almost ten days, corresponding to the traffic loss spike examined in Figure 8, at which all traffic was lost and most *io* processes were terminated as a result; most likely the highest process lifetime would be longer if this event had not occurred.

Overall, the observed process lifetimes would appear to often be somewhat short. There is obviously a tradeoff in the server between quickly adapting to load changes and the overhead from frequently creating and terminating processes, but the load observed would appear to be sufficiently stable for a process lifetime of less than two minutes to seem somewhat short. Possibly there is some room for handling this more efficiently at the expense of complicating the algorithms Dante uses to decide whether to terminate a process.

11.2 Client overhead

With multiple processes being used for handling clients, there is an obvious relationship between the number of clients and memory usage. Figure 59 shows a scatter plot of memory usage (VSZ) and the number of client slots and processes. This plot does not consider that a large portion of the memory is shared between the processes, meaning that the actually required memory is much smaller, but the relationships between the different values are still interesting as client processes might still have variations in memory usage.

The most obvious relationship is between the number of processes and memory usage. The data does not however form a single straight line; there is some variation, indicating that memory usage does not increase in an exact linear relation to the number of processes. Either some processes or process types see increases in memory, for

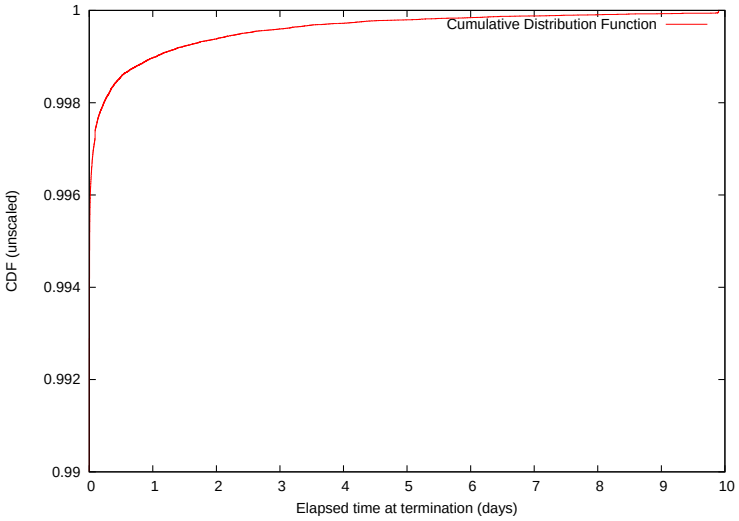

Figure 58: Observed Dante process lifetime CDF, upper range

example due to handling different numbers of clients, or the different process types have different memory requirements.

The total number of client slots is also closely related to the number of processes, and has an almost linear relationship with the amount of memory consumed. The total number of used client slots and active I/O slots also show fairly linear relationships with memory consumption; the overhead appears to increase with the load.

The memory actually consumed by each process, when shared memory is taken into consideration, is unfortunately not available, but the observable behavior is largely as expected, with the most interesting observation being that there is no abnormal or unexpected behavior visible.

A scatter plot showing the relationship between client slots and transfer rates is shown in Figure 60. Greatest variation is seen in the range between 7000 and 11,000 clients. The highest data rates are also observed in this range, with the peak slightly below a send rate of 0.9 Gigabit/s (in one direction). For higher numbers of clients only lower transfer rates were observed. While there might be many factors that can influence transfer rates, and there are fewer data points for the periods when there are more than 13,000 clients, it is possible to interpret the data as performance gradually falling off for higher number of clients. In the range 13,000 to 16,000 active *io* clients, the highest observed rate is around a send rate of 0.7 Gigabit/s (in one direction). However, lower rates are also observed with much lower numbers of clients, so the number of clients is clearly not the only factor affecting performance. Obviously at least one other factor must be the traffic pattern exhibited by the individual clients (e.g., interactive telnet-style traffic, or FTP-style traffic). Changes in network capacity over the links the traffic passes over between the Dante server and the machines it communicates with will obviously also affect transfer rates.

