Tasting Client/Network/Server Pie

STUART KENDRICK

Stuart Kendrick works as a third-tier tech at the Fred Hutchinson Cancer Research Center in Seattle, where he dabbles in trouble-shooting, deep infrastructure design, and developing tools to monitor and manage devices. He started earning money as a geek in 1984, writing in FORTRAN on Cray-1s for Science Applications International Corporation, worked in desktop support, server support, and network support at Cornell University, and reached FHCRC in 1993. He has a BA in English, contributes to BRIITE (http://www.briite.org), and spends free time on yoga and CrossFit.
skendric@fhcrc.org

"The system is slow." How often do we hear those words? Here is one technique I use to start narrowing the fault domain, to better focus my attention on where the cause of the slowness resides. This approach relies on comparing packet traces taken near the client and near the server.

From a high level, I see three players contributing to the performance characteristics of an application: the client, the network, and the server. With this model in mind, I calculate how much time the client spends formulating requests, how much time the server spends formulating responses, and how much time the network spends flinging the packets back and forth. From there, I can represent the relative contribution of each component as a slice in a pie, which naturally focuses my attention on the largest slice (slowest component).

I will sketch three techniques (manual, semi-automated, mostly automated) for constructing the pie and illustrate how this approach can be useful for analyzing application performance issues.

Capture the Pie

The Campus Network (Figure 1) in this article consists of a classic access/distribution/core layer design, where the access layer functions at Layer 2 (k2-esx and s4-esx) and the distribution and core layers function at Layer 3. In this diagram I hide the distribution and core layers inside the cloud. In the pies below, the clients Europa and Deimos copy files across the Campus Network to the server Mars, while the probes Hale, Bopp, and Tempel capture the relevant traffic.

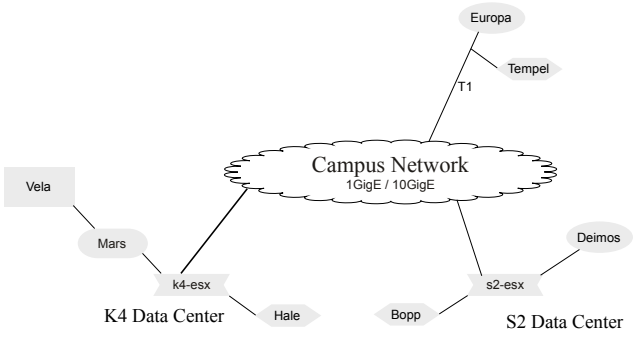

Figure 1: Campus Network

To make pies, I capture [1] two simultaneous packet traces of the experience from separate locations: one near the server (via the probe Hale), the other near the client (via the probe Tempel in the case of Europa, or via the probe Bopp in the case of Deimos). I filter the two traces, such that they include only traffic between client and server and only traffic relevant to the application I'm analyzing:

```
hale# dumpcap –i eth0 –w server.pcap –f "ip host europa and ip host mars"
```

Make the Pie from Scratch

This technique relies heavily on the DeltaT column in a trace. Here are the first handful of frames from a 10 MB SMB file copy from the client Europa to the server Mars across a T1 incurring 250 ms of latency.

Client-side trace:

No.	DeltaT	RelT	Bytes	Src	Dst	Info
1	0.000000	0.000000	70	Client	Server	TCP SYN
2	0.253718	0.253718	70	Server	Client	TCP SYN/ACK
3	0.000386	0.254104	64	Client	Server	TCP ACK
4	0.000007	0.254111	217	Client	Server	Negotiate Protocol Request
5	0.252704	0.506815	64	Server	Client	TCP ACK
6	0.256627	0.763442	226	Server	Client	Negotiate Protocol Response

Server-side trace:

No.	DeltaT	RelT	Bytes	Src	Dst	Info
1	0.000000	0.000000	70	Client	Server	TCP SYN
2	0.000059	0.000059	70	Server	Client	TCP SYN/ACK
3	0.254032	0.254091	64	Client	Server	TCP ACK
4	0.000272	0.254363	217	Client	Server	Negotiate Protocol Request
5	0.000410	0.254773	64	Server	Client	TCP ACK
6	0.257468	0.512241	226	Server	Client	Negotiate Protocol Response

The DeltaT (Delta Time) column records the time that has elapsed since the previous frame. The RelT (relative time, sometimes called cumulative time) column records time elapsed from the beginning of the trace through that frame.

Consider the trace taken close to the client. When the source address is the client, then DeltaT fairly closely represents the amount of time the client spent processing the previous message from the server. When the source address is the server, then DeltaT represents the time the server spent processing the client's request plus the time the network spent transmitting the server's response.

