An Engineer's Guide to Bandwidth (Yahoo! Developer Network Blog)

October 1, 2009

Web app developers spend most of our time not thinking about how data is actually transmitted through the bowels of the network stack. Abstractions at the application layer let us pretend that networks read and write whole messages as smooth streams of bytes. Generally this is a good thing. But knowing what's going underneath is crucial to performance tuning and application design. The character of our users' internet connections is changing and some of the rules of thumb we rely on may need to be revised.

In reality, the Internet is more like a giant cascading multiplayer game of **pachinko**. You pour some balls in, they bounce around, lights flash and — usually— they come out in the right order on the other side of the world.

What we talk about, when we talk about bandwidth

It's common to talk about network connections solely in terms of "bandwidth". Users are segmented into the high-bandwidth who get the best experience, and low-bandwidth users in the backwoods. We hope some day everyone will be high-bandwidth and we won't have to worry about it anymore.

That mental shorthand served when users had reasonably consistent wired connections and their computers ran one application at a time. But it's like talking only about the top speed of a car or the MHz of a computer. Latency and asymmetry matter at least as much as the notional bits-per-second and I argue that they are becoming even more important. The quality of the "last mile" of network between users and the backbone is in some ways getting worse as people ditch their copper wires for shared wifi and mobile towers, and clog their uplinks with video chat.

It's a rough world out there, and we need to to a better job of thinking about and testing under realistic network conditions. A better mental model of bandwidth should include:

- packets-per-second
- packet latency
- upstream vs downstream

Packets, not bytes

The quantum of internet transmission is not the bit or the byte, it's the *packet*. Everything that happens on the 'net happens as discrete pachinko balls of regular sizes. A message of N bytes is chopped into ceil(N / 1460) packets [**1**] which are then sent willy-nilly. That means there is little to no difference between sending 1 byte or 1,000. It also means that sending 1,461 bytes is *twice* the work of sending 1,460: two packets have to be sent, received, reassembled, and acknowledged.

Packet #1 Payload

```
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
............................................................
```

Packet #2 Payload

```
.
```

Listing 0: Byte 1,461 aka The Byte of Doom

Crossing the packet line in HTTP is very easy to do without knowing it. Suppose your application uses a third-party web analytics library which, like most analytics libraries, stores a big hunk of data about the user inside long-lived cookie tied to your domain. Suppose you also stuff a little bit of data into the cookie too. This cookie data is thereafter echoed back to your web server upon each request. The boilerplate HTTP headers (Accept, User-agent, etc) sent by every modern browser take up a few hundred more bytes. Add in the actual URL, Referer header, query parameters... and you're dead. There is also the **little-known fact** that browsers **split certain POST requests** into at least two packets regardless of the size of the message.

One packet, more or less, who cares? For one, none of your fancy caching and CDNs can help the client send data upstream. **TCP slow-start** means that the client will wait for acknowledgement of the first packet before sending the second. And as we'll see below, that extra packet can make a large difference in the responsiveness of your app when it's compounded by latency and narrow upstream connections.

Packet Latency

Packet latency is the time it takes a packet to wind through the wires and hops between points A and B. It is roughly a function of the physical distance (at **2/3 of the speed of light**) plus the time the packet spends queued up inside various network devices along the way. A typical packet sent on the 'net backbone between San Francisco and New York will take about 60 milliseconds. But the latency of a user's last-mile internet connection can vary enormously [**2**]. Maybe it's a hot day and their router is running slowly. The EDGE mobile network has a best-case latency of 150msec and a real-world average of 500msec. There is a semi-famous **rant from 1996** complaining about *100*msec latency from substandard telephone modems. If only.

Packet loss

Packet loss manifests as packet latency. The odds are decent that a couple packets that made up the copy of this article you are reading got lost along the way. Maybe they had a collision, maybe they stopped to have a beer and forgot.

The sending end then has to notice that a packet has not been acknowledged and re-transmit.

