Network Applications:
Operational Analysis;
MultiServer Request Routing Overview;
DNS Protocol Design

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs434/

03/03/2021
Outline

- Admin and recap
- High-performance, robust network server
  - Basic transport API
  - Network server design
    - Overview of basic issues
    - Multi-thread/process network server
    - Reactive (selector multiplexing) network server
    - Proactive network server
    - Basic operational analysis methods
- Infrastructure with multiple servers
  - overview
  - DNS
Remaining office hours this week and early next week

- Wednesday: 5:15-6:30
- Thursday: 10:00-11:00
- Friday: 11:00 – 12:00
- Saturday: 5-6 pm
- Sunday: 8-9 pm
- Monday: 5:15-6:30
Recap: General, Extensible Select Server

- Fixed accept/read/write functions (handlers) are not general, extensible design
- There are multiple possible design options, one common design
  - Attachment stores generic event handler (which stores state)
    - Define interfaces
      - IAcceptHandler and
      - IReadWriteHandler
    - Retrieve handlers at run time
    - EchoLine last class assumes a simpler design

```java
if (key.isAcceptable()) { // a new connection is ready
    IAcceptHandler aH = (IAcceptHandler) key.attachment();
    aH.handleAccept(key);
}

if (key.isReadable() || key.isWritable()) {
    IReadWriteHandler rwH = IReadWriteHandler)key.attachment();
    if (key.isReadable())  rwH.handleRead(key);
    if (key.isWritable())  rwH.handleWrite(key);
}
```
Class Diagram of Handler Based Design

- **Dispatcher**
  - ...  

- **IChannelHandler**
  - handleException();

- **IAcceptHandler**
  - handleAccept();

- **IAcceptHandler**
  - implements 
  - **Acceptor**

- **IReadWriteHandler**
  - handleRead();
  - handleWrite();
  - getInitOps();

- **ISocketReadWriteHandlerFactory**
  - createHandler();

- **EchoReadWriteHandlerFactory**
  - createHandler();
Class Diagram of Handler Based Design

- **Dispatcher**
  - ...

- **IChannelHandler**
  - handleException();

- **IAcceptHandler**
  - handleAccept();

- **IReadWriteHandler**
  - handleRead();
  - handleWrite();
  - getInitOps();

- **NewReadWriteHandler**
  - handleRead();
  - handleWrite();
  - getInitOps();

- **ISocketReadWriteHandlerFactory**
  - createHandler();

- **EchoReadWriteHandlerFactory**
  - createHandler();

- **NewReadWriteHandlerFactory**
  - createHandler();

- **EchoReadWriteHandler**
  - handleRead();
  - handleWrite();
  - getInitOps();

- **Accept**
  - implements **IAcceptHandler**

- **NewReadWriteHandler**
  - handleRead();
  - handleWrite();
  - getInitOps();
Summary: Single Server Software Architecture

- The server architecture may evolve, e.g.,
  - Apache processes requests with Multi-Processing-Modules:
    - Oldest prefork (https://httpd.apache.org/docs/2.4/mod/prefork.html) spawns new process with one thread on every request
      => multithreaded worker (https://httpd.apache.org/docs/2.4/mod/worker.html)
      => event (https://httpd.apache.org/docs/2.4/mod/event.html)

- One may use hybrid architectures

- Good source of inspiration: nginx, Apache, Netty source code
Outline

- Admin and recap
- High-performance, robust network server
  - Basic transport API
  - Network server design
    - Overview of basic issues
    - Multi-thread/process network server
    - Reactive (selector multiplexing) network server
    - Proactive network server
    - Basic operational analysis methods
Goal: Limited Only by Resource Bottleneck

Before

CPU
DISK
NET

After

CPU
DISK
NET
YouTube Design Alg.

while (true)
{
    identify_and_fix_bottlenecks();
    drink();
    sleep();
    notice_new_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
Operational Analysis

- Relationships that do not require any assumptions about the distribution of service times or inter-arrival times
  - Hence focus on measurements; recall Little’s Law