A similar plot for the packet receive and send rate is shown in Figure 61 and the

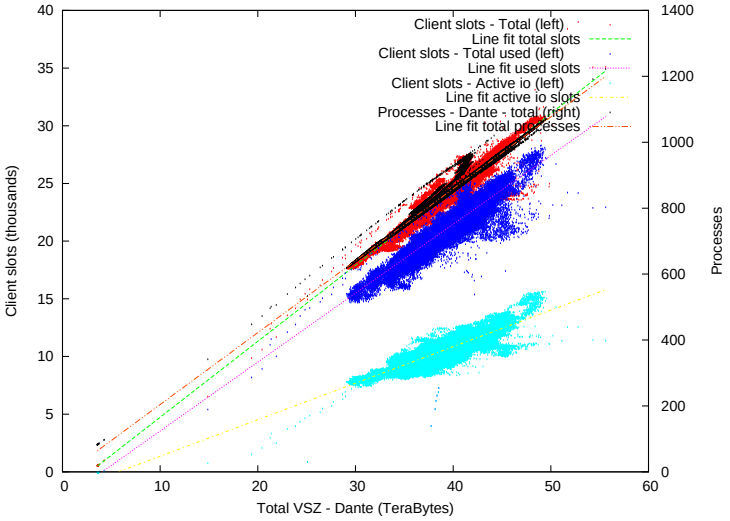

Figure 59: Dante client and process memory relationship

same relationship can be seen here. With between 7000 and 11,000 active *io* clients there is large variation in the packet rate, while for higher numbers of active clients the packet rate appears to gradually fall off.

The relationship between the total number of Dante processes and the packet rate is shown in Figure 62, and again similar behavior appears to be present. The highest transfer rates were measured with around 850 processes, and for higher number of processes, gradually lower packet rates were observed. Again, however, lower rates were observed with lower numbers of processes and there are fewer data points with many processes, but there appears to be an upper limit on transfer rates that gradually falls as the number of processes increases.

Figure 63 shows a scatter plot with the relationship between the number of active *io* process clients and the CPU time consumption per second, and also here the same trend can be seen; there is a peak in CPU usage at around 11,000 active *io* clients and, for the values available, lower CPU usage is seen for higher number of clients. This is despite the general trend, seen in the fitted line, being an increase in CPU usage along with an increase in the number of clients. The CPU usage being lower for high numbers of clients possibly indicates that the CPU in itself is not a limiting factor; less CPU time appears to be spent when there are many clients. There are fewer data points in this range, meaning that it is not possible to make any definite conclusions, but one possible explanation is that at this point the overhead from handling all the clients, either in the kernel, the machine hardware, NIC, or the network, limits the rate at which I/O can be performed, and for this reason results in less CPU usage.

A more general look at CPU usage on the machine is shown in Figure 64, and this figure shows that for high numbers of active *io* clients, the CPU idle time appears to increase. The change in system CPU is much lower than for user CPU. The exact reason for this behavior is unknown, but it does look like the CPU is idle around 10%

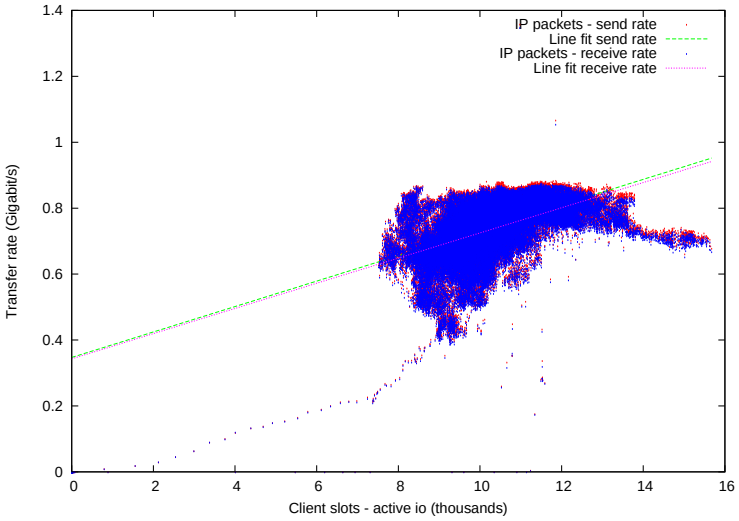

Figure 60: Dante client and transfer rate relationship

of the time when the number of clients is between $12{,}000$ and $14{,}000$. Whatever the bottleneck is at this point, it does not appear to directly be the CPU. Similar behavior is seen when looking at the number of processes (not shown).

