Network Applications: High-Performance Server Design (Async Select NonBlocking Servers)

Y. Richard Yang

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

10/9/2018

Admin

- Assignment Three (HTTP server) Part 1 check point
- Assignment Part 2 to be posted on Wednesday

Recap: Thread-Based Network Servers

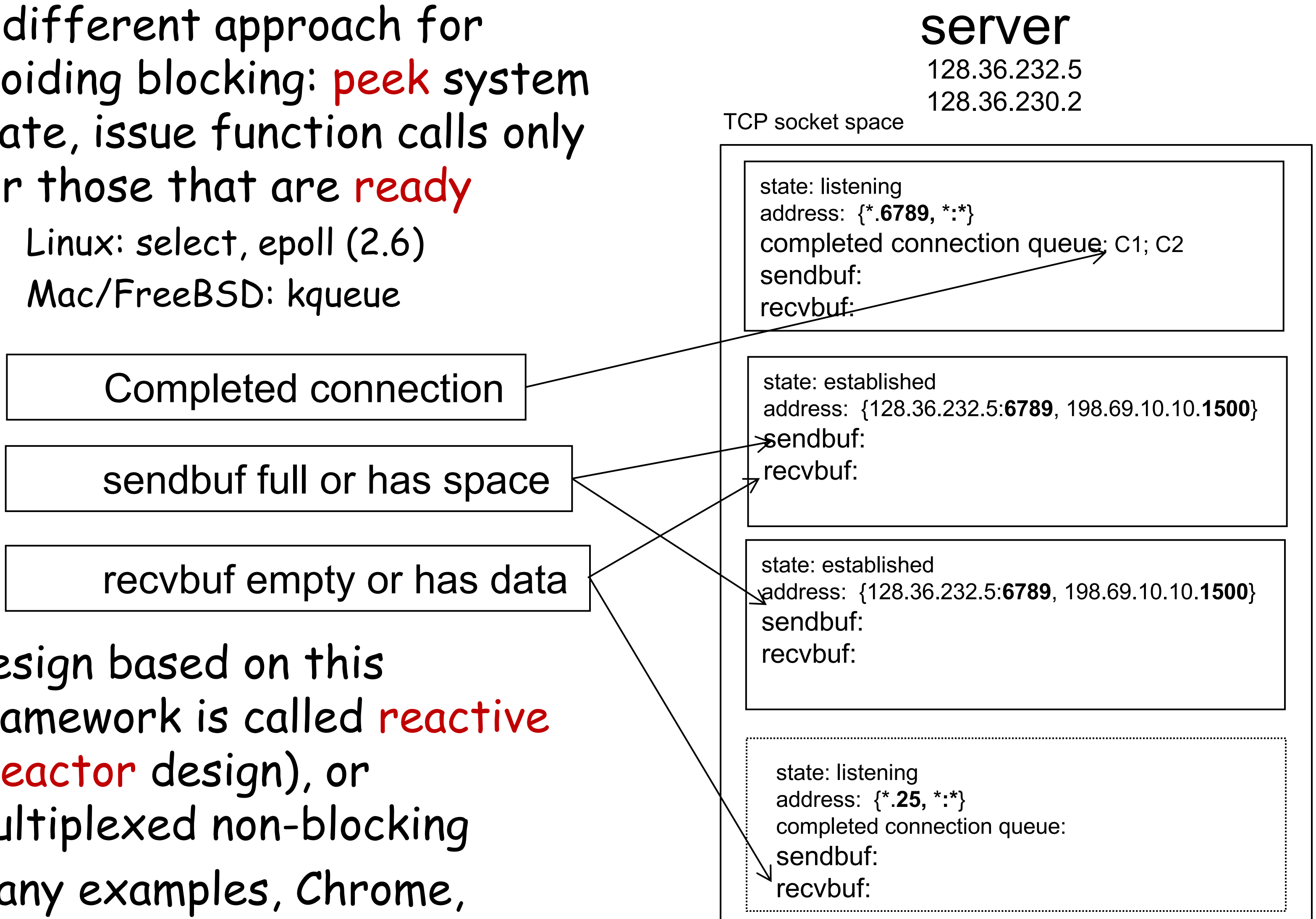
- Why: blocking operations; threads (execution sequences) so that only one thread is blocked
- How:
  - Per-request thread
    - problem: large # of threads and their creations/deletions may let overhead grow out of control
  - Thread pool
    - Design 1: Service threads compete on the welcome socket
    - Design 2: Service threads and the dispatcher thread coordinate on a shared queue
      - polling (busy wait)
      - suspension: wait/notify
    - An example control see http://httpd.apache.org/docs/2.4/mod/worker.html

