Overview

- HTTP Basics
- HTTP Fixes
- ALF
- Congestion Manager

HTTP Basics

- HTTP layered over bidirectional byte stream
- Almost always TCP
- Interaction
  - Client sends request to server, followed by response from server to client
  - Requests/responses are encoded in text
- How to mark end of message?
  - Size of message → Content-Length
  - Must know size of transfer in advance
  - Delimiter → MIME style Content-Type
    - Server must "byte-stuff"
  - Close connection
    - Only server can do this

HTTP Request

- Request line
  - Method
    - GET – return URI
    - HEAD – return headers only of GET response
    - POST – send data to the server (forms, etc.)
  - URI
    - E.g. http://www.seshan.org/index.html with a proxy
    - E.g. /index.html if no proxy
  - HTTP version
- Request headers
  - Authorization – authentication info
  - Acceptable document types/encodings
  - From – user email
  - If-Modified-Since
  - Referer – what caused this page to be requested
  - User-Agent – client software
- Blank-line
- Body
HTTP Request Example
GET / HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Host: www.seshan.org
Connection: Keep-Alive

HTTP Response
• Status-line
• HTTP version
• 3 digit response code
  • 1XX – informational
  • 2XX – success
  • 3XX – redirection
  • 4XX – client error
  • 5XX – server error
• Reason phrase

HTTP Response
• Headers
  • Location – for redirection
  • Server – server software
  • WWW-Authenticate – request for authentication
  • Allow – list of methods supported (get, head, etc)
  • Content-Encoding – E.g x-gzip
  • Content-Length
  • Content-Type
  • Expires
  • Last-Modified
• Blank-line
• Body

HTTP Response Example
HTTP/1.1 200 OK
Date: Tue, 27 Mar 2001 03:49:38 GMT
Server: Apache/1.3.14 (Unix) (Red-Hat/Linux) mod_ssl/2.7.1
  OpenSSL/0.9.5a DAV/1.0.2 PHP/4.0.1pl2 mod_perl/1.24
Last-Modified: Mon, 29 Jan 2001 17:54:18 GMT
ETag: "7a11f-10ed-3a75ae4a"
Accept-Ranges: bytes
Content-Length: 4333
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html

Typical Workload
• Multiple (typically small) objects per page
• Request sizes
  • In this paper → Median 1946 bytes, mean 13767 bytes
  • Why such a difference? Heavy-tailed distribution
    • Pareto – \( p(x) = ax^{-(a+1)} \)
• File sizes
  • Why different than request sizes?
  • Also heavy-tailed
    • Pareto distribution for tail
    • Lognormal for body of distribution

Typical Workload
• Popularity
  • Zipf distribution (\( P = kr^{-1} \))
  • Surprisingly common
• Embedded references
  • Number of embedded objects = pareto
• Temporal locality
  • Modeled as distance into push-down stack
  • Lognormal distribution of stack distances
• Request interarrival
  • Bursty request patterns
HTTP Caching

- Clients often cache documents
  - Challenge: update of documents
  - If-Modified-Since requests to check
    - HTTP 0.9/1.0 used just date
    - HTTP 1.1 has file signature as well
  - When/how often should the original be checked for changes?
    - Check every time?
    - Check each session? Day? Etc?
- Use Expires header
  - If no Expires, often use Last-Modified as estimate

Example Cache Check Request

GET / HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
If-Modified-Since: Mon, 29 Jan 2001 17:54:18 GMT
If-None-Match: "7a11f-10ed-3a75ae4a"
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Host: www.seshan.org
Connection: Keep-Alive

Example Cache Check Response

HTTP/1.1 304 Not Modified
Date: Tue, 27 Mar 2001 03:50:51 GMT
Server: Apache/1.3.14 (Unix) (Red-Hat/Linux) mod_ssl/2.7.1 OpenSSL/0.9.5a
DAV/1.0.2 PHP/4.0.1pl2 mod_perl/1.24
Connection: Keep-Alive
Keep-Alive: timeout=15, max=100
ETag: "7a11f-10ed-3a75ae4a"