A look at the division of client slots is also interesting, in that it shows how many clients can be found in the different stages of protocol processing.

An overview of the slot type ratios, showing the ratio of a given slot type to the total number of slots available at a given time, is given in Figure 65. Around 85% of the available slots are used, with most of these being *io* client slots. Around $1 - 2\%$ are *negotiate* processes and less than one percent are *request* slots. The number of free slots lies at around 15%. With the exception of some spikes, most values are fairly stable, despite variations in the load on the machine. As the main purpose of Dante is to transmit data, this is a good sign as it indicates that most clients are in the I/O state, and not busy with protocol negotiation or connection setup. If the majority of the slots had been in *negotiate* or *request* processes, this would indicate either that most client requests were very short (which is not necessarily a problem or an indication of something being wrong), there being a communication problem resulting in long SOCKS negotiations between Dante and the SOCKS clients, or a bottleneck in Dante itself as it transfers the SOCKS clients between the different processes and phases (*negotiate*, *request*, *io*).

Looking at the *io* client slots in more detail, the division of *io* client slot types is shown in Figure 66. Around 90% of the available slots are used, with around half of these being active I/O, and half being *in progress*, possibly indicating that there are many short-lived client sessions or failed *connect()* attempts. Around 10% of the available slots are free, meaning that there is a fairly good utilization of the available resources and room for new client requests.

As the main purpose of the Dante server (apart from access control) is to transfer

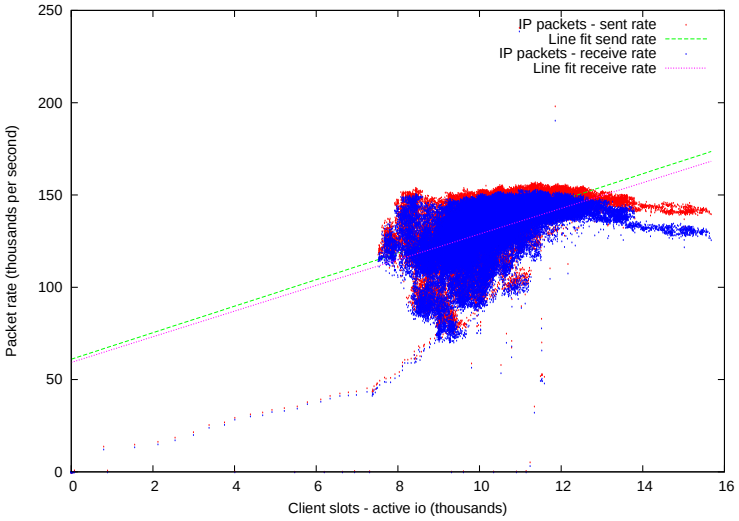

Figure 61: Dante client and packet rate relationship

data, it is interesting to look at the ratio of the total number of active *io* slots to the total number of Dante processes. This should give a rough idea of how many processes are required to handle a given number of active clients, for the client and traffic patterns observed at this particular production site.

Figure 67 shows this ratio, which lies between 10 and 14 during the majority of the time. The spike down at at the end of day 10 is clearly visible, indicating that this ratio changes when there is no load present. However, for the majority of the time, around one Dante process will be needed per $10 - 14$ active I/O clients. This number is somewhat lower than the total number of clients that can be handled in a single *io* process, which is 32 with the current default compile time value. The extra process overhead is likely to be mainly a result of the high number of client sessions that are *in progress*[3] in the *io* clients. If all used *io* slots are considered, also those *in-progress*, the number becomes a fairly stable 25, which is much closer to the 32 limit.