Recap: Program Correctness Analysis

- Safety
  - Consistency (exclusive access)
  - app requirement, e.g., Q.remove() is not on an empty queue
- Liveness (progress)
  - main thread can always add to Q
  - every connection in Q will be processed
- Fairness
  - For example, in some settings, a designer may want the threads to share load equally

Recap: Multiplexed, Reactive I/O

- A different approach for avoiding blocking: peek system state, issue function calls only for those that are ready
  - Linux: select, epoll (2.6)
  - Mac/FreeBSD: kqueue

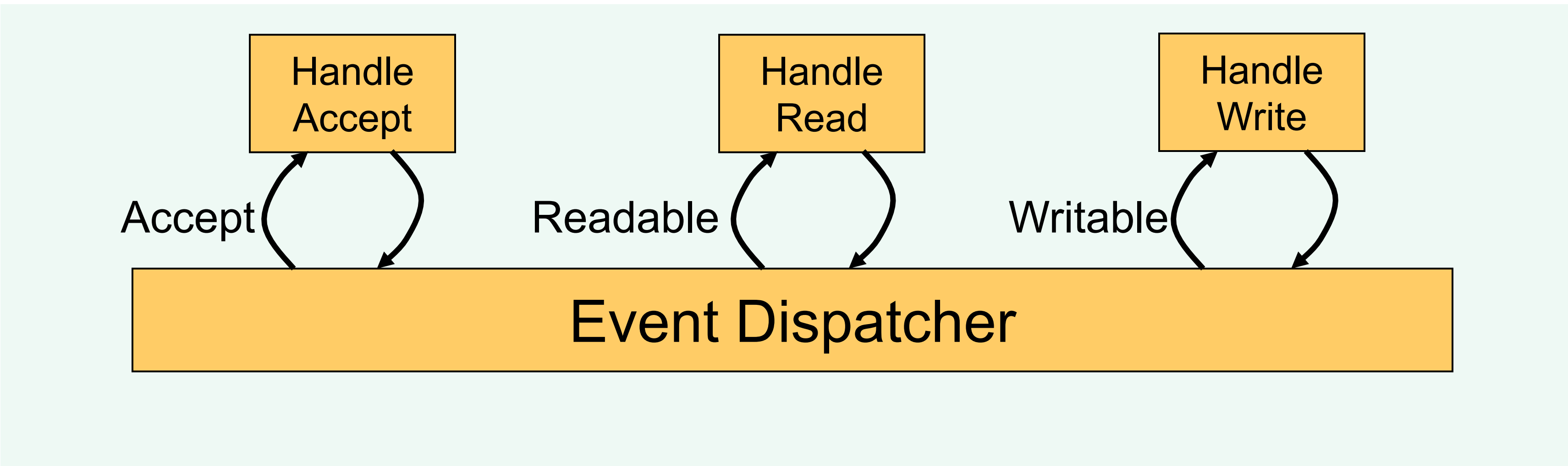


- Design based on this framework is called reactive (Reactor design), or multiplexed non-blocking
- Many examples, Chrome, Dropbox, nginx

