Network Applications: Async Servers and Operational Analysis

Y. Richard Yang

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

10/03/2013
Assignment Three will be posted later today.

Dates for the two exams
Recap: Async Network Server

- Basic idea: non-blocking operations
  - asynchronous initiation (e.g., aio_read) and completion notification (callback)
  - peek system state to issue only ready operations
Recap: Java Async I/O Basis: OS State Polling

Example: a system call to check if sockets are ready for ops.

Completed connection

sendbuf full or has space

recvbuf empty or has data
Recap: Basic Dispatcher Structure

//clients register interests/handlers on events/sources
while (true) {
    - ready events = select() /* or selectNow(), or select(int timeout) is call to check the ready events from the registered interest events of sources */

    - foreach ready event {
        switch event type:
        accept: call accept handler
        readable: call read handler
        writable: call write handler
    }
}
while (true) {
    - selector.select()
    - Set readyKeys = selector.selectedKeys();

    - foreach key in readyKeys {
        switch event type of key:
        accept: call accept handler
        (accept conn, set READ/WRITE interest)
        readable: call read handler
        writable: call write handler
    }
}
Recap: Two Problems of V1

- Empty write
- Still read after no longer having data to read
Finite State Machine (FSM)

AsyncEchoServer/v2/EchoServer.java

FSM for each socket channel in v2:

- Read input
  - Interest = READ
  - Read data
    - Interest = Read+Write

- Read + Write
  - Read close
    - Interest = Write
  - Read + Write

- Write

- Finite state machine after fixing the two issues
  - Problem of the finite state machine?
Fix Remaining Empty Write

Initial

Read only

Read + Write

Write only

Idle

Write all data

Read data

read close

Write all data

Write all data
Finite-State Machine and Thread

- Why no need to introduce FSM for a thread version?
Another State Machine

InitInterest = READ

Read from client channel

Request complete (find terminator or client request close)

Generating response

RequestReady

Interest =

ResponseReady

ResponseSent

Write response

RequestReady

ResponseReady

ResponseReady

Interest = Write

RequestReady

ResponseReady

Close

Interest =

ResponseSent
Comparing FSMs

- V2:
  - Mixed read and write

- Example last slide: staged
  - First read request and then write response

- Exact design depends on application, e.g.,
  - HTTP/1.0 channel may use staged
  - Chat channel may use mixed
Extending v2

- Many real programs run the dispatcher in a separate thread to allow main thread to interact with users
  -> start dispatcher in its own thread

- Protocol specific coding, not reusable
  -> derive an async/io TCP server software framework so that porting it to a new protocol involves small edits (e.g., defining read/write handlers)
Attachment

- Attaching a `ByteBuffer` to each channel is a narrow design for simple echo servers
- A general design can use the attachment to store a callback that indicates not only data (state) but also the handler (function)
Extensible Dispatcher Design

- Attachment stores generic event handler
  - Define interfaces
    - IAcceptHandler and
    - IReadWriteHandler
  - Retrieve handlers at run time

```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);
}
```
Dispatcher Interface

- Register a channel to be selected and its handler object
- Update interest of a selectable channel
- Deregister
Handler Design: Acceptor

- What should an accept handler object know?
  - ServerSocketChannel (so that it can call accept)
    - Can be derived from SelectionKey in the call back
  - Dispatcher (so that it can register new connections)
    - Need to be passed in constructor or call back
  - What ReadWrite object to create (different protocols may use different ones)?
    - Pass a Factory object: SocketReadWriteHandlerFactory
Handler Design: ReadWriteHandler

- What should a ReadWrite handler object know?
  - SocketChannel (so that it can read/write data)
    - Can be derived from SelectionKey in the call back
  - Dispatcher (so that it can change state)
    - Need to be passed in constructor or in call back
Class Diagram of v3

**Dispatcher**
- registerNewSelection();
- deregisterSelection();
- updateInterests();
- ...

**IChannelHandler**
- handleException();

**IAcceptHandler**
- handleAccept();

**Acceptor**
- implements

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

**ISocketReadWriteHandlerFactory**
- createHandler();

**NewReadWriteHandlerFactory**
- createHandler();

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

**EchoReadWriteHandlerFactory**
- createHandler();
See AsyncEchoServer/v3/* .java
Discussion on v3

- In our current implementation (Server.java)

1. Create dispatcher
2. Create server socket channel and listener
3. Register server socket channel to dispatcher
4. Start dispatcher thread

Can we switch 3 and 4?
Extending v3

- A production network server often closes a connection if it does not receive a complete request in TIMEOUT.