Finally, we look at the CPU time spent per *io* client, on average. Figure 68 shows both the number of active *io* client slots, and the amount of CPU time spent per client. The total number of active clients has a significant amount of variation, but the CPU-time spent per client is much more stable. There are some spikes in the data, but for the most part it would seem that the overhead per client is fairly stable, making it possible to estimate the capacity of the machine based on these values; a per-client CPU time of 0.001 second gives at most 16,000 clients before the CPU would definitely be a bottleneck. This assumes that the system scales linearly, which is likely not the case, but it is still interesting to see that the highest spike up in active *io* clients is up towards 16,000.

In summary, Dante uses a fixed number of clients per process, meaning that the number of processes increases with the client load. There are some indications that

[3] Waiting for the connection to the target server to complete.

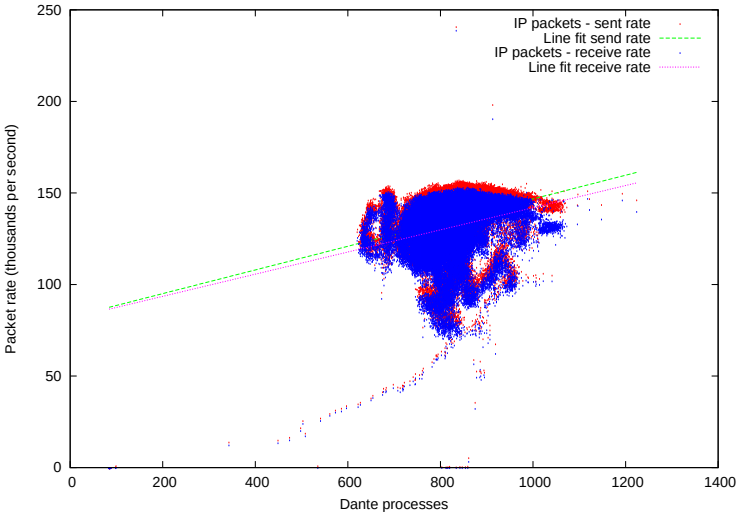

Figure 62: Dante process count and packet rate relationship

transfer rates gradually start decreasing with higher number of clients (and processes), but no definite correlation was found between the number of processes and the slightly lower transfer rates.