The converse holds for the trace taken close to the server. By using a little arithmetic, we can estimate the time the network contributed to the experience.

1. Filter the client-side trace so that we see only frames sourced from the client:

No.	DeltaT	RelT	Bytes	Src	Dst	Info
1	0.000000	0.000000	70	Client	Server	TCP SYN
3	0.000386	0.254104	64	Client	Server	TCP ACK
4	0.000007	0.254111	217	Client	Server	Negotiate Protocol request

2. Sum the DeltaT column to produce the client time estimate:

```
0.000000 + 0.000386 + 0.000007 = 0.000393s
```

3. Filter the server-side trace so that we see only frames sourced from the server:

```
No. DeltaT     RelT       Bytes  Src     Dst      Info

2   0.000059   0.000059    70    Server  Client   TCP SYN/ACK
5   0.000410   0.254773    64    Server  Client   TCP ACK
6   0.257468   0.512241   226    Server  Client   Negotiate Protocol Response
```

4. Sum the DeltaT column to produce the server time estimate:

```
0.000059 + 0.000410 + 0.257468 = 0.257937s
```

5. Estimate the network's contribution by grabbing the relative time from either trace and subtracting the client and server contributions:

```
Relative – (Client + Server) = Network
0.763442 – (0.000393 + 0.257937) = 0.505112s
```

6. Calculate the size of each slice in the CNS Pie:

```
Client Time   0.000393s
Server Time   0.257937s
Network Time  0.505112s
Total Time    0.763442s

Client%   = Client Time   / Relative Time = 0.000393 / 0.763442 = 0%
Server%   = Server Time   / Relative Time = 0.257937 / 0.763442 = 34%
Network%  = Network Time  / Relative Time = 0.505112 / 0.763442 = 66%
```

The skeptical reader may question why I plucked relative time from the client-side trace rather than from the server-side trace—in this trivial example, I agree that the choice makes a difference (using server-side RelT results in 50% server time and 50% network time, as opposed to the 66% and 34% produced above). However, I claim that the two will be identical, or nearly so, across large traces, and thus we can arbitrarily choose either one.

Naturally, my fingers become tired of punching buttons on a calculator, so I script [2] the process, invoking tshark (part of the Wireshark suite) to produce appropriately filtered text files containing just the summary lines, per above, then crawling through those text files while summing DeltaT. Attentive readers who examine the code will notice that I use a more laborious method for estimating the network contribution than the one sketched here.

As it turns out, once we chew [3] through all 10,776 frames in each of these traces, the results turn out as follows:

```
Client %    2.8s   / 63.5s =  4%
Server %    5.7s   / 63.5s =  9%
Network %  55s     / 63.5s =  87%
```

Use a Food Processor

Alternatively, tshark will perform the calculation for us [4], delivering numbers that are within a few percent of the ones I produce above using my home-grown code.

```
guru> tshark -nlr europa-to-mars-T1-250ms-at-europa.pcap –o
tcp.calculate_timestamps:TRUE -R "(tcp.dstport==445)"
-qz io,stat,600,"SUM(tcp.time_delta)tcp.time_delta"

================================================================
IO Statistics
Interval: 600.000 secs
Column #0: SUM(tcp.time_delta)tcp.time_delta
                  |     Column #0
Time              |     SUM
000.000-600.000         2.6
================================================================
guru>

guru> tshark -nlr europa-to-mars-T1-250ms-at-mars.pcap –o
tcp.calculate_timestamps:TRUE -R "(tcp.srcport==445)"
-qz io,stat,600,"SUM(tcp.time_delta)tcp.time_delta"

================================================================
IO Statistics
Interval: 600.000 secs
Column #0: SUM(tcp.time_delta)tcp.time_delta
                  |     Column #0
Time              |     SUM
000.000-600.000         6.0
================================================================
guru>
```

Capinfos, another Wireshark utility, tells us how long the trace lasted:

```
guru> capinfos europa-to-mars-T1-250ms-at-europa.pcap
File name: europa-to-mars-T1-250ms-at-europa.pcap
[…]
Capture duration: 63 seconds
[…]
```

Knowing that client time is 2.6 seconds, server time is 6.0 seconds, and total time is 63 seconds, we can calculate network time:

```
Network Time  = 63s – 2.6s – 6s = 53.4s
```

Calculate percentages:

```
Client %  2.6s   / 63s = 4%
Server %  6.0s   / 63s = 10%
Network %53.4s   / 63s = 85%
```

Finally, in Figure 2 (next page), we use our favorite charting program to produce the first pie.

