Network Applications: 
High-Performance Server Design 
(Proactive Async Servers; Operational Analysis; Multi-Servers)

Y. Richard Yang

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

10/11/2018
Outline

- Admin and recap
- High performance server
  - Thread design
  - Asynchronous design
  - Operational analysis
- Multiple servers
Assignment Three (HTTP server) Part 1 check point

Assignment Part 2 posted (there is one to-do place to be fixed today)
Recap: Multiplexed, Reactive I/O

- A different approach for avoiding blocking: *peek* system state, issue function calls only for those that are *ready*

**Basic abstractions**
- Channel (source)
- Selector
- PCB
Designing a good FSM is key for a good non-blocking select design.

There can be multiple types of FSMs:
- **Staged**: first read request and then write response
- **Mixed**: read and write mixed

Choice depends on protocol and tolerance of complexity, e.g.,
- HTTP/1.0 channel may use staged
- HTTP/1.1/2/Chat channel may use mixed
Outline

- Admin and recap
- High performance servers
  - Thread design
    - Per-request thread
    - Thread pool
      - Busy wait
      - Wait/notify
  - Asynchronous design
    - Overview
    - Nonblocking, selected servers--reactive programming
    - Proactive programming
Basic Idea: Asynchronous Initiation and Callback

- Issue of only peek:
  - Cannot handle initiation calls (e.g., read file, initiate a connection by a network client)

- Idea: asynchronous initiation (e.g., aio_read) and program specified completion handler (callback)
  - Also referred to as proactive (Proactor) nonblocking
Asynchronous Channel using Future/Completion Handler

Java 7 introduces ASynchronousServerSocketChannel and ASynchnorousSocketChannel beyond ServerSocketChannel and SocketChannel

- accept, connect, read, write return Futures or have a callback.

https://docs.oracle.com/javase/7/docs/api/java/nio/channels/AsynchronousServerSocketChannel.html

https://docs.oracle.com/javase/7/docs/api/java/nio/channels/AsynchronousSocketChannel.html
## Asynchronous I/O

<table>
<thead>
<tr>
<th>Asynchronous I/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsynchronousFileChannel</td>
<td>An asynchronous channel for reading, writing, and manipulating a file</td>
</tr>
<tr>
<td>AsynchronousSocketChannel</td>
<td>An asynchronous channel to a stream-oriented connecting socket</td>
</tr>
<tr>
<td>AsynchronousServerSocketChannel</td>
<td>An asynchronous channel to a stream-oriented listening socket</td>
</tr>
<tr>
<td>CompletionHandler</td>
<td>A handler for consuming the result of an asynchronous operation</td>
</tr>
<tr>
<td>AsynchronousChannelGroup</td>
<td>A grouping of asynchronous channels for the purpose of resource sharing</td>
</tr>
</tbody>
</table>

[Link to Oracle documentation](https://docs.oracle.com/javase/8/docs/api/java/nio/channels/package-summary.html)
## Example Async Calls

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abstract Future&lt;AsynchronousSocketChannel&gt;</code></td>
<td><code>accept()</code>: Accepts a connection.</td>
</tr>
<tr>
<td><code>abstract &lt;A&gt; void</code></td>
<td><code>accept(A attachment, CompletionHandler&lt;AsynchronousSocketChannel,? super A&gt; handler)</code>: Accepts a connection.</td>
</tr>
<tr>
<td><code>abstract Future&lt;Integer&gt;</code></td>
<td><code>read(ByteBuffer dst)</code>: Reads a sequence of bytes from this channel into the given buffer.</td>
</tr>
<tr>
<td><code>abstract &lt;A&gt; void</code></td>
<td><code>read(ByteBuffer [] dsts, int offset, int length, long timeout, TimeUnit unit, A attachment, CompletionHandler&lt;Long,? super A&gt; handler)</code>: Reads a sequence of bytes from this channel into a subsequence of the given buffers.</td>
</tr>
</tbody>
</table>

[https://docs.oracle.com/javase/8/docs/api/java/nio/channels/AsynchronousServerSocketChannel.html](https://docs.oracle.com/javase/8/docs/api/java/nio/channels/AsynchronousServerSocketChannel.html)
Using Future

```java
SocketAddress address = new InetSocketAddress(args[0], port);
AsynchronousSocketChannel client = AsynchronousSocketChannel.open();
Future<Void> connected = client.connect(address);
ByteBuffer buffer = ByteBuffer.allocate(100);
// wait for the connection to finish
connected.get();
// read from the connection
Future<Integer> future = client.read(buffer);
// do other things...
// wait for the read to finish...
future.get();
// flip and drain the buffer
buffer.flip();
WritableByteChannel out = Channels.newChannel(System.out);
out.write(buffer);
```