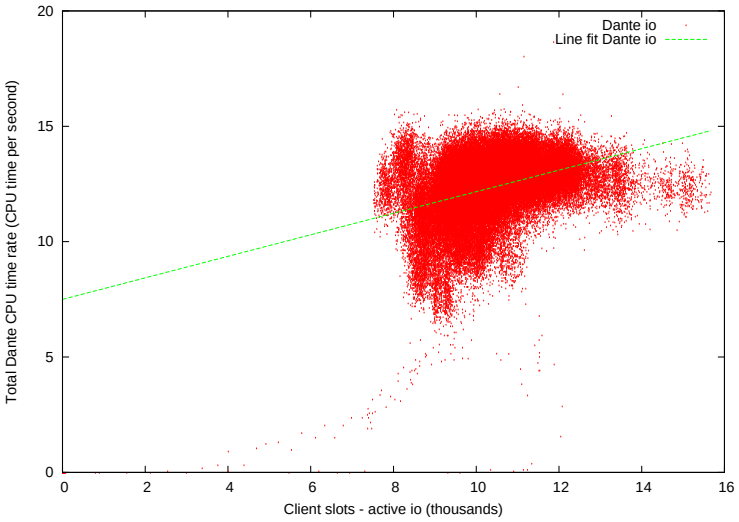

Figure 63: Dante *io* client and CPU usage relationship

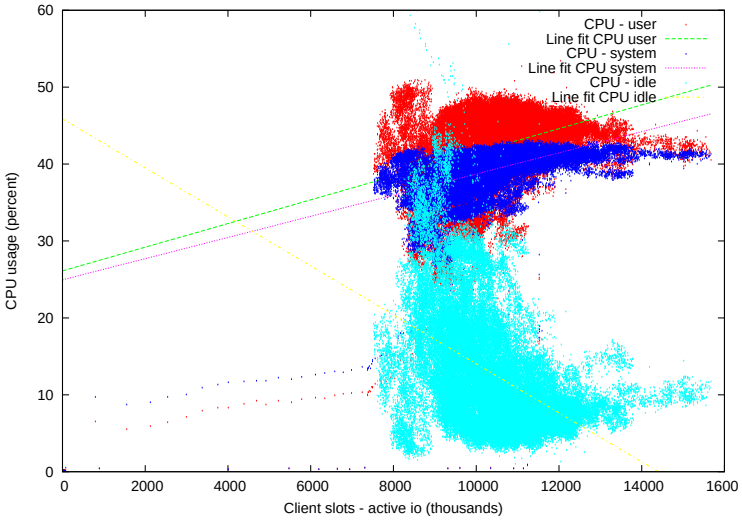

Figure 64: Dante I/O client and general CPU usage relationship

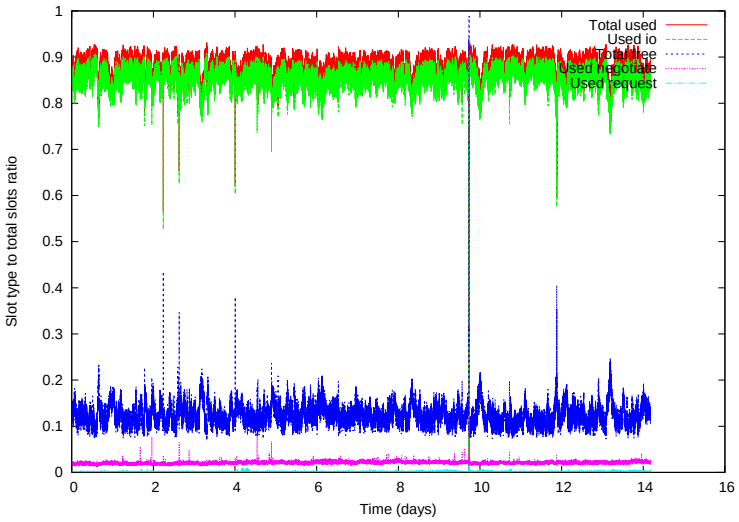

Figure 65: Slot type ratios, all child processes

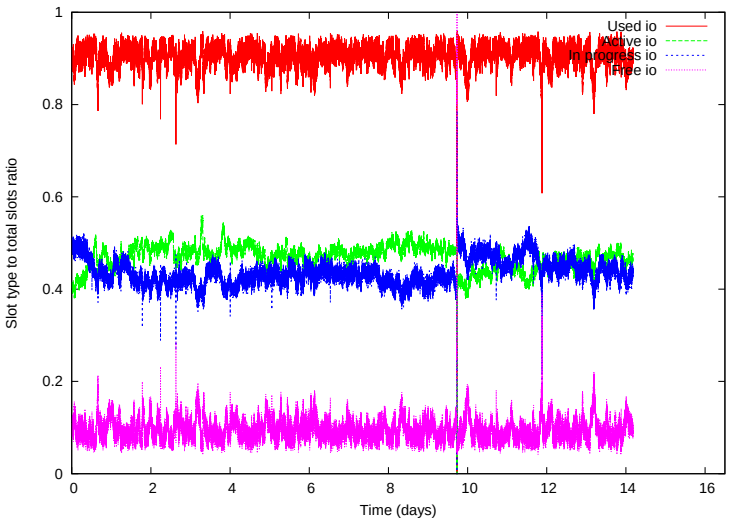

Figure 66: Slot type ratios, *io* processes

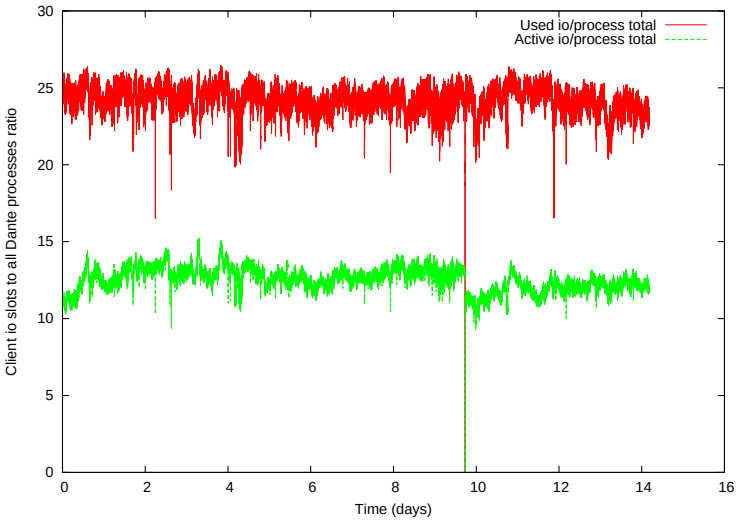

Figure 67: Slot type ratios, *io* slots to all Dante processes

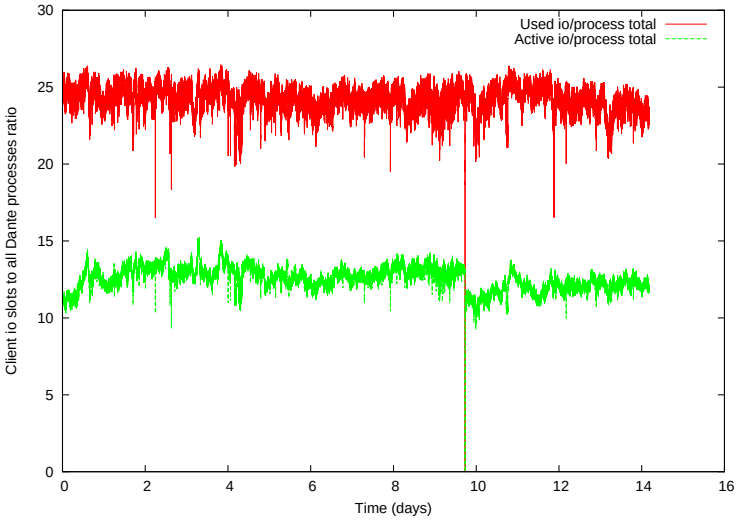

Figure 68: Slot type ratios, CPU time per active *io* client

12 Free slot management

A previous performance report [1] noted that a possibly higher than necessary number of free slots were available in version 1.3.1 of Dante. The number of free slots also varied quite significantly in that version.