HTTP 0.9/1.0

- One request/response per TCP connection
  - Simple to implement
  - Disadvantages
    - Multiple connection setups  three-way handshake each time
    - Several extra round trips added to transfer
    - Multiple slow starts

Single Transfer Example

More Problems

- Short transfers are hard on TCP
  - Stuck in slow start
  - Loss recovery is poor when windows are small
- Lots of extra connections
  - Increases server state/processing
- Server also forced to keep TIME_WAIT connection state
  - Why must server keep these?
  - Tends to be an order of magnitude greater than #of active connections, why?
Overview

- HTTP Basics
- HTTP Fixes
- ALF
- Congestion Manager

Netscape Solution

- Use multiple concurrent connections to improve response time
- Different parts of Web page arrive independently
- Can grab more of the network bandwidth than other users
- Doesn’t necessarily improve response time
  - TCP loss recovery ends up being timeout dominated because windows are small

Persistent Connection Solution

- Multiplex multiple transfers onto one TCP connection
  - Serialize transfers → client makes next request only after previous response
- How to demultiplex requests/responses
  - Content-length and delimiter → same problems as before
  - Block-based transmission – send in multiple length delimited blocks
  - Store-and-forward – wait for entire response and then use content-length
  - PM95 solution – use existing methods and close connection otherwise

Persistent Connection Example

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 RTT</td>
<td>Server reads from disk</td>
</tr>
<tr>
<td>1 RTT</td>
<td>Client sends HTTP request for HTML</td>
</tr>
<tr>
<td>1 RTT</td>
<td>Server reads from disk</td>
</tr>
<tr>
<td>2 RTT</td>
<td>Client parses HTML</td>
</tr>
<tr>
<td>2 RTT</td>
<td>Client sends HTTP request for image</td>
</tr>
<tr>
<td>Image begins to arrive</td>
<td></td>
</tr>
</tbody>
</table>

Persistent Connection Solution

- Serialized requests do not improve response time
- Pipelining requests
  - Getall – request HTML document and all embeds
    - Requires server to parse HTML files
    - Doesn’t consider client cached documents
  - Getlist – request a set of documents
- Prefetching
  - Must carefully balance impact of unused data transfers
  - Not widely used due to poor hit rates

Persistent Connection Performance

- Benefits greatest for small objects
- Up to 2x improvement in response time
- Server resource utilization reduce due to fewer connection establishments and fewer active connections
- TCP behavior improved
  - Longer connections help adaptation to available bandwidth
  - Larger congestion window improves loss recovery
Remaining Problems

- Application specific solution to transport protocol problems
- Stall in transfer of one object prevents delivery of others
- Serialized transmission
  - Much of the useful information in first few bytes
  - Can "packetize" transfer over TCP
    - HTTP 1.1 recommends using range requests
    - MUX protocol provides similar generic solution
- Solve the problem at the transport layer
  - Tcp-Int/CM/TCP control block interdependence

Overview

- HTTP Basics
- HTTP Fixes
- ALF
- Congestion Manager

Integrated Layer Processing (ILP)

- Layering is convenient for architecture but not for implementations
- Combining data manipulation operations across layers provides gains
  - E.g. copy and checksum combined provides 90Mbps vs. 60Mbps separated
- Protocol design must be done carefully to enable ILP
- Presentation overhead, application-specific processing >> other processing
  - Target for ILP optimization

Application Lever Framing (ALF)

- Objective: enable application to process data ASAP
- Application response to loss
  - Retransmit (TCP applications)
  - Ignore (UDP applications)
  - Recompute/send new data (clever application)
- Expose unit of application processing (ADU) to protocol stack

Application Data Units Requirements

- ADUs can be processed in any order
- Naming of ADUs should help identify position in stream
- Size
  - Enough to process independently
  - Impact on loss recovery
  - What if size is too large?
Socket API