Outline

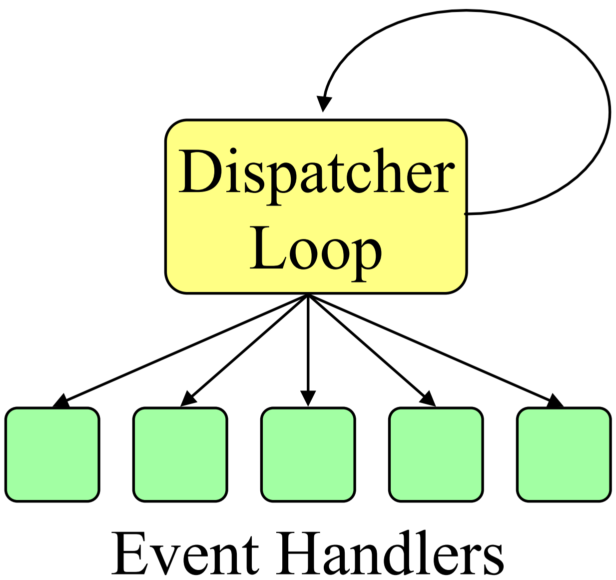
- Admin and recap
- High performance servers
  - Thread design
    - Per-request thread
    - Thread pool
      - Busy wait
      - Wait/notify
  - Asynchronous design
    - Overview
    - Multiplexed (selected), reactive programming

Multiplexed, Reactive Server Architecture



- Program registers events (e.g., acceptable, readable, writable) on channels (sources) to be monitored
- An infinite dispatcher loop:
  - Dispatcher asks OS to check if any ready event
  - Dispatcher calls (multiplexes) the handler of each ready event of each source
    - Handler should be non-blocking, to avoid blocking the event loop

Multiplexed, Non-Blocking Network Server




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

  - foreach ready event {
        switch event type:
        accept: call accept handler
        readable: call read handler
        writable: call write handler
    }