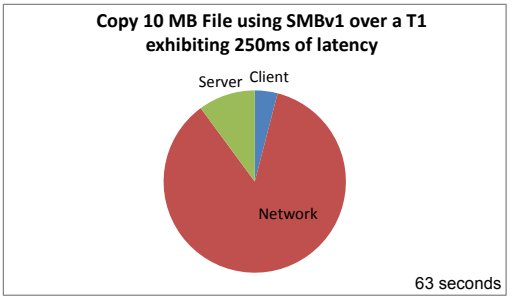

Figure 2: Our first pie

Buy the Pie from a Bakery

For those of us with money to spend, consider purchasing commercial software to provide a more sophisticated estimate of these three components. With these tools, we import the two traces into the analysis software, which then performs the tedious work described above. In Figure 3, I use Fluke Networks' ClearSight Analyzer to produce a stacked chart, functionally equivalent to a pie.

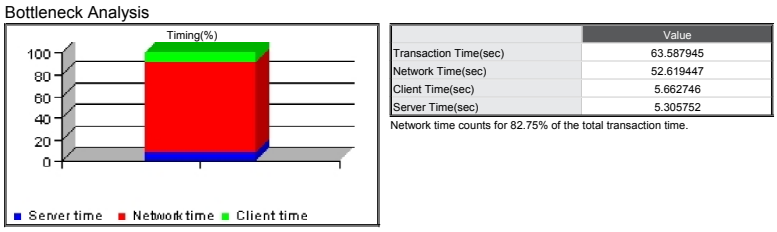

Figure 3: Bottleneck analysis chart from ClearSight Analyzer

More subtly, there are a range of issues which the home-baked or food-processed approaches miss, including TCP window size, TCP congestion window, application block size, packet loss, and parallel threads. As these factors arise in your situation, the manual approaches become increasingly inaccurate, and this is where the introspection baked into the commercial applications shines. Commercial packages also support importing more than two traces, captured at various points along the path between client and server, and are smart enough to track transactions through middleware (e.g., browser to Web server to back-end database).

Try a Slice

In these pies, Deimos copies files to Mars using NFSv3. Figure 4 illustrates situations in which I want to focus attention on the client, as it contributes 80–90% of the total transaction time.

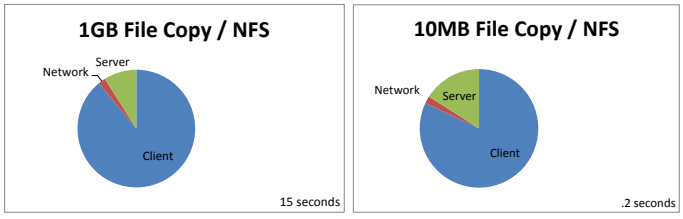

Figure 4: Copy big files across Campus Network

In Figure 5, I want to focus attention on the server, as it contributes 60–70% of the total time.

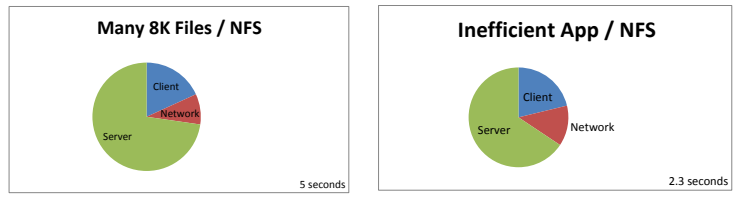

Figure 5: Copy small files across Campus Network

The Many 8K Files test involves copying a thousand 8K files, while the Inefficient App copies a single 1K file one byte at a time, closing and re-opening the file between each byte:

```
#!/usr/bin/perl
# Inefficient application
[…]
$destination = '/mnt/server';
$source = '/var/tmp/test_file';
open $read_fh, '<', $source;  # Open source file
while (read $read_fh, my $tmp, 1) { # Read next byte from the source file
       open $write_fh, '>>', $destination; # Open destination file
       print {$write_fh} $tmp;  # Write this byte to destination file
       close $write_fh;   # Close destination file
}
```

For the one repository I analyzed during this job, Inefficient App turned out to be a dead-ringer for Subversion, making it useful for modeling Subversion behavior (see Figure 6) when testing new client / network / server combinations.

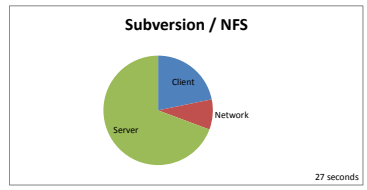

Figure 6: Subversion across Campus Network

Direct Our Attention

The pies suggest that for large file copies—a streaming application—we direct our attention toward Deimos. On investigation, we might find that beefing up its CPU or giving it a faster drive reduces the total transaction time.