Wireless home networks are becoming the norm and they are unfortunately very susceptible to interference from devices sitting on the 2.4GHz band, like microwaves and baby monitors. They are also notorious for cross-vendor incompatibilities. Another dirty secret is that consumer-grade wifi devices you'll find in cafés and small offices don't do **traffic shaping**. All it takes is one user watching a video to flood the uplink.

Upstream < Downstream

Internet providers lie. That "6 Megabit" cable internet connection is actually 6mbps down and 1mbps up. The bandwidth reserved for upstream transmission is often 20% or less of the total available. This was an almost defensible thing to do until users started file sharing, VOIPing, video chatting, etc en masse. Even though users still pull more information down than they send up, the asymmetry of their connections means that the upstream is a chokepoint that will probably get worse for a long time.

A Dismal Testing Harness

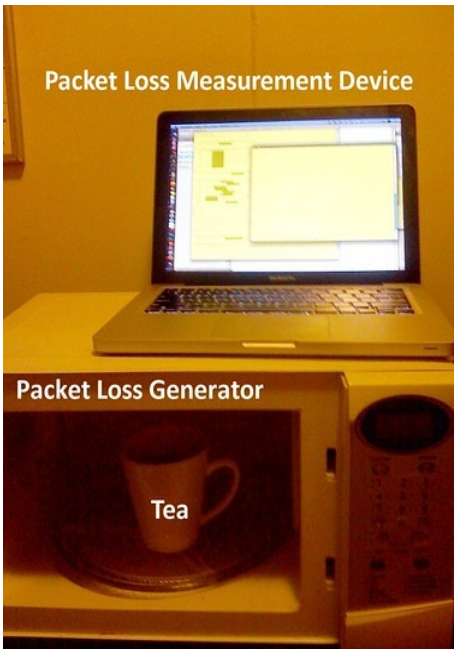

Figure 0: It's popcorn for dinner tonight, my love. I'm doing science!

We need a way to simulate high latency, variable latency, limited packet rate, and packet loss. In the olden days a good way to test the performance of a system through a bad connection was to configure the switch port to run at half-duplex. Sometimes we even did such testing on purpose. :) **Tor** is pretty good for simulating a crappy connection but it only works for publicly-accessible sites. Microwave ovens consistently cause packet loss (my parents' old monster kills wifi at 20 paces) but it's a waste of electricity.

The **ipfw** on Mac and FreeBSD comes in handy for local testing. The command below will approximate an iPhone on the EDGE network with a 350kbit/sec throttle, 5% packet loss rate and 500msecs latency. Use sudo ipfw flush to deactivate the rules when you are done.

```
$ sudo ipfw pipe 1 config bw 350kbit/s plr 0.05 delay 500ms

$ sudo ipfw add pipe 1 dst-port http
```

Here's another that will randomly drop half of all DNS requests. Have fun with that one.

```
$ sudo ipfw pipe 2 config plr 0.5

$ sudo ipfw add pipe 2 dst-port 53
```

To measure the effects of latency and packet loss I chose a **highly-cached 130KB file** from Yahoo's servers. I ran a script to download it as many times as possible in 5 minutes under various ipfw rules [**3**]. The "baseline" runs were the control with no ipfw restrictions or interference.