Using CompletionHandler

```java
class LineHandler implements CompletionHandler<Integer, ByteBuffer> {
    @Override
    public void completed(Integer result, ByteBuffer buffer) {
        buffer.flip();
        WritableByteChannel out = Channels.newChannel(System.out);
        try {
            out.write(buffer);
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }

    @Override
    public void failed(Throwable ex, ByteBuffer attachment) {
        System.err.println(ex.getMessage());
    }
}

ByteBuffer buffer = ByteBuffer.allocate(100);
CompletionHandler<Integer, ByteBuffer> handler = new LineHandler();
channel.read(buffer, buffer, handler);
```
Asynchronous Channel Implementation

- Asynchronous is typically based on Thread pool. If you are curious on its implementation, please read https://docs.oracle.com/javase/8/docs/api/java/nio/channels/AsynchronousChannelGroup.html
Summary: Event-Driven (Asynchronous) Programming

- Advantages
  - Single address space for ease of sharing
  - No synchronization/thread overhead

- Many examples: Google Chrome (libevent), Dropbox (libevent), nginx, click router, NOX controller, ...
Problems of Event-Driven Server

- Obscure control flow for programmers and tools

- Difficult to engineer, modularize, and tune

- Difficult for performance/failure isolation between FSMs
Another view

- Events obscure control flow
  - For programmers and tools

### Threads

```c
thread_main(int sock) {
  struct session s;
  accept_conn(sock, &s);
  read_request(&s);
  pin_cache(&s);
  write_response(&s);
  unpin(&s);
}

pin_cache(struct session *s) {
  pin(&s);
  if( !in_cache(&s) )
    read_file(&s);
}
```

### Events

```c
AcceptHandler(event e) {
  struct session *s = new_session(e);
  RequestHandler.enqueue(s);
}

RequestHandler(struct session *s) {
  ...; CacheHandler.enqueue(s);
}

CacheHandler(struct session *s) {
  pin(s);
  if( !in_cache(s) )
    ReadFileHandler.enqueue(s);
  else
    ResponseHandler.enqueue(s);
}
```

...  

```c
ExitHandler(struct session *s) {
  ...; unpin(&s); free_session(s); }
```
Summary: The High-Performance Network Servers Journey

- Avoid blocking (so that we can reach bottleneck throughput)
  - introduce threads, async select, async callback
- Limit unlimited thread overhead
  - Thread pool (share welcome, share Q)
- Coordinating data access
  - synchronization (lock, condition, synchronized)
- Coordinating behavior: avoid busy-wait
  - wait/notify; select FSM, Future/Listener
- Extensibility of SW/robustness
  - language support/design using interfaces
Beyond Java: Design Patterns

- We have seen Java as an example
- C++ and C# can be quite similar. For C++ and general design patterns:
HTTP Servers

<table>
<thead>
<tr>
<th>Ranking</th>
<th>Technology</th>
<th>Domains</th>
<th>Market Share</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Apache HTTP Server</td>
<td>13,593,009</td>
<td>52.41%</td>
</tr>
<tr>
<td>2</td>
<td>nginx</td>
<td>8,244,455</td>
<td>31.79%</td>
</tr>
<tr>
<td>3</td>
<td>Microsoft IIS</td>
<td>2,443,642</td>
<td>9.42%</td>
</tr>
</tbody>
</table>
Summary: Server Software Architecture

- Architectures
  - Multi threads
  - Asynchronous
  - Hybrid
    - Assigned reading: SEDA
    - Netty design

Figure 5: Staged event-driven (SEDA) HTTP server: This is a structural representation of the SEDA-based Web server, described in detail in Section 5.1. The application is composed as a set of stages separated by queues. Edges represent the flow of events between stages. Each stage can be independently managed, and stages can be run in sequence or in parallel, or a combination of the two. The use of event queues allows each stage to be individually load-conditioned, for example, by thresholding its event queue. For simplicity, some event paths and stages have been elided from this figure.
Recap: Best Server Design Limited Only by Resource Bottleneck
Some Questions

- When is CPU the bottleneck for scalability?
  - So that we need to add helper threads
- How do we know that we are reaching the limit of scalability of a single machine?