- 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 $\lambda$)

CPU

I/O request

network

File I/O

Memory cache

exit

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

- **CPU**: \( V_{CPU} \) visits for \( S_{CPU} \) units of resource time per visit
- **Network**: \( V_{Net} \) visits for \( S_{Net} \) units of resource time per visit
- **Disk**: \( V_{Disk} \) visits for \( S_{Disk} \) units of resource time per visit
- **Memory**: \( V_{Mem} \) visits for \( S_{Mem} \) units of resource time per visit
Operational Quantities

- $T$: observation interval
- $B_i$: busy time of device $i$
- $C_i$: # 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

Throughput $X_i = \frac{C_i}{T}$

\[
= \frac{C_i}{C_0} \frac{C_0}{T} \\
= V_i X
\]
Bottleneck Device

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.
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 ms (e.q. 100 requests/s)

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

- **Disk I/O:**
  - $D_{disk}$ = 0.5 * 1 ms * 1M/8K = 62.5 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
  - 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 $T$, the number of requests generated by each user: $\frac{T}{R+Z}$ requests
Interactive Response Time Law

- If N users and flow balanced:

System Throughput \( X = \text{Total# req.}/T \)

\[
T = \frac{N \cdot T}{R + Z} = \frac{N}{R + Z}
\]

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

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

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

- Here D is the sum of Di
Proof

- We know

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

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

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: $X_i = V_i X$
- Bottleneck device: largest $D_i = V_i S_i$
- Little’s Law: $Q_i = X_i R_i$
- 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: Additional Bottlenecks

- No more file descriptors
- Sockets stuck in TIME_WAIT
- High memory use (swapping)
- CPU overload
- Interrupt (IRQ) overload
Summary: The High-Performance, Robust Network Server Journey

- Avoid blocking (so that we can reach bottleneck throughput)
  - Introduce threads
- Limit unlimited thread overhead
  - Thread pool, select, async io
- Coordinating data access
  - Synchronization (lock, synchronized)
- Coordinating behavior: avoid busy-wait
  - Wait/notify; select FSM, Future/Listener
- Extensibility/robustness
  - Language support/design for interfaces
- Data driven architecture design
  - Operational analysis
### Production HTTP Servers/Frameworks Complexity

- **Deployment/success depends on many factors, e.g.,**
  - **Apache [1995]**
    - Apache configuration system, per directory .htaccess
    - Benefit: users can configure details in a shared hosting environment, without touching the global server configuration.
    - Counter: [https://www.danielmorell.com/guides/htaccess-seo/basics/dont-use-htaccess-unless-you-must](https://www.danielmorell.com/guides/htaccess-seo/basics/dont-use-htaccess-unless-you-must)
    - Modular design: [https://httpd.apache.org/docs/2.4/mod/](https://httpd.apache.org/docs/2.4/mod/)
  - **Nginx [2004]**
    - Nginx does not have a configuration system like Apache and hence it is less widely employed in shared, retail hosting servers
    - Nginx modules typically need to be enabled at build time, which can be technically more challenging, and the post-installation adding of modules is a bit more complicated.
Outline

- Admin and recap
- High-performance, robust network server
- Infrastructure with multiple servers
  - overview
Discussion: Why Multiple Servers
Common Benefits of Multiple Servers

- **Scalability**
  - Scaling beyond single server **throughput**
    - There is a fundamental limit on what a single server can
      - process (CPU/bw/disk throughput)
      - store (disk/memory)
  - Scaling beyond single geo location **latency**
    - There is a limit on the speed of light
    - Network detour and delay further increase the delay

- **Redundancy and fault tolerance**
  - **Redundancy** (e.g., to handle failures)
  - **Operation/management** (e.g., incremental upgrade)
Common Benefits of Multiple Servers

- System/software architecture
  - Software constraint (e.g., run a single copy of a database server due to single license)
  - Software modularity (e.g., front end, business logic, and database; microservice architectures)
Roadmap

- We first focus on using multiple largely homogenous (similar) servers providing resource/latency scaling
  - Later we will look at the general case of fault tolerance (e.g., distributed consistency) and network system architecture (e.g., microservice architectures, ...)