  - handle other events
}
```

Main Abstractions

- Main abstractions of multiplexed IO for network servers:
  - Channel (source): represents a connection to an entity capable of performing I/O operations;
  - Selection facilities;
  - Protocol control block (PCB): container to keep state/handler for each event/channel.
- Java abstractions see: https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html

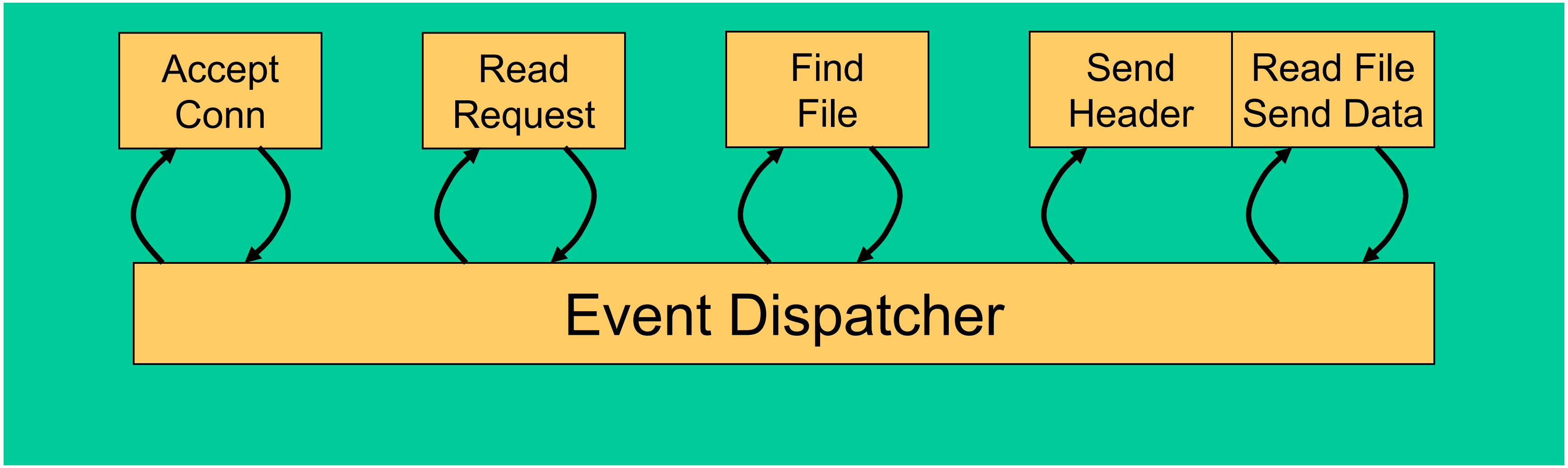
Multiplexed (Selectable), Non-Blocking Channels

SelectableChannel	A channel that can be multiplexed
DatagramChannel	A channel to a datagram-oriented socket
Pipe.SinkChannel	The write end of a pipe
Pipe.SourceChannel	The read end of a pipe
ServerSocketChannel	A channel to a stream-oriented listening socket
SocketChannel	A channel for a stream-oriented connecting socket

- Use `configureBlocking(false)` to make a channel non-blocking
- Note: Java `SelectableChannel` does not include file I/O

Selector

- The class `Selector` is the base of the multiplexer/dispatcher
- Constructor of Selector is protected; create by invoking the `open` method to get a selector (which design pattern?)



Selector and Registration

- A selectable channel registers events to be monitored with a `selector` with the `register` method

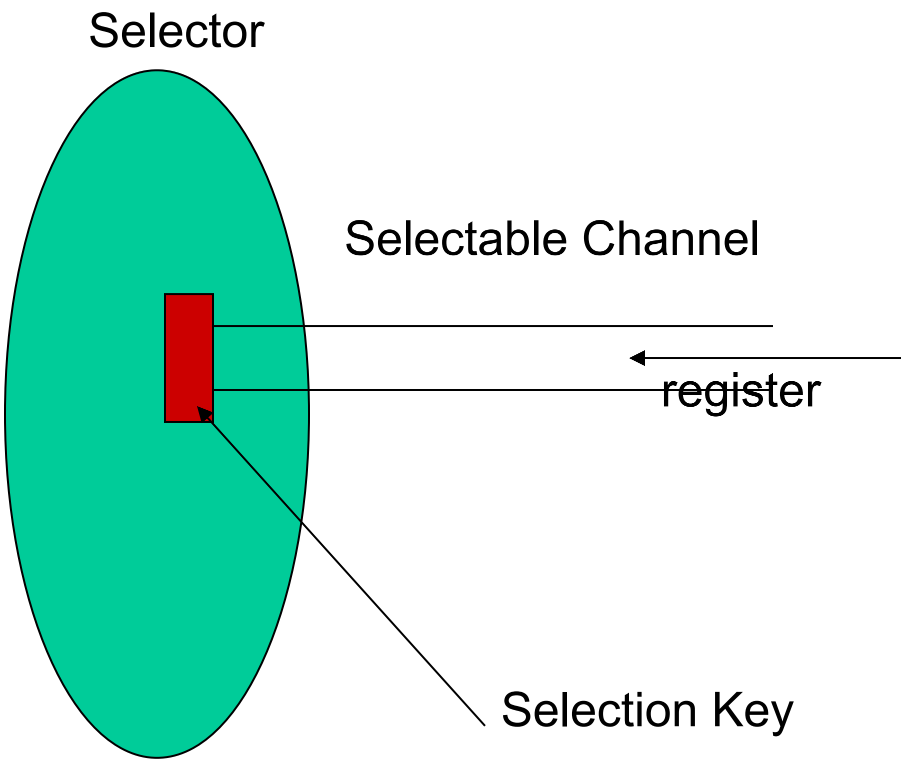
- The registration returns an object called a `SelectionKey`:

