6.896 Lecture 8: HTTP, HTTP/TCP Interactions

Hari Balakrishnan

September 30, 1998

1 Outline

- Announcements
- **Readings: PM95, N+97**
- HTTP overview
- Problems with HTTP/1.0
- Solutions: persistent connections, pipelining; HTTP/1.1
- Problems with HTTP/1.1 and solutions

2 Announcements

- Project proposals due by 5pm (either in room 305 or get it to me between 2 and 3 in 510).
- Extension to Wednesday *if* you need it.
- Will get back to you by email with comments.
- Next week: unicast routing.
- Fair queueing recap and problems.

3 HTTP overview

- Client establishes connection with server, typically on port 80; server sends response to client.
- First get index, which has list of embedded objects. Then, send requests for inlines.
- HTTP/0.9 and 1.0: separate TCP connection per request.
- Format of HTTP request message is: request line, headers, blank-line, body of message. Three request types: GET, HEAD, POST.
- Format of response message is: status-line, headers, blank-line, body. Status-line has a *response code*.
- Distinguishing characteristic: multiple, often-short data items. In one data set, median of 1946 bytes and mean of 13767 bytes. Q: why is the median smaller by so much?

3.1 Problems with original HTTP/1.0

- One TCP connection per request with no sharing.
- Has advantages: simple to implement, transfers are independent.
- Disadvantages:
  - Multiple connection setups and extra round-trips per download. Draw picture of three-way handshake.
  - Multiple slow start phases
  - *Slow user performance!*
- More disadvantages:
  - Short transfers makes TCP adaptation hard: often stuck in slow start. Can't recover from losses promptly.
  - Lots of open connections increases server state.
  - But worse, lots of connections in the recent past increases state by an order of magnitude or more. Why? TIME_WAIT connections. Q: why do we need TIME_WAIT?
  - Slows down response if we need to authenticate per request.
- Netscape's solution to improve user response time: *multiple concurrent connections.*
- This is absolutely awful for the network! In fact, this isn't the best way of getting quick response time either in a lossy network. Q: Why?

4 Solution: make connections longer!

- Padmanabhan & Mogul (and Spero, independently) analyzed these problems in 1994-95. They proposed two (actually three) solutions: persistent connections, pipelining, and prefetching. The first two have been adopted by HTTP/1.1 (with minor mods).
- Idea: make connection durations longer by multiplexing independent transfers on one TCP connection. This sounds easy enough to do. But there are some issues to be resolved.
- Issue #1: How to demux?
- Option #1: Add a multiplexing header to each packet. This isn't compatible with older versions and isn't as easily deployable.
- Option #2: Serialize transmissions. But how do we know when one item ends and the next begins? Problem even if client and server operate "synchronously" (i.e. if client makes next request only after previous response). Q: BTW, why is this not a problem with HTTP/1.0?
  - Simple: send content length up-front. But what if you don't know length a priori?
  - Boundary delimiter: like in telnet. But this needs byte-by-byte processing. Inefficient. (Q: why is this OK for telnet?)
  - Block-based transmissions with length per block: OK, but needs data copying at server. Also needs protocol change.

- Store-and-forward: cp data temp; wc temp; etc. Again, inefficient.
- PM95's solution: If you can use content length or boundary delimiter, great. Else, just close. Essentially, they punted on the problem. (Probably the right thing to do).

- Issue #2: when can server or client terminate connection? Server can't close both sides of connection arbitrarily.
- Persistent connections are great for network and somewhat OK for client, but awful for response time.
- Pipelining requests addresses this. GETALL gets all embedded stuff. More useful is GETLIST, to take cached items into account.
- Note: in HTTP/1.1, two concurrent connections can occur (one for long dynamically generated document, so you need the other for embedded objects. HTTP/1.1 also does pipelining.
- Prefetching: client-driven option. Not widely adopted. (Q: Why not?)
- Performance: measured network retrieval times because they had a slow machine. Q: how might they have done this?
- Benefits best for small sizes (of course). Up to 2X performance improvement in response time compared to old HTTP protocol (no concurrent connections).
- Server resource utilization reduces heavily (Mogul, SIGCOMM 95).
- Longer connections permit better TCP adaptation to available bandwidth and better loss recovery.
- Other issues (from N+97):
  - Interactions with Nagle algorithm and SWS (Heidemann 97, verified in N+97).
  - Range requests and cache validations reduce amount of data sent.
  - Compression is good. Of course, since it reduces amount of data transmitted on network.
  - CSS tends to reduce number of embedded objects.
  - PNG and MNG produce smaller images. This is also good.

5 Problems with HTTP/1.1

- An application-specific solution to a transport-level problem.
- Assumes that HTTP will be the *only* application-level transport. Already, there are other examples cropping up.
- Persistent connections in HTTP/1.1 lead to losses on one object stalling subsequent ones, thus destroying user-perceived parallelism.
- Of course, independent concurrent connections fix the last problem, but are awful for the network (not to mention multiple slow starts and excessive timeouts).
- Solution: TCP-INT (BP+98, Padmanabhan98) or TCP control-block interdependence (Touch97). Share congestion and loss recovery state, but preserve logical independence of transfers. Draw picture.

- Works well: better congestion management, better loss recovery, avoids coupling problems of persistent connections.
- This is still not enough! User control of bandwidth allocation; transport-independence is getting to be important (importance of non-TCP apps).