- **Connection establishment**
  - Connect – active connect
  - Bind, listen, accept – passive connect
- **Transmission/reception**
  - Send & sends
  - Recv & recvfrom
  - Pass large buffer to protocol stack
  - Protocol transmits at either congestion controlled or line rate
- **Configuration/status**
  - E.g. advertised window, Nagle, multicast membership, etc.
  - Setsockopt, getsockopt
  - Very little information really passed to/controlled by application

CM Requirements

- Determining correct rate of transmission = congestion control
- What is wrong with congestion control today?
  - “Multimedia Transmissions Drive Net Toward Gridlock”
    - New York Times
    - 8/23/99
  - Tightly integrated with TCP
  - Congestion control is vital to network stability

Problems with Concurrent Flows

- Compete for resources
  - N “slow starts” = aggressive
  - No shared learning = inefficient

Problems Adapting to Network State

- TCP hides network state
  - New applications may not use TCP
  - Often do not adapt to congestion
  - Need system that helps applications learn and adapt to congestion

High Level View

- All congestion management tasks performed in CM
  - Applications learn and adapt using API

Design Requirements

- How does CM control when and whose transmissions occur?
  - Keep application in control of what to send
- How does CM discover network state?
  - What information is shared?
  - What is the granularity of sharing?

Key issues: API and information sharing
The CM Architecture

Transmitting Application (TCP, conferencing app, etc)

Prober

Congestion Controller

Scheduler

Sender Receiver

CM Protocol

Responder

Congestion Detector

Receiving Application

Application Protocol

Congestion Controller

- Responsible for deciding when to send a packet
- Window-based AIMD with traffic shaping
- Exponential aging when feedback low
- Halve window every RTT (minimum)
- Plug in other algorithms
- Selected on a “macro-flow” granularity

Scheduler

- Responsible for deciding who should send a packet
- Hierarchical round robin
- Hints from application or receiver
- Used to prioritize flows
- Plug in other algorithms
- Selected on a “macro-flow” granularity
- Prioritization interface may be different

Transmission API

- Buffered send
  - cm_send(data, length)
- Request/callback-based send
  - cm_request()
  - cmapp_send()
  - send()
  - cm_notify(nsent)

Transmission API (cont.)

- Request API: asynchronous sources
  - wait for (some_events) {
  - get_data();
  - send();
  }
- Synchronous sources
  - do_every_t_ms {
  - get_data();
  - send();
  }
- Solution: cmapp_update(rate, srtt) callback

Feedback about Network State

- Monitoring successes and losses
  - Application hints
  - Probing system
- Notification API (application hints)
  - Application calls cm_update(nsent, nrecd, congestion indicator, rtt)
**Probing System**
- Receiver modifications necessary
  - Support for separate CM header
  - Uses sequence number to detect losses
  - Sender can request count of packets received
- Receiver modifications detected/negotiated via handshake
  - Enables incremental deployment

**How will applications use CM?**
- TCP
  - Asynchronous API
  - Congestion controlled UDP
  - Buffered API
  - Conferencing applications
    - Synchronous API for real-time streams
    - Combination of others for other streams

**TCP/CM**
1. connect
2. cm_open
3. write
4. cm_request
5. cmapp_send
6. transmit
7. cm_notify
8. recv ACK
9. cm_update
10. close
11. cm_close

**CM Web Performance**
TCP/Newreno
CM greatly improves predictability and consistency

**Layered Streaming Audio**
- Audio adapts to available bandwidth
- Combination of flows compete equally with normal TCP

**CM Summary**
- Enables proper and stable congestion behavior
- Provides simple API that enables application to learn and adapt to network state
- Improves consistency and predictability of network transfers
- Provides benefit even when deployed at senders alone
Next Lecture: Caching & CDN’s

- Web caching hierarchies
- Content distribution networks
- Peer-to-peer networks
- Assigned reading
  - [K+99] Web Caching with Consistent Hashing