This behavior now appears to be improved, with a smaller percentage of the total number of slots being unused. In version 1.3.1, the number of free slots was at times observed to be upwards to 50% of the total number of slots, while for version 1.4.0, this number lies fairly stable at around 15%, as seen in Figure 65.

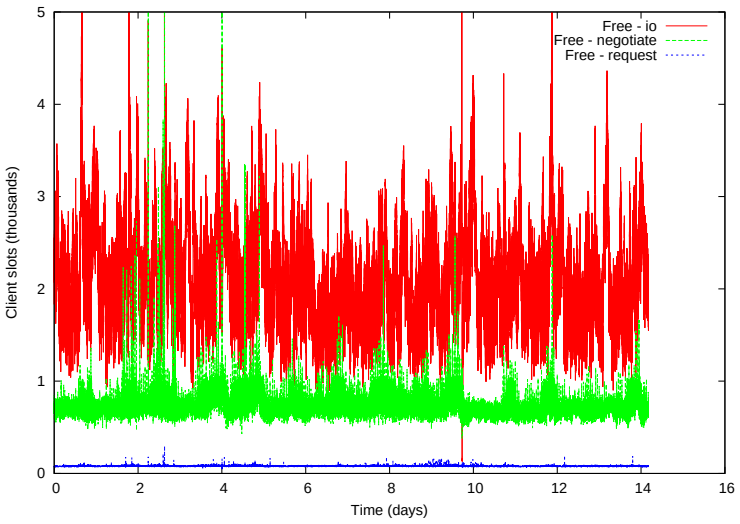

Figure 69: Free Dante client slots

The actual numbers of free slots for the different client process types is shown in Figure 69. While percentage wise they are not so large, they are still somewhat large in absolute numbers; the number of free *io* client slots vary between 1000 and 3000 for most of the time, while the number of free *negotiate* slots lies around $800 - 1000$. The number of free *request* slots is about 75.