- These questions drive network server architecture design
- Some basic performance analysis techniques are good to have
Outline

- Admin and recap
- High performance server
  - Thread design
  - Asynchronous design
  - Operational analysis
- Multiple servers
Operational Analysis

- Relationships that do not require any assumptions about the distribution of service times or inter-arrival times
  - Hence focus on measurements

- Identified originally by Buzen (1976) and later extended by Denning and Buzen (1978).

- We touch only some techniques/results
  - In particular, bottleneck analysis
- More details see linked reading
Under the Hood (An example FSM)

start (arrival rate $\lambda$)

| Network | File I/O | Memory cache | CPU |

exit (throughput $\lambda$ until some center saturates)
## Operational Analysis: Resource Demand of a Request

<table>
<thead>
<tr>
<th>Resource</th>
<th>Visits</th>
<th>Units of Resource Time Per Visit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>$V_{CPU}$</td>
<td>$S_{CPU}$</td>
</tr>
<tr>
<td>Network</td>
<td>$V_{Net}$</td>
<td>$S_{Net}$</td>
</tr>
<tr>
<td>Disk</td>
<td>$V_{Disk}$</td>
<td>$S_{Disk}$</td>
</tr>
<tr>
<td>Memory</td>
<td>$V_{Mem}$</td>
<td>$S_{Mem}$</td>
</tr>
</tbody>
</table>
Operational Quantities

- **T**: observation interval
- **Ai**: # arrivals to device i
- **Bi**: busy time of device i
- **Ci**: # completions at device i
- **i = 0** denotes system

**Arrival rate** \( \lambda_i = \frac{A_i}{T} \)

**Throughput** \( X_i = \frac{C_i}{T} \)

**Utilization** \( U_i = \frac{B_i}{T} \)

**Mean service time** \( S_i = \frac{B_i}{C_i} \)
Utilization Law

Utilization \( U_i = \frac{B_i}{T} \)

\[ = \frac{C_i}{T} \frac{B_i}{C_i} \]

\[ = X_i S_i \]

- The law is independent of any assumption on arrival/service process.
- Example: Suppose NIC processes 125 pkts/sec, and each pkt takes 2 ms. What is utilization of the network NIC?
Deriving Relationship Between R, U, and S for one Device

- Assume flow balanced (arrival=throughput), Little’s Law:
  \[ Q = \lambda R = XR \]

- Assume PASTA (Poisson Arrival--memory-less arrival--Sees Time Average), a new request sees Q ahead of it, and FIFO
  \[ R = S + QS = S + XRS \]

- According to utilization law, \( U = XS \)
  \[ R = S + UR \quad \Rightarrow \quad R = \frac{S}{1-U} \]
Forced Flow Law

Assume each request visits device $i$ $V_i$ times

\[
\text{Throughput } X_i = \frac{C_i}{T}
\]
\[
= \frac{C_i}{C_0} \frac{C_0}{T}
\]
\[
= V_i X
\]
Define $D_i = V_i S_i$ as the total demand of a request on device $i$

The device with the highest $D_i$ has the highest utilization, and thus is called the **bottleneck**

Utilization $U_i = X_i S_i$

$$= V_i X S_i$$

$$= X V_i S_i$$
Bottleneck vs System Throughput

Utilization $U_i = XV_i S_i \leq 1$

$\rightarrow X \leq \frac{1}{D_{\text{max}}}$
Example 1

- A request may need
  - 10 ms CPU execution time
  - 1 Mbytes network bw
  - 1 Mbytes file access where
    - 50% hit in memory cache

- Suppose network bw is 100 Mbps, disk I/O rate is 1 ms per 8 Kbytes (assuming the program reads 8 KB each time)

- Where is the bottleneck?
Example 1 (cont.)

- **CPU:**
  - $D_{CPU} = 10 \text{ ms (e.q. 100 requests/s)}$

- **Network:**
  - $D_{Net} = \frac{1 \text{ Mbytes}}{100 \text{ Mbps}} = 80 \text{ ms (e.q., 12.5 requests/s)}$

- **Disk I/O:**
  - $D_{disk} = \frac{0.5 \times 1 \text{ ms} \times 1\text{M}/8\text{K}}{} = 62.5 \text{ ms (e.q. = 16 requests/s)}$
Example 2

- A request may need
  - 150 ms CPU execution time (e.g., dynamic content)
  - 1 Mbytes network bw
  - 1 Mbytes file access where
    - 50% hit in memory cache

- Suppose network bw is 100 Mbps, disk I/O rate is 1 ms per 8 Kbytes (assuming the program reads 8 KB each time)