```
SelectionKey key =
   channel.register(selector, ops);
```

Java Selection I/O Structure

- A `SelectionKey` object stores:
  - interest set: events to check: `key.interestOps(ops)`
  - ready set: after calling select, it contains the events that are ready, e.g. `key.isReadable()`
  - an attachment that you can store anything you want, typically PCB

`key.attach(myObj)`



Checking Events

- A program calls `select` (or `selectNow()`, or `select(int timeout)`) to check for ready events from the registered SelectableChannels
  - Ready events are called the selected key set

```
selector.select();
Set readyKeys = selector.selectedKeys();
```

- The program iterates over the selected key set to process all ready events

I/O in Java: ByteBuffer

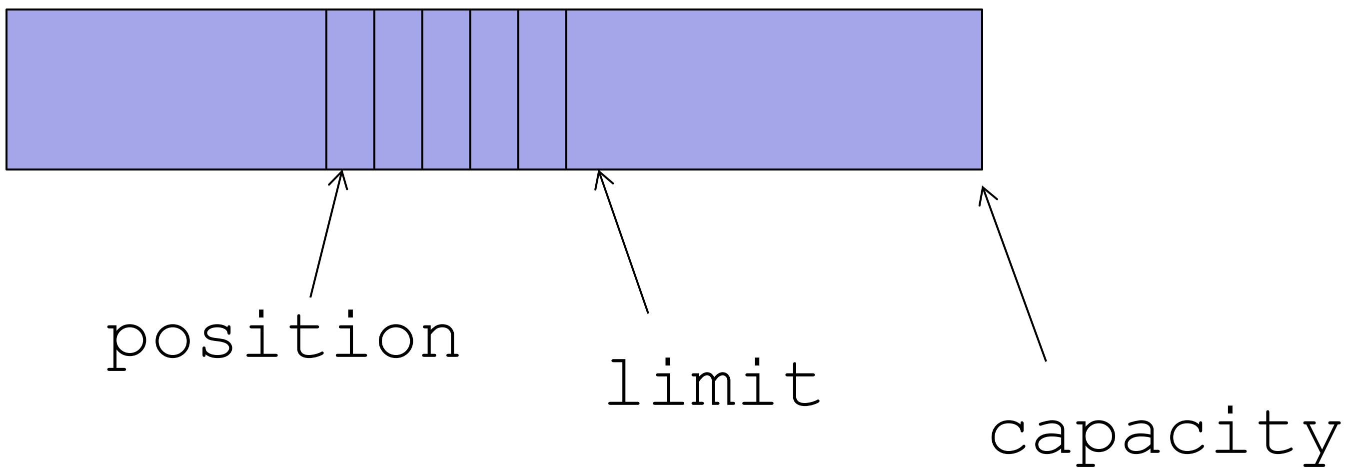