These number might still seem a little high, but there is a tradeoff between being able to quickly serve a sudden burst of clients by having an adequate reserve of free client slots available, and having to suddenly create many new processes and having the clients wait for process creation and setup to complete. There is also the practical problem of adjusting slot numbers as it is only possible to terminate a process when it no longer has any clients.

Still, it would seem that the number of free slots could ideally be more similar between process types. As each client request must pass through each process type before it is completed, and not all requests can be expected to be completed successfully, it would seem logical that the highest number of free slots would be needed in the *negotiate* process, and fewest in the *io* processes. As noted above, practical difficulties make it difficult to quickly adjust the number of free slots to a desired number, but there

might still be some room for reducing resource consumption by having fewer free *io* slots available, to the extent that this is possible.

13 Summary and conclusions

This summary is split in three parts:

- Our conclusions regarding the data collection procedure.
- Our conclusions regarding the state and configuration of the machine on which Dante was running.
- Our conclusions regarding the behavior and performance of the Dante server.

13.1 Data collection and analysis procedure

A set of shell scripts running various system commands such as *ifconfig*, *ps* and *top* was used to collect information about Dante and the machine it was running on.

Overall, this procedure worked well, and it was possible to analyze and understand much of what occurred on the machine in the two week period that was examined, even if not all situations were equally simply to analyze.

The biggest limitation was that additional data that would have been useful was not available. For this analysis, this included the following:

- Per-protocol byte information (UDP/TCP).
- Per-application (Dante) byte information.
- TCP connection states.
- Aggregated Dante child process creation counts.
- Aggregated Dante client counts.
- Total terminated Dante child process used CPU time.

The first was partly compensated for by having packet counters for each protocol, but not having exact byte information available made it impossible to attempt to estimate, for example, UDP transfer rates, which would have been interesting when analyzing the relationship between UDP and CPU usage on the machine.

For Dante, interesting information is known by the Dante processes, but obtaining this information without significantly increasing the overhead from logging is currently not possible. The amount of information potentially obtainable via *setproctitle()* is also somewhat limited, so this is a problem it might be necessary to live with.

Adding kernel-based interfaces for collecting this information would solve many of these problems, but would make deploying the data collection scripts much more difficult, and also impose an added overhead on the kernel, with unknown consequences in general. On some platforms, tools such as *DTrace* might provide ways of storing information about processes as they terminate, but this would again impose additional overhead and increase storage requirements.

Minor tweaks and incremental improvements to the current approach, such as adding an overview of TCP connection states based on *netstat*, and using a more exact delay between collection intervals will likely suffice for the near future, as long as the limitations are know and considered during analysis.

13.2 Machine state and configuration

The machine running the Dante server examined in this document is a fairly powerful machine with many CPUs and a high amount of memory.

The machine was fairly highly loaded, with CPU usage of 90% across 16 CPUs and transfer rates of around 0.7 and 0.8 Gigabit/s in each direction, giving a total rate of data passing through the machine of around 1.5 Gigabit/s for much of the time. The machine generally appears to have handled this load well, with the exception of some shorter periods when it became overloaded and unable to handle scheduling in a timely manner. The machine also has a certain amount of dropped packets, likely due to periods of high system load.

Memory never appeared to be a problem; more than enough was available at all times and the machine never started swapping.

There is very little disk I/O being done on the machine, with most of the data being written to disk coming from the data script used to collect data for the performance analysis in this report.

13.3 Dante behavior

The Dante server on this machine handles a high load, with up towards $25{,}000$ active clients and traffic rates of around 1.5 Gigabit/s, including both TCP and UDP traffic.