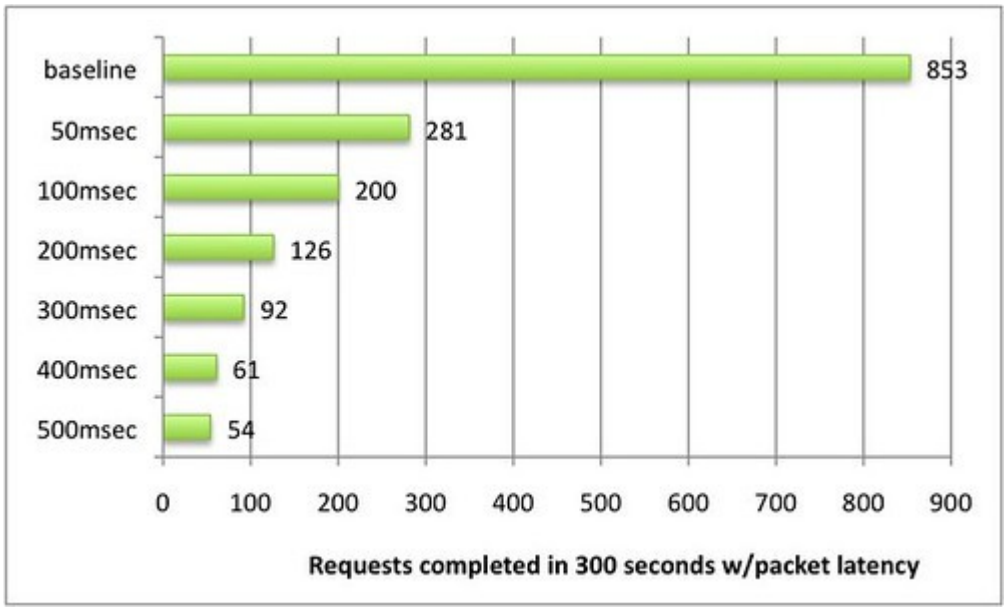

Figure 1: The effect of packet latency on download speed

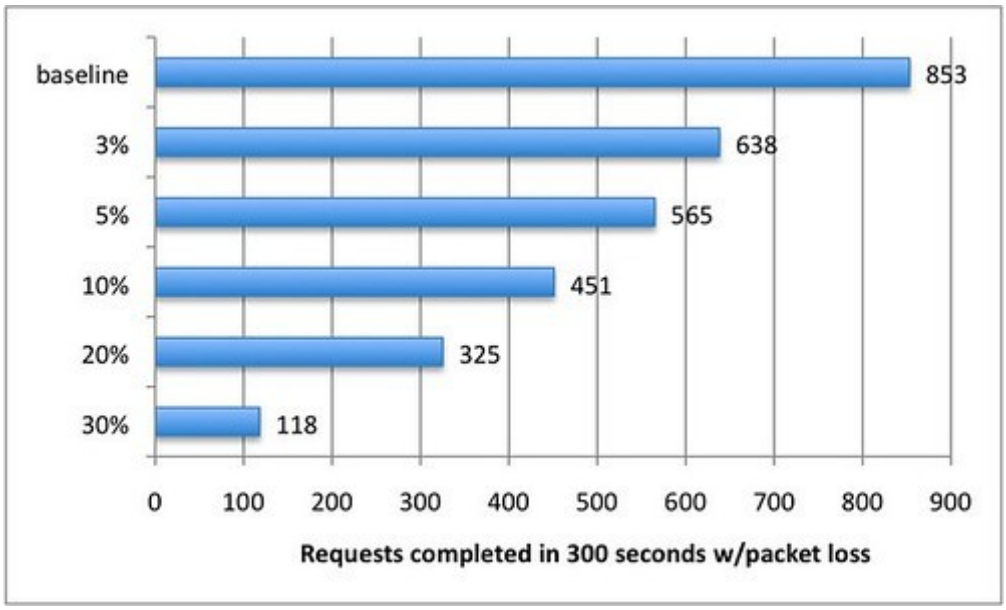

Figure 2: Effect of packet loss on download speed

Just 100 milliseconds of packet latency is enough to cause a smallish file to download in an average of 1500 milliseconds instead of 350 milliseconds. And that's not the worst part: the individual download times ranged from 1,000 to 3,000 milliseconds. Software that's consistently slow can be endured. Software that halts for no obvious reason is maddening.