- Bottleneck: CPU -> use multiple threads to use more CPUs, if available, to avoid CPU as bottleneck
Interactive Response Time Law

- **System setup**
  - Closed system with $N$ users (e.g., remote desktops)
  - Each user sends in a request, after response, think time, and then sends next request

- **Notation**
  - $Z = $ user think-time, $R = $ Response time

- The total cycle time of a user request is $R+Z$

In duration $T$, #requests generated by each user: $\frac{T}{R+Z}$ requests
Interactive Response Time Law

- If $N$ users and flow balanced:

System Throughput $X = \text{Toal# req.} / T$

$$X = \frac{N \frac{T}{R+Z}}{T}$$

$$= \frac{N}{R+Z}$$

$$R = \frac{N}{X} - Z$$
Bottleneck Analysis

\[ X(N) \leq \min \left\{ \frac{1}{D_{\max}}, \frac{N}{D+Z} \right\} \]

\[ R(N) \geq \max \left\{ D, ND_{\max} - Z \right\} \]

- Here D is the sum of D_i
Proof

\[ X(N) \leq \min\{\frac{1}{D_{\text{max}}}, \frac{N}{D+Z}\} \]

\[ R(N) \geq \max\{D, ND_{\text{max}} - Z\} \]

We know

\[ X \leq \frac{1}{D_{\text{max}}} \quad R(N) \geq D \]

Using interactive response time law:

\[ R = \frac{N}{X} - Z \quad \Rightarrow \quad R \geq ND_{\text{max}} - Z \]

\[ X = \frac{N}{R+Z} \quad \Rightarrow \quad X \leq \frac{N}{D+Z} \]
Summary: Operational Laws

- Utilization law: \( U = XS \)
- Forced flow law: \( Xi = Vi X \)
- Bottleneck device: largest \( Di = Vi Si \)
- Little’s Law: \( Qi = Xi Ri \)
- Bottleneck bound of interactive response (for the given closed model):
  \[
  X(N) \leq \min\left\{ \frac{1}{D_{\text{max}}} , \frac{N}{D+Z} \right\}
  \]
  \[
  R(N) \geq \max \{ D, ND_{\text{max}} - Z \}
  \]
In Practice: Common Bottlenecks

- No more file descriptors
- Sockets stuck in TIME_WAIT
- High memory use (swapping)
- CPU overload
- Interrupt (IRQ) overload
while (true)
{
    identify_and_fix_bottlenecks();
    drink();
    sleep();
    notice_new_bottleneck();
}
Summary: High-Perf. Network Server

- Avoid blocking (so that we can reach bottleneck throughput)
  - Introduce threads, async io
- Limit unlimited thread overhead
  - Thread pool
- Shared variables
  - Synchronization (lock, synchronized, condition)
- Avoid busy-wait
  - Wait/notify; FSM; asynchronous channel/Future/Handler
- Extensibility/robustness
  - Language support/Design for interfaces
- System modeling and measurements
  - Queueing analysis, operational analysis
Outline

- Admin and recap
- High performance server
  - Thread design
  - Asynchronous design
  - Operational analysis
- Multiple servers
Why Multiple Servers?

- Scale a single server that encounters bottleneck
- Scale a single server that has too large latency
- Add fault tolerance to a single server
- Match with settings where resources may be naturally distributed at different machines (e.g., run a single copy of a database server due to single license; access to resource from third party)
- Achieve modular software architecture (e.g., front end, business logic, and database)
FB Data Centers

*POP = points of presence.
Discussion: Requirements in Designing Load-Balancing Multiple Servers

- Provide naming abstraction
- Optimize resource utilization/performance goal
- Achieve fault tolerance
- ...


Components of a Load-Balancing Multiple Servers System

- Service/resource discovery (static, zookeeper, etc, consul)
- Health/state monitoring of servers/connecting networks
- Load balancing mechanisms/algorithms
  - Also called a request routing system
Request Routing: Basic Architecture

server state
(load/capacity/content)

net state/capacity: path property between servers/clients

notify client about selection (direction mech.)

requests

server selection algorithm
Request Routing Mechanisms

- DNS based request routing
- L4/network request routing
- L7/application request routing
Example: FB Architecture
Example: Wikipedia Architecture