- Java SelectableChannels typically use ByteBuffer for read and write
  - channel.read(byteBuffer);
  - channel.write(byteBuffer);

- ByteBuffer is a powerful class that can be used for both read and write
- It is derived from the class Buffer
- Please be sure to read these data structures

Java ByteBuffer Hierarchy

Buffers	Description
Buffer	Position, limit, and capacity; clear, flip, rewind, and mark/reset
ByteBuffer	Get/put, compact, views; allocate, wrap
MappedByteBuffer	A byte buffer mapped to a file
CharBuffer	Get/put, compact; allocate, wrap
DoubleBuffer	' '
FloatBuffer	' '
IntBuffer	' '
LongBuffer	' '
ShortBuffer	' '

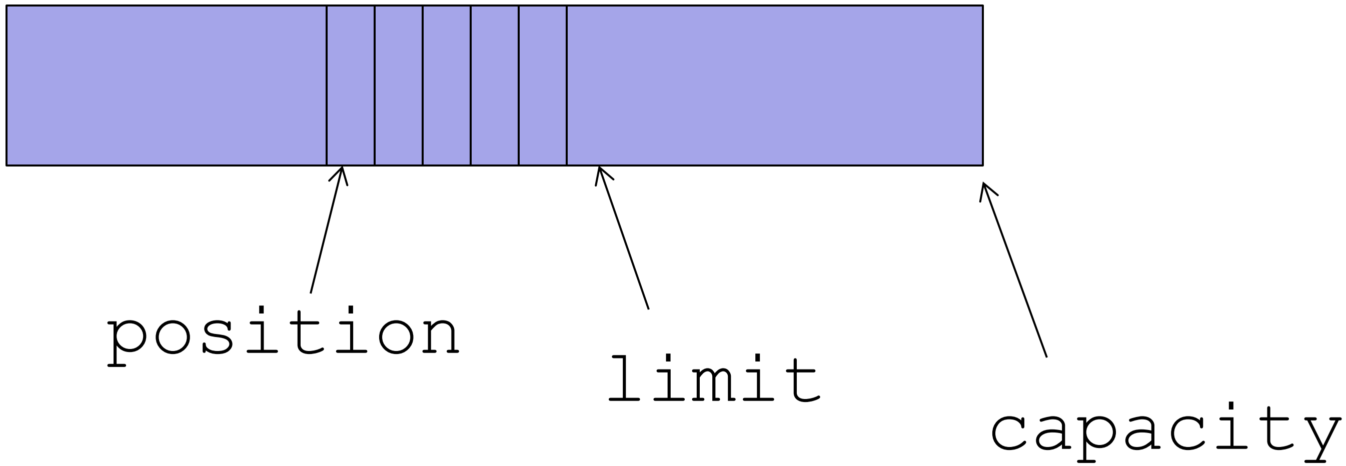
Buffer (relative index)



- Each Buffer has three numbers: position, limit, and capacity
  - Invariant: 0 <= *position* <= *limit* <= *capacity*

