Network Applications:
High-performance Server Design:
Async Servers/Operational Analysis

Y. Richard Yang

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

2/24/2016
Admin

- Assignment three posted.

- Decisions
  - Projects or exam 2
  - Date for exam 1
Recap: Async Network Server

- Basic ideas: non-blocking operations
  1. **peek** system state (using a `select` mechanism) to issue only **ready** operations
  2. **asynchronous initiation** (e.g., `aio_read`) and completion **notification** (callback)
Recap: Async Network Server using Select

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

- Completed connection
- Sendbuf full or has space
- Recvbuf empty or has data
Recap: Async Network Server using Select

- A event loop issues commands, waits for events, invokes handlers (callbacks)

```java
// clients register interests/handlers on events/sources
while (true) {
  - ready events = select()
    /* or selectNow(),
    or select(int timeout) */

  - foreach ready event {
    switch event type:
      accept: call accept handler
      readable: call read handler
      writable: call write handler
  }
}
```
Recap: Need to Manage Finite State Machine

- Init
- Read input
- Read data
- Read + Write
- Read close
- Write
- Idle
- Accept Client Connection
- Read Request
- Find File
- Send Response Header
- Read File
- Send Data
Another Finite State Machine

Read from client channel

Request complete (find terminator or client request close)

Generating response

Closed

ResponseSent

ResponseReady

ResponseReady

ResponseReady

RequestReady

ResponseReady

InitInterest = READ

Interest= -

Interest= Write

RequestReady

ResponseReady

ResponseReady

RequestReady

ResponseReady

RequestReady

ResponseReady

RequestReady

ResponseReady
Finite State Machine Design

- **EchoServerV2:**
  - Mixed read and write

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

- **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
Our example EchoServer is for a specific protocol.

A general async/io programming framework tries to introduce structure to allow substantial reuse.

- Async io programming framework is among the more complex software systems.
- We will see one simple example, using EchoServer as a basis.
A More Extensible Dispatcher Design

- Fixed accept/read/write functions are not general design

- Requirement: map from key to handler

- A solution: Using attachment of each channel
  - Attaching a `ByteBuffer` to each channel is a narrow design for simple echo servers
  - A more general design can use the attachment to store a callback that indicates not only data (state) but also the handler (function)
A More 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);
}
```
What should an accept handler object know?

- ServerSocketChannel (so that it can call accept)
  - Can be derived from SelectionKey in the call back

- Selector (so that it can register new connections)
  - Can be derived from SelectionKey in the 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
  - Selector (so that it can change state)
    - Can be derived from SelectionKey in the call back
Class Diagram of SimpleNAIO

```
Dispatcher
 ...

IChannelHandler
 handleException();

IAcceptHandler
 handleAccept();

Acceptor
 implements

IReadWriteHandler
 handleRead();
 handleWrite();
 getInitOps();

EchoReadWriteHandlerFactory
 createHandler();

ISocketReadWriteHandlerFactory
 createHandler();
```
Class Diagram of SimpleNAI0

- **Dispatcher**
  - ...

- **IChannelExceptionHandler**
  - handleException();

- **IAcceptExceptionHandler**
  - handleAccept();

- **Acceptor**
  - implements

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

- **ISocketReadWriteHandlerFactory**
  - createHandler();

- **NewReadWriteHandlerFactory**
  - createHandler();

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

- **EchoReadWriteHandlerFactory**
  - createHandler();

SimpleNAIO

- See AsyncEchoServer/v3/*.java
Discussion on SimpleNAIO

- 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 SimpleNAIO

- 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

```java
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();
}
```
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(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.
}
Recap: Async Network Server

Basic idea: non-blocking operations

1. **peek** system state (select) to issue only **ready** operations

2. **asynchronous initiation** (e.g., aio_read) and **completion notification** (callback)
Alternative Design: Asynchronous Channel using Future/Listener

- Java 7 introduces `AsynchronousServerSocketChannel` and `AsynchronousSocketChannel` beyond `ServerSocketChannel` and `SocketChannel`
  - accept, connect, read, write return Futures or have a callback. Selectors are not used

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
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);
try {
    out.write(buffer);
} catch (IOException ex) {
    System.err.println(ex);
}

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);
Extending FSM

- 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

```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);
}

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) )
        read_file(&s);
    unpin(&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);
}
```

---

[von Behren]

Web Server

- Accept Conn.
- Read Request
- Pin Cache
- Read File
- Write Response
- Exit
Summary: The High-Performance Network Servers Journey

- 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; select FSM, Future/Listener
- 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

Memory cache

File I/O

I/O request

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
- 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 $U_i = \frac{B_i}{T}$

$= \frac{C_i}{T} \cdot \frac{B_i}{C_i}$

$= X_i S_i$

- 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, $U = XS$

$$R = S + UR \quad \Rightarrow \quad R = \frac{S}{1-U}$$