Example: Hybrid Request Routing View

Cluster 1 in US East
  Load balancer

Cluster 2 in US West
  Load balancer

Cluster 3 in Europe
  Load balancer
  proxy
  Load balancer
  servers

DNS name1
  IP1

DNS name2
  IP2
  IPn
Outline

- Recap
- Single network server
- Multiple network servers
  - Why multiple servers
  - Overview
  - DNS based request routing
Basic DNS Indirection and Rotation

%dig +short cnn.com

157.166.226.25
157.166.226.26

IP address of cnn.com

157.166.226.25
157.166.226.26

DNS server for cnn.com

157.166.226.25
157.166.226.26
157.166.255.18
CDN Using DNS (Akamai Architecture as an Example)

- Content publisher (e.g., cnn)
  - provides base HTML documents
  - runs origin server(s); but delegates heavy-weight content (e.g., images) to CDN

- Akamai runs
  - (~240,000) edge servers in 130 countries within 1700 networks
    - Claims 85% Internet users are within a single "network hop" of an Akamai CDN server.
  - customized DNS redirection servers to select edge servers based on
    - closeness to client browser, server load

Linking to Akamai

- Originally, URL Akamaization of embedded content: e.g.,
  
  `<IMG SRC= http://www.provider.com/image.gif >`
  
  changed to
  
  `<IMG SRC = http://a661.g.akamai.net/hash/image.gif>`

  Note that this DNS redirection unit is per customer, not individual files.

- URL Akamaization is becoming obsolete and supported mostly for legacy reasons
Exercise

- Check any web page of cnn and find a page with an image
- Find the URL
- Use 
  \%dig +trace 
  to see DNS load direction
Akamai Load-Balancing DNS Name

- Akamai
  - e2466.dscg.akamaiedge.net (why two levels in the name?)
Two-Level Direction

- high-level DNS determines proximity, directs to low-level DNS;
  With query dscg.akamaiedge.net and client IP, directs to region (low-level)

- low-level DNS: each manages a close-by cluster of servers
  With query e2466.dscg.akamaiedge.net and client IP, directs to specific server
If the directed edge server does not have requested content, it goes back to the original server (source).
Local DNS Alg: Considerations

- Load on each edge server does not exceed its server capacity
- Maximize caching state of each server
- Minimize the number of busy servers
Example Local DNS Alg:

- Details of Akamai algorithms are proprietary
- A Bin-Packing algorithm (column 12 of Akamai Patent) every T second
  - Compute the load to each publisher $k$ (called serial number)
  - (estimate the number of needed servers)
  - Sort the publishers from increasing load
  - For each publisher, compute a sequence of random numbers using a hash function
  - Assign the publisher to the first server that does not overload
Backup Slides
Experimental Study of Akamai Load Balancing

- **Methodology**
  - 2-months long measurement
  - 140 PlanetLab nodes (clients)
    - 50 US and Canada, 35 Europe, 18 Asia, 8 South America, the rest randomly scattered
  - Every 20 sec, each client queries an appropriate CNAME for Yahoo, CNN, Fox News, NY Times, etc.

See http://www.aqualab.cs.northwestern.edu/publications/Ajsu06DBA.pdf
Server Pool: to Yahoo

Target: a943.x.a.yimg.com (Yahoo)

Client 1: Berkeley
Client 2: Purdue

Day

Night

Web replica IDs

Timestamp
Server Diversity for Yahoo

- Majority of PL nodes see between 10 and 50 Akamai edge-servers.
- Nodes far away from Akamai hot-spots.
Server Pool: Multiple Akamai Hosted Sites

- Yahoo!
- Amazon
- AOL
- NYTimes
- Apple
- Monster
- FOX News
- MSN
- PCWorld
- FEMA

![Graph showing the number of Akamai Web Replicas for various clients including cs.purdue.edu, comet.columbia.edu, cs.uoregon.edu, cs.umass.edu, and lbnl.nodes.planet-lab.org.](image)
Load Balancing Dynamics

![Graph showing CDF of inter-redirection time for Berkeley, Brazil, and Korea.]
Redirection Effectiveness: Measurement Methodology

Akamai Low-Level DNS Server

Planet Lab Node

9 Best Akamai Replica Servers
Do redirections reveal network conditions?

- Rank = r1 + r2 - 1
  - 16 means perfect correlation
  - 0 means poor correlation

MIT and Amsterdam are excellent

Brazil is poor

csail.mit.edu
 cs.vu.nl
 pop-ce.rnp.br

Percentage of time Akamai's selection is better or equal to rank
A system named Cartographer (written in Python) processes measurement data and configures the DNS maps of individual DNS servers (open source tinydns)