- Problem setting for now:
  - Direct clients to close-by service with available resources

- Challenge: Direction is scalable, transparent to clients as much as possible
Basic Abstractions We Next Cover

- **Name Abstraction** (DNS)
  - Example: www.yale.edu
  - Mapping: 1 IP
  - IP: 1 IP
- **IP/Locator Abstraction** (LB, IP anycast)
  - Example: 8.8.8.8
  - Mapping: 1 IP
  - IP: 1 IP
- **Connection/Path Abstraction** (MP TCP)
  - Example: Connection
  - Mapping: 1 Conn
  - Connection: 1 Conn
Hybrid Deployment

- Cluster 1 in US East
  - Load balancer
  - IP1
- Cluster 2 in US West
  - Load balancer
  - IP2
- Cluster 3 in Europe
  - Load balancer
  - IPn

Load balancer

App

name1

name2

Proxy

Load balance

Server
Example: FB Architecture
Example: Wikipedia Architecture
Outline

- Admin and recap
- High-performance, robust network server
- Using multiple network servers
  - overview
  - DNS (domain name system)
    - background
Domain Name System (DNS) Overview

- Basic function is to map name to some attributes; used for many functions beyond scaling
  - A good read is history of DNS, RFC 1034, RFC 1035

- Different types of usage of DNS for scaling:
  - DNS load direction by individual user, e.g.,
    - Facebook DNS direction: A system named Cartographer (written in Python) processes measurement data and configures the DNS maps of individual DNS servers (open source tinydns)
  - DNS load direction as a service, e.g.,
    - Amazon AWS Route 53 service https://aws.amazon.com/route53/
  - CDN (content distribution network) using DNS as the integration mechanism, e.g., Akamai

- Our goal: not only understand DNS for scaling, but DNS overall design as a tool and as an example of network system design
Domain Name System (DNS)

- Basic function
  - map between (domain name, service) to value, e.g.,
    - (www.cs.yale.edu, Addr) -> 128.36.229.30
    - (cs.yale.edu, Email) -> netra.cs.yale.edu
DNS Design: Dummy Design

- DNS itself can be considered as a client-server system as well
- How about a dummy design: introducing one super Internet DNS server?
Problems of a Single DNS Server

- Scalability and robustness bottleneck
- Administrative bottleneck
Outline

- Admin and recap
- High-performance, robust network server
- Using multiple network servers
  - overview
  - DNS (domain name system)
    - background
DNS Architecture: Distributed Management of the Domain Name Space

- A distributed name space managed by authoritative name servers
  - divided into zones, where each zone is a sub-tree of the global tree
  - each zone has its own (one or more) authoritative name servers
  - an authoritative name server of a zone may delegate a subset (i.e. a sub-tree) of its zone to another name server
Backup Slides
Example: Netflix

<table>
<thead>
<tr>
<th>Hostname</th>
<th>Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.netflix.com">www.netflix.com</a></td>
<td>Netflix</td>
</tr>
<tr>
<td>signup.netflix.com</td>
<td>Amazon</td>
</tr>
<tr>
<td>movies.netflix.com</td>
<td>Amazon</td>
</tr>
<tr>
<td>agmoviecontrol.netflix.com</td>
<td>Amazon</td>
</tr>
<tr>
<td>nflix.i.87f50a04.x.lcdn.nflximg.com</td>
<td>Level 3</td>
</tr>
<tr>
<td>netflix-753.vo.llnwd.net</td>
<td>Limelight</td>
</tr>
<tr>
<td>netflix753.as.nflximg.com.edgesuite.net</td>
<td>Akamai</td>
</tr>
</tbody>
</table>
Example: Netflix Manifest File

- Client player authenticates and then downloads manifest file from servers at Amazon Cloud.