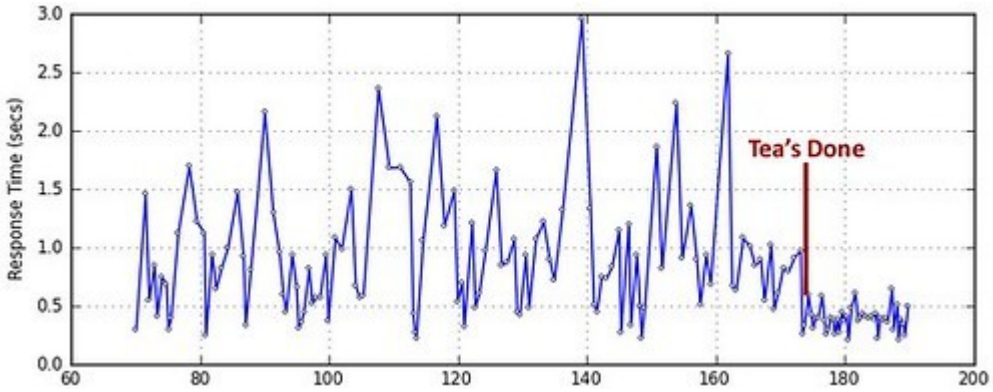

Figure 3: Extreme volatility of response times during packet loss.

So, latency sucks. Now what?

Yahoo's **web performance guidelines** are still the most complete resource around, and backed up by real-world data. The key advice is to reduce the number of HTTP requests, reduce the amount of data sent, and to order requests in ways that use the observed behavior of browsers to best effect. However there is a simplification which buckets users into high/low/mobile categories. This doesn't necessarily address poor-quality bandwidth across all classes of user. The user's connection quality is often very bad and getting worse, which changes the calculus of what techniques to employ. In particular we should also take into account that:

- Upstream packets are almost always expensive.
- Any client can have high or low overall bandwidth.
- High latency is not an error condition, it's a fact of life.
- TCP connections and DNS lookups are expensive under high latency.
- Variable latency is in some ways worse than low bandwidth.

Assuming that a large but unknown percentage of your users labor under adverse network conditions, here are some things you can do:

- To keep your user's HTTP requests down to one packet, stay within a budget of about 800 bytes for cookies and URLs. Note that every byte of the URL counts twice: once for the URL and once for the Referer header on subsequent clicks. An interesting technique is to store app state that doesn't need to go to the server in fragment identifiers instead of query string parameters, e.g. /blah#foo=bar instead of /blah?foo=bar. Nothing after the # mark is sent to the server.

•If your app sends largish amounts of data upstream (excluding images, which are already compressed), consider implementing client-side compression. It's possible to get 1.5:1 compression with a simple LZW+Base64 function; if you're willing to monkey with ActionScript you could probably do **real gzip compression**.

•YSlow says you should **flush() early** and put **Javascript at the bottom**. The reasoning is sound: get the HTML <head> portion out as quickly as possible so the browser can start downloading any referenced stylesheets and images. On the other hand, JS is supposed to go on the bottom because script tags halt parallel downloads. The trouble comes when your page arrives in pieces over a long period of time: the HTML and CSS are mostly there, maybe some images, but the JS is lost in the ether. That means the application may look like it's ready to go but actually isn't — the click handlers and logic and ajax includes haven't arrived yet.

