Network Applications: HTTP/1.1/2;
High-Performance Server Design (Per Thread)

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

9/28/2017
Outline

- Admin and recap
- HTTP “acceleration”
- Network server design
Admin

- HTTP server assignment (part 1) to be posted later today
Recap: HTTP

- Wide use of HTTP for Web applications

- Example: RESTful API
  - RESTful design
Recap: HTTP

- C-S app serving Web pages
  - message format
    - request/response line, header lines, entity body
    - simple methods, rich headers
  - message flow
    - stateless server, thus states such as cookie and authentication are needed in each message
Recap: Basic HTTP/1.0 Server

create ServerSocket(6789)

connSocket = accept()

read request from connSocket

Map URL to file

Read from file/ write to connSocket

close connSocket

It does not have to be a static file
Recap: Latency of Basic HTTP/1.0

- >= 2 RTTs per object:
  - TCP handshake --- 1 RTT
  - client request and server responds --- at least 1 RTT (if object can be contained in one packet)

Discussion: How to reduce HTTP latency?
Outline

- Admin and recap
- HTTP “acceleration”
Substantial Efforts to Speedup Basic HTTP/1.0

- Reduce the number of objects fetched [Browser cache]
- Reduce data volume [Compression of data]
- Header compression [HTTP/2]
- Reduce the latency to the server to fetch the content [Proxy cache]
- Remove the extra RTTs to fetch an object [Persistent HTTP, aka HTTP/1.1]
- Increase concurrency [Multiple TCP connections]
- Asynchronous fetch (multiple streams) using a single TCP [HTTP/2]
- Server push [HTTP/2]
Browser Cache and Conditional GET

- **Goal:** don’t send object if client has up-to-date stored (cached) version
- **client:** specify date of cached copy in http request
  - If-modified-since: <date>
- **server:** response contains no object if cached copy up-to-date:
  - HTTP/1.0 304 Not Modified

- **server:** response contains object modified
  - HTTP/1.1 200 OK
  - ...<data>
Web Caches (Proxy)

Goal: satisfy client request without involving origin server
Two Types of Proxies

(FORWARD) PROXY server

- Cache for static data
- Filter
- Access control

Typically in the same network as the client

REVERSE-PROXY server

- Load-balancing
- Filter
- Cache for static data

Typically in the same network as the server

The reverse-proxy is in the same network as the server

The proxy is in the same network as the client
Benefits of Forward Proxy

Assume: cache is “close” to client (e.g., in same network)

- smaller response time: cache “closer” to client
- decrease traffic to distant servers
  - link out of institutional/local ISP network often is bottleneck
No Free Lunch: Problems of Web Caching

- The major issue of web caching is how to maintain consistency

- Two ways
  - **pull**
    - Web caches periodically pull the web server to see if a document is modified
  - **push**
    - Whenever a server gives a copy of a web page to a web cache, they sign a lease with an expiration time; if the web page is modified before the lease, the server notifies the cache
HTTP/1.0 allows a single request outstanding, while HTTP/1.1 allows request pipelining

- On same TCP connection: server parses request, responds, parses new request, ...
- Client sends requests for all referenced objects as soon as it receives base HTML

**Benefit**

- Fewer RTTs
- See Joshua Graessley WWDC 2012 talk: 3x within iTunes
Example

- Visit cs home page using Chrome
HTTP/1.0, Keep-Alive, Pipelining

Is this the best we can do?

Source: http://chimera.labs.oreilly.com/books/1230000000545/ch11.html
HTTP/2 Basic Idea:
Remove Head-of-Line Blocking in HTTP/1.1

Data flows for sequential to parallel: two requests must be concurrent.

Demo: https://http2.akamai.com/demo

Source: http://chimera.labs.oreilly.com/books/1230000000545/ch11.html
Observing HTTP/2

- export SSLKEYLOGFILE=/tmp/keylog.txt
- Start Chrome, e.g. 
  /Applications/Google Chrome.app/Contents/MacOS/Google Chrome
- Visit HTTP/2 pages, such as https://http2.akamai.com
- See chrome://net-internals/#http2
HTTP/2 Design: Multi-Streams

HTTP/2 connection

Bit | +0..7 | +8..15 | +16..23 | +24..31 |
---|------|--------|--------|--------|
0   | Length | Type   |
32  | Flags  |
40  | R      |
... |        |

Frame Payload

HTTP/2 Binary Framing

HTTP/2 Header Compression

<table>
<thead>
<tr>
<th>Request headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>:method</td>
</tr>
<tr>
<td>:scheme</td>
</tr>
<tr>
<td>:host</td>
</tr>
<tr>
<td>:path</td>
</tr>
<tr>
<td>user-agent</td>
</tr>
<tr>
<td>custom-hdr</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Static table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 :authority</td>
</tr>
<tr>
<td>2 :method</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>51 referer</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>62 user-agent</td>
</tr>
<tr>
<td>63 :host</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

| Dynamic table                    |

<table>
<thead>
<tr>
<th>Encoded headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>63</td>
</tr>
<tr>
<td>19 Huffman(“/resource”)</td>
</tr>
<tr>
<td>62 Huffman(“custom-hdr”)</td>
</tr>
<tr>
<td>Huffman(“some-value”)</td>
</tr>
</tbody>
</table>
HTTP/2 Stream Dependency and Weights