The CPU usage of Dante lies at around $80 - 90\%$, with data forwarding by Dante being the main activity on the machine. Most of the Dante processes, and most of the CPU usage, comes from the Dante *io* child processes that handle forwarding of data between clients and their target peers. The number of Dante *io* processes vary between $600 - 800$. Active *io* client numbers vary between $17{,}500$ and $25{,}000$, with around $10{,}000$ clients in the *in progress* state, waiting for *connect()* to complete. Approximately 475 clients are active with SOCKS protocol negotiation and around 30 clients are in the connection setup *request* state.

The data set also provides several opportunities for looking at the behavior of Dante under interesting situations, such as a sudden full loss of network traffic. At the end of day 10 in the data set, all network traffic to the machine appears to stop, resulting in no data being forwarded. This state persists for around 15 minutes and causes the Dante server to reduce most of the resources it uses by terminating unused processes, likely as the TCP connections to the clients terminate. When the traffic returns, the Dante server reverses the process, by creating new processes to adapt to the suddenly returning high load.

The only unexpected observation made during this period of traffic loss was that as many as 71 *request* processes still remained unkilled, a much higher number than for the other child process types. This is something we will need to investigate further.

One additional observation made in relation to the period of traffic loss was that the overhead from handling UDP traffic appears to be more CPU intensive than TCP. TCP traffic in general appears to have a much lower effect on CPU usage than UDP traffic. The underlying reason for this is that much more work is required by a SOCKS server to process UDP requests compared to TCP requests, because SOCKS UDP headers must be be added or removed for each UDP packet, while TCP traffic can be forwarded directly as long as encryption is not enabled.

An issue partly related to this is that a large part of the overhead from a SOCKS server comes from data movement within the machine. Network data must be read from the network, copied to kernel memory, copied to application memory and then

pass back the other way. Around half of the CPU time spent on the machine comes from the system/kernel, and data copy operations likely make up a significant portion of this time. Copy elimination techniques such as *splice()* can likely aid in reducing this overhead. For UDP (or encrypted TCP traffic) using these types of techniques will be much more difficult due to the need for processing in the application, but for the common usage scenarios such as the one found on this machine, reducing the overhead from TCP will likely benefit also UDP by freeing up CPU time and memory bus capacity.

Another area in which improvements are possible is in the handling of client process termination and creation, especially with regards to the *negotiate* processes. There are indications that some periods of especially high load saw a high degree of possibly unnecessary process churn, resulting in a further increase in the system load when the system was already highly loaded. Possibly this is a result of already running child processes without any currently active clients being terminated only to a short time later be replaced by a new process of the same type. The decision to terminate inactive child processes is based on the number of requests handled and total runtime for each process, and if either is satisfied, processes are terminated despite there potentially being a high number of client requests. A slightly more complex algorithm, taking into consideration a few other additional factors, might be beneficial.

A related issue, handling of free client slots, appears to have improved compared to version 1.3.1 of Dante, with a much more stable number of free client slots, but also here there is possibly room for additional improvement as there appears to generally be a much higher number of free *io* client slots, compared to *negotiate* and *request* slots. This is despite clients first needing to progress through both the *negotiate* and *request* processes before needing an *io* process slot. While doing exact adjustments do the number of available slots is difficult, especially for *io* processes where there is no upper limit on how long client sessions can last, being able to reduce the number of processes further would free up unneeded resources.

A peak performance point related to the number of clients/processes and achievable packet rates was possibly observed on the machine, but no definitive conclusion could be made as to if this was due to the system overhead from handling the high number of connections and processes, or if it was because of some other reason, external or otherwise.

In summary, the Dante server is able to handle a very high load with tens of thousands of clients, at times, almost at saturated Gigabit link speed. It is also able to quickly adapt to changes in load. Still, there are some areas that can likely be improved for even better performance.

Feedback for this document can be sent to *misc-feedback@inet.no*.

References

[1] Inferno Nettverk A/S. Performance analysis of dante version 1.3.1. Technical Report 1, Inferno Nettverk A/S, `http://www.inet.no/dante/doc/1.3.x/dante_performance_1.3.1.pdf`, August 2011.