- One way to implement time out is that
  - the read handler registers a timeout event with a timeout watcher thread with a call back
  - the watcher thread invokes the call back upon TIMEOUT
  - the callback closes the connection

Any problem?
Extending Dispatcher Interface

- Interacting from another thread to the dispatcher thread can be tricky
- Typical solution: async command queue

```
while (true) {
    - process async. command queue
    - ready events = select (or selectNow(), or select(int timeout)) to check for ready events from the registered interest events of SelectableChannels

    - foreach ready event
    call handler
}
```
Question

- How may you implement the async command queue to the selector thread?

```java
public void invokeLater(Runnable run) {
    synchronized (pendingInvocations) {
        pendingInvocations.add(run);
    }
    selector.wakeup();
}
```

see SelectorThread.java invokeLater
Question

- What if another thread wants to wait until a command is finished by the dispatcher thread?
public void invokeAndWait(final Runnable task)
        throws InterruptedException {
    if (Thread.currentThread() == selectorThread) {
        // We are in the selector's thread. No need to schedule
        // execution
        task.run();
    } else {
        // Used to deliver the notification that the task is executed
        final Object latch = new Object();
        synchronized (latch) {
            // Uses the invokeLater method with a newly created task
            this.invokeLater
                .call(new Runnable() {
                    public void run() {
                        task.run();
                        // Notifies
                        synchronized (latch) { latch.notify(); }
                    }
                });
            // Wait for the task to complete.
            latch.wait();
        }
        // Ok, we are done, the task was executed. Proceed.
    }
}
Extending v3

- In addition to management threads, a system may still need multiple threads for performance (why?)
  - FSM code can never block, but page faults, file io, garbage collection may still force blocking
  - CPU may become the bottleneck and there maybe multiple cores supporting multiple threads (typically 2 n threads)
Summary: Architecture

- Architectures
  - Multi threads
  - Asynchronous
  - Hybrid

- Assigned reading: SEDA
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
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
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);
}

...;
ExitHandler(struct session *s) {
    ...; unpin(&s); free_session(s); }
```
# State Management

- Events require manual state management
- Hard to know when to free
  - Use GC or risk bugs

## Threads

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

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

## Events

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

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

ExitHandler(struct session *s) {
    ...; unpin(&s); free_session(s);
}

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

---

**Web Server**

- Accept Conn.
- Read Request
- Pin Cache
- Read File
- Write Response
- Exit

[von Behren]
Avoid blocking (so that we can reach bottleneck throughput)
- Introduce threads

Limit unlimited thread overhead
- Thread pool, async io

Coordinating data access
- Synchronization (lock, synchronized)

Coordinating behavior: avoid busy-wait
- Wait/notify; FSM

Extensibility/robustness
- Language support/Design for interfaces
Beyond Class: Design Patterns

- We have seen Java as an example

- C++ and C# can be quite similar. For C++ and general design patterns:
Some Questions

- When is CPU the bottleneck for scalability?
  - So that we need to add helpers

- How do we know that we are reaching the limit of scalability of a single machine?

- These questions drive network server architecture design
Operational Analysis

- Relationships that do not require any assumptions about the distribution of service times or inter-arrival times.
- 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

File I/O

I/O request

exit

(throughput $\lambda$ until some center saturates)

Memory cache

network
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
- 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.

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

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

- Why is the problem difficult?
  - What are potential problems of just sending each new client to the lightest load server?