For transactional applications, such as copying many small files or Subversion, the pies suggest that we direct our attention toward Mars. On investigation, we find that Mars is backed by the mass storage device Vela, which contains ~500 spindles of 1 TB and 2 TB 10 K SATA drives working in parallel. Vela divides each disk into ~700 MB chunklets, picks at least one chunklet from each disk, and glues them together to produce the LUN which Mars exposes to Deimos.

As a result, Mars performs well for streaming applications and poorly for transactional applications. Why? When we copy a big file to Mars, 500 spindles work together to swallow the datastream: the single spindle inside Deimos cannot keep up, making Deimos the primary contributor to the pie. When we copy many small files one at a time, the two systems are more evenly matched: only a single spindle inside Vela handles each write request, and that request must complete before Deimos can forward the next request.

For Mars, we might experiment with adding a LUN serviced by small, fast spindles—say, a dozen 15 K 250 MB drives—and moving Subversion to a volume hosted on that LUN. These platters are small and they rotate rapidly, reducing seek latency. For transactional applications, we might predict that such a LUN would deliver faster performance.

The CNS Pie provides a visually intuitive tool for narrowing the fault domain and for communicating the contours of the issue to our colleagues [5].

Too Much Sugar

Yes, I've been oversimplifying. Sure, sometimes the CNS Pie accurately directs our attention to the bottleneck. But the real world can be complex, and there are plenty of times when the CNS Pie misdirects our attention.

In Figure 7, both pies suggest that we focus on the network in order to improve performance, but notice how the transaction time drops from 64 seconds to 14 seconds when we upgrade from SMBv1 to v2.

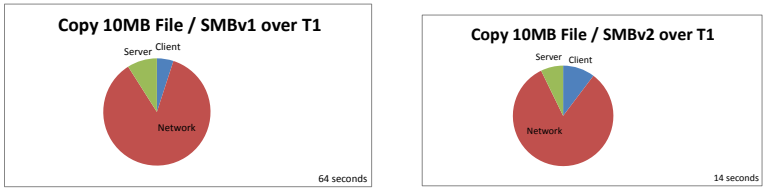

Figure 7: SMBv1 vs. SMBv2 over T1 with 250 ms latency

A simplistic reading of the left-hand CNS Pie would have focused our attention on the network, which might have pushed us to purchase a network pipe with more throughput. This would not have helped: SMBv1's *application block size* is 61 K, and so once latency reaches ~40 ms, no amount of fatter pipe will improve performance [6]. We could have purchased a 10 GigE network service (assuming latency remained the same) without improving total time. On the other hand, by upgrading the client and server to run SMBv2 (available in Windows Vista+ and Samba 3.5+), we improved performance by a factor of ~5. This new version of SMB auto-tunes its application block size and streamlines metadata operations, thus improving performance.

While I find the CNS Pie useful in shedding light on application performance issues, it remains only one tool in the toolkit: there are no silver bullets.

Appreciation

I feel particular gratitude to Mike Pennacchi of Network Protocol Specialists for teaching me to make my first CNS Pie. (As far as I can tell, Mike invented the

CNS Pie back in the 1990s. If you know of prior art, please drop me a note.) I also appreciate the professionals who have given their time to coach me on the topics covered in this article: Glenn Boyle of BT Global Services for opening my eyes to how I could bake at home, for refining my recipes, for teaching me how to use a food processor, and for helping me understand numerous subtleties; Gary Kaiser of Compuware for help understanding yet more subtleties plus the sophistication which commercial products bring to this space; and my colleagues Robert McDermott, for teaching me about the complex world of storage systems, and Wolfe Maykut, for the Inefficient App.

Thank you also to the community active on the LinkedIn Protocol Analysis and Troubleshooting group—I appreciate your contributions to the rich discussions there.

References

[1] I use dumpcap, tcpdump, and tshark interchangeably, depending on mood—their syntax is almost identical. For those interested in high-performance capture, check out Corey Satten's gulp: http://staff.washington.edu/corey/gulp.

[2] Data mangling code is available at http://www.skendric.com/app/code/extract-summary-lines-from-pcap and http://www.skendric.com/app/code/calculate-cns-pie.

[3] For a detailed description of this process, see http://www.skendric.com/app/make-cns-pie/Make-Client-Network-Server-Pie.pdf.

[4] Requires Wireshark 1.7.1 or later.

[5] For more examples of how the CNS Pie can direct our attention, see the "Make Client/Network/Server Pie" article at http://www.skendric.com/app.

[6] Bandwidth Delay Product calculation: 1,544,000 b/s * .04s = 61,760 bits =~ 61K.