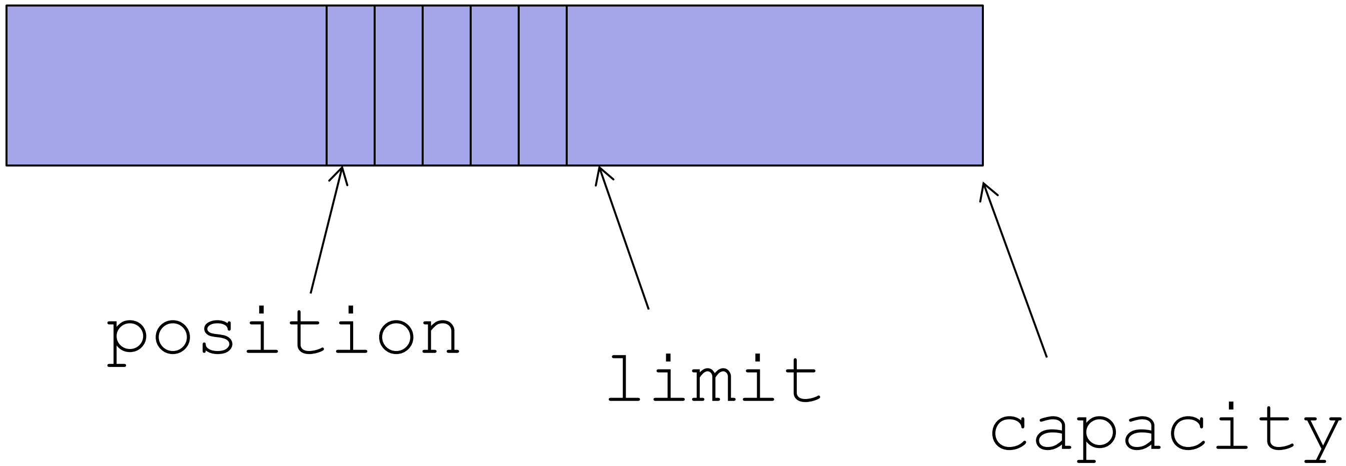
- `Buffer.clear(): position = 0; limit=capacity`

channel.read(Buffer)



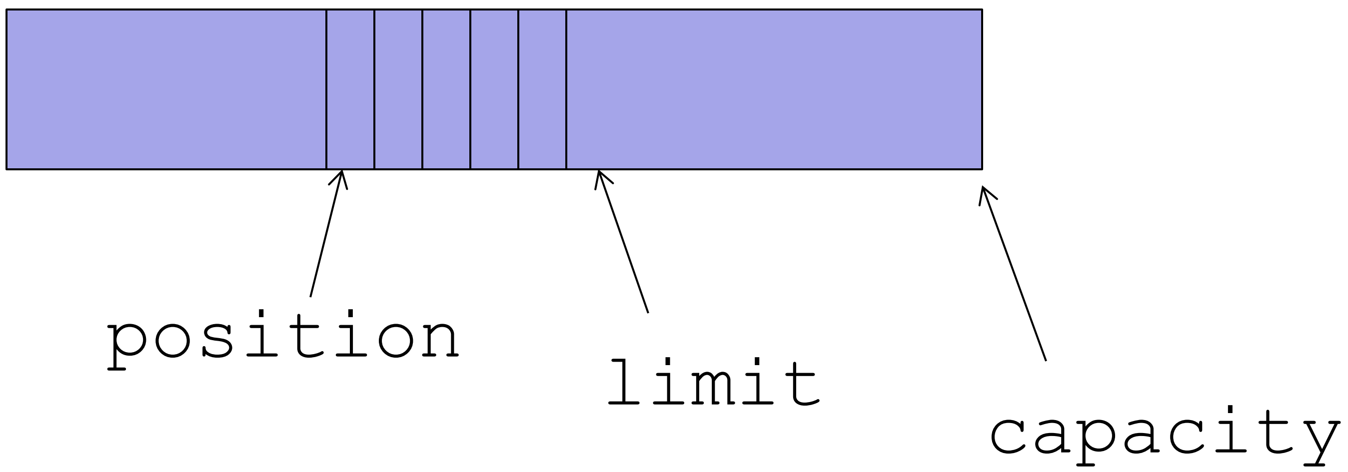
- Put data into Buffer, starting at `position`, not to reach `limit`

channel.write(Buffer)



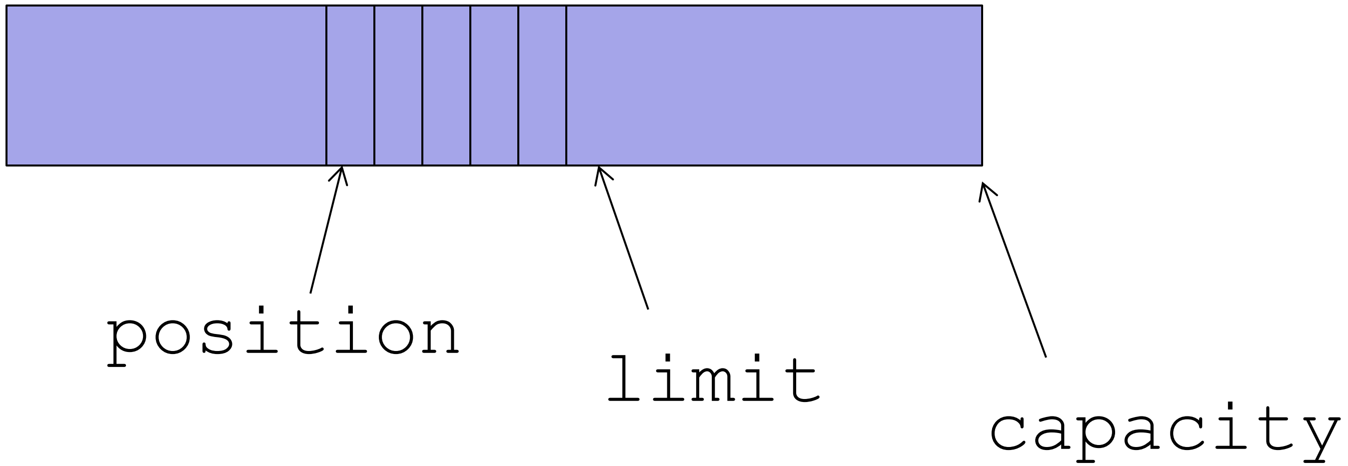
- Move data from Buffer to channel, starting at `position`, not to reach `limit`

Buffer.flip()



- Buffer.flip(): limit=position; position=0
- Why flip: used to switch from preparing data to output, e.g.,
  - buf.put(header); // add header data to buf
  - in.read(buf); // read in data and add to buf
  - buf.flip(); // prepare for write
  - out.write(buf);
- Typical pattern: read, flip, write

Buffer.compact()



- Move [position , limit) to 0
- Set position to limit-position, limit to capacity