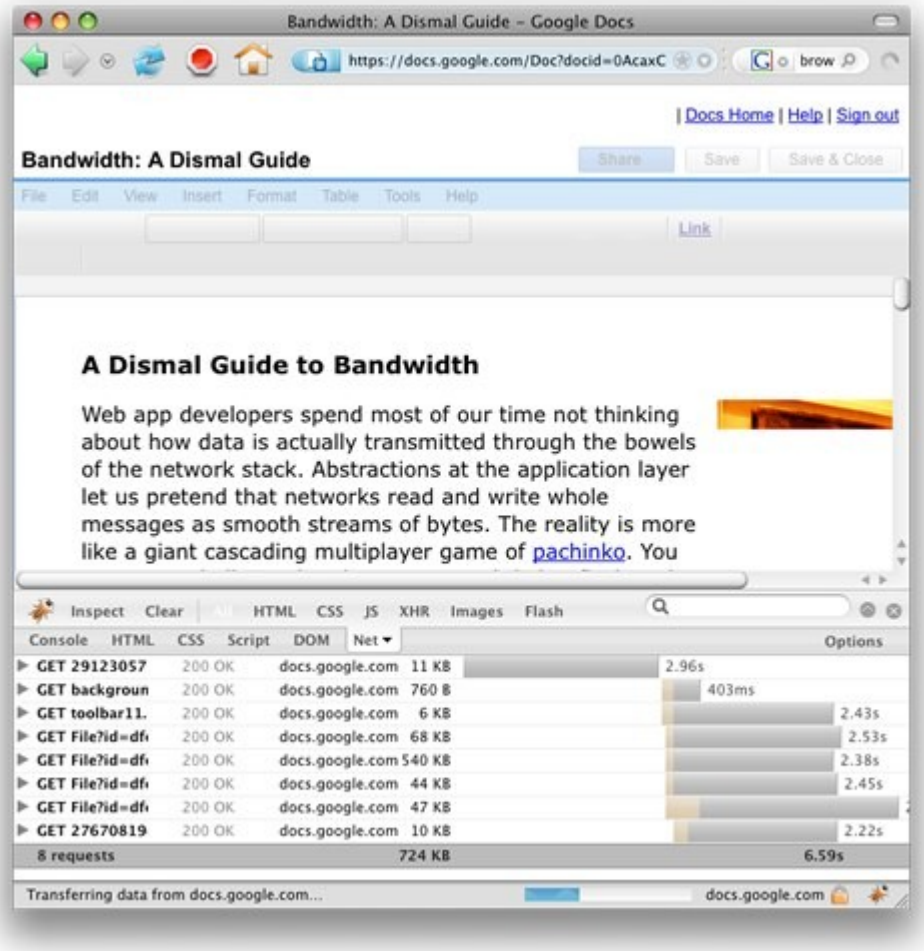

Figure 4: docs is loading slowly... dare I click?

Maybe in addition to the CSS/HTML/Javascript sandwich you could stuff a minimal version of the UI into the first 1-3KB, which gets replaced by the full version. Google Docs presents document contents as quickly as possible but disables the buttons until its sanity checks pass. Yahoo's home page does something similar.

This won't do for heavier applications, or those that don't have a lot of passive text to distract the user with while frantic work happens offstage. Gmail compromises with a loading screen which times out after X seconds. On timeout it asks the user to choose whether to reload or use their lite version.

•Have a plan for disaster: what should happen when one of your scripts or styles or data blobs never arrives? Worse, what if the user's cached copy is corrupted? How do you detect it? Do you retry or fail? A quick win might be to add a checksum/eval step to your javascript and stylesheets.
•We also recommend that you should make as much CSS and Javascript as possible external and to **parallelize HTTP requests**. But is it wise to do more DNS lookups and open new TCP connections under very high latency? If each new connection takes a couple seconds to establish, it may be better to inline as much as possible.
•The trick is how to decide that an arbitrary user is suffering high latency. For mobile users you can pretty much take high latency as a given [**4**]. Armed with per-IP statistics on client network latency from bullet #4 above, you can build a lookup table of high-latency subnets and handle requests from those subnets differently. For example if your servers are in Seattle it's a good bet that clients in the 200.0.0.0/8 subnet will be slow. 200.* is for Brasil but the point is that you don't need to know it's for Brasil or iPhone or whatever — you're just acting on observed behavior. Handling individual users from "fast" subnets who happen to have high latency is a taller order. It may be possible to get information from the socket layer about how long it took to establish the initial connection. I don't know the answer yet but there is promising research here and there.

•A good technique that seems to go in and out of fashion is KeepAlive. Modern high-end load balancers will try to keep the TCP connection alive between themselves and the client, no matter what, while also honoring whatever KeepAlive behavior the webserver asks for. This saves expensive TCP connection setup and teardown without tying up expensive webserver

> processes (the reason why some people turn it off). There's no reason why you couldn't do the same with a software load balancer / proxy like Varnish.

This article is the first in a series and part of ongoing research on bandwidth and web app performance. It's still early in our research, but we chose to share what we've found early so you can join us on our journey of discovery. Next, we will dig deeper into some of the open questions we've posed, examine real-world performance in the face of high latency and packet loss, and suggest more techniques on how to make your apps work better in adverse conditions based on the data we collect.

Carlos Bueno
Software Engineer, Yahoo! Mail