* implicit root

A 12

B 4

D 1

C 8

stream

weight

A 12

B 4

D 1

C 8

E 8

D 1

C 8

A 12

B 4

E 8

C 8

A 12

B 4
HTTP/2 Server Push

HTTP/2 connection

Stream 1: /page.html (client request)
Stream 2: /script.js (push promise)
Stream 4: /style.css (push promise)
Outline

- Admin and recap
- HTTP “acceleration”
- Network server design
**WebServer Implementation**

- **create ServerSocket(6789)**
  - connSocket = accept()
    - read request from connSocket
      - read local file
        - write file to connSocket
          - close connSocket

Discussion: what does each step do and how long does it take?
Demo

- Try TCPServer
- Start two TCPClient
  - Client 1 starts early but stops
  - Client 2 starts later but inputs first
Server Processing Steps

1. Accept Client Connection
2. Read Request
3. Find File
4. Send Response Header
5. Read File
6. Send Data

Notes:
- May block waiting on network
- May block waiting on disk I/O
Many socket and IO operations can cause a process to block, e.g.,
- `accept`: waiting for new connection;
- `read a socket`: waiting for data or close;
- `write a socket`: waiting for buffer space;
- `I/O read/write`: for disk to finish
Goal: Limited Only by Resource Bottleneck

Before

After
Outline

- Admin and recap
- HTTP “acceleration”
- Network server design
  - Overview
  - Multi-thread network servers
Multi-Threaded Servers

- **Motivation:**
  - Avoid blocking the whole program (so that we can reach bottleneck throughput)

- **Idea: introduce threads**
  - A thread is a sequence of instructions which may execute in parallel with other threads
  - When a blocking operation happens, only the flow (thread) performing the operation is blocked
Every Java application has at least one thread

- The “main” thread, started by the JVM to run the application’s main() method
- Most JVMs use POSIX threads to implement Java threads

main() can create other threads

- Explicitly, using the Thread class
- Implicitly, by calling libraries that create threads as a consequence (RMI, AWT/Swing, Applets, etc.)
Thread vs Process

A computer

```java
int x;
foo() {
...x...
}
```

Processes do not share data

```java
int x;
foo() {
...x...
}
```

Threads share data within a process
Background: Java Thread Class

- Threads are organized into thread groups
  - A thread group represents a set of threads
    ```java
class
activeGroupCount();
```
  - A thread group can also include other thread groups to form a tree
  - Why thread group?

http://java.sun.com/javase/6/docs/api/java/lang/ThreadGroup.html
Creating Java Thread

- Two ways to implement Java thread
  1. Extend the Thread class
     - Overwrite the run() method of the Thread class
  2. Create a class C implementing the Runnable interface, and create an object of type C, then use a Thread object to wrap up C

- A thread starts execution after its start() method is called, which will start executing the thread’s (or the Runnable object’s) run() method
- A thread terminates when the run() method returns

http://java.sun.com/javase/6/docs/api/java/lang/Thread.html
Option 1: Extending Java Thread

class PrimeThread extends Thread {
    long minPrime;

    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime . . .
    }
}

PrimeThread p = new PrimeThread(143);
p.start();
Option 1: Extending Java Thread

class RequestHandler extends Thread {
    RequestHandler(Socket connSocket) {
        // …
    }
    public void run() {
        // process request
    }
    …
}

Thread t = new RequestHandler(connSocket);
t.start();
Option 2: Implement the Runnable Interface

class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime . . .
    }
}

PrimeRun p = new PrimeRun(143);
new Thread(p).start();
Example: a Multi-threaded TCPServer

- Turn TCPServer into a multithreaded server by creating a thread for each accepted request
Per-Request Thread Server

```java
main() {
    ServerSocket s = new ServerSocket(port);
    while (true) {
        Socket conSocket = s.accept();
        RequestHandler rh = new RequestHandler(conSocket);
        Thread t = new Thread(rh);
        t.start();
    }
}

class RequestHandler implements Runnable {
    RequestHandler(Socket connSocket) {
    ...
}
    public void run() {
        //
    }
}
```

Try the per-request-thread TCP server: TCPServerMT.java
Summary: Implementing Threads

class RequestHandler  
    extends Thread  
    {  
        RequestHandler(Socket connSocket)  
        {  
            ...  
        }  
        public void run() {  
            // process request  
        }  
        ...  
    }  

Thread t = new RequestHandler(connSocket);  
t.start();
Modeling Per-Request Thread Server: Theory

\[ \lambda \left( k+1 \right) \mu \]

Welcome
Socket
Queue
Problem of Per-Request Thread: Reality

- High thread creation/deletion overhead
- Too many threads → resource overuse → throughput meltdown → response time explosion
  - Q: given avg response time and connection arrival rate, how many threads active on avg?
Background: Little’s Law (1961)

- For any system with no or (low) loss.
- Assume
  - mean arrival rate \( \lambda \), mean time \( R \) at system, and mean number \( Q \) of requests at system
- Then relationship between \( Q, \lambda, \) and \( R \):

\[
Q = \lambda R
\]

Example: Yale College admits 1500 students each year, and mean time a student stays is 4 years, how many students are enrolled?
Little’s Law

\[ Q = \lambda R \]

\[ \lambda = \frac{A}{t} \quad R = \frac{\text{Area}}{A}^t \quad Q = \frac{\text{time}}{A} \]
Discussion: How to Address the Issue

(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)