```
// typical design pattern
buf.clear(); // Prepare buffer for use
for (;;) {
   if (in.read(buf) < 0 && !buf.hasRemaining())
      break; // No more bytes to transfer
   buf.flip();
   out.write(buf);
   buf.compact(); // In case of partial write
}
```

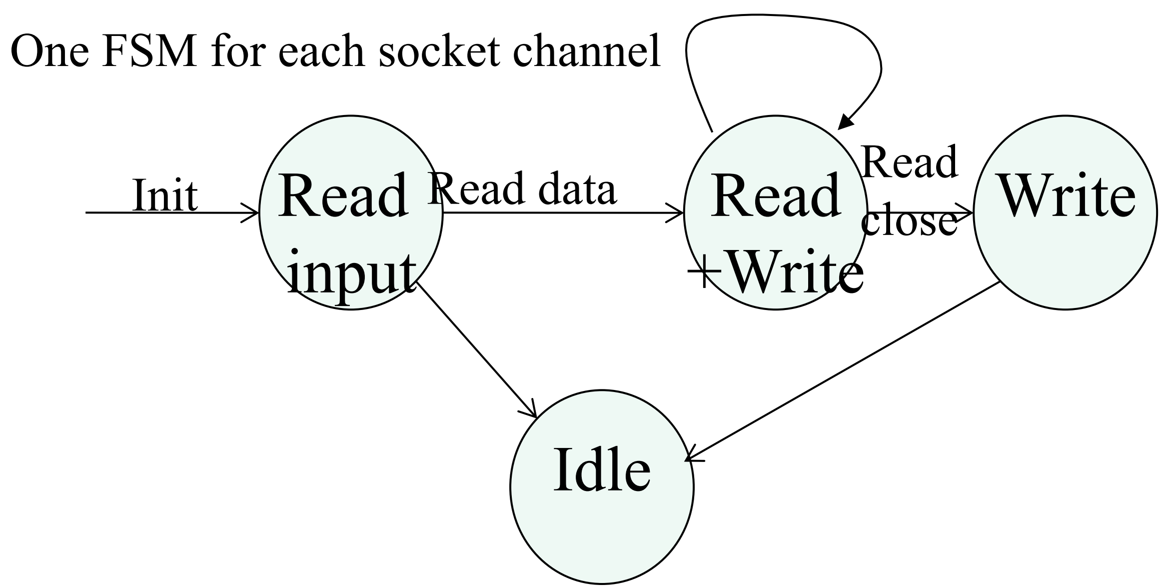
Example and Design Exercise

- See SelectEchoServer/v1/SelectEchoServer.java

Summary: Steps We Took to Refine the Echo Server

- Register READ for newly accepted connection
  - otherwise, no read events
- Register only READ, not WRITE
  - otherwise empty write
    - Imagine empty write with 10,000 sockets
- After read data, turn on write to enable echo output
  - otherwise no output
- After write, check if there is data remaining to write, if no, turn off write
  - otherwise, empty write calls
- After reading end of stream (read returns -1), turn off read interest (or better deregister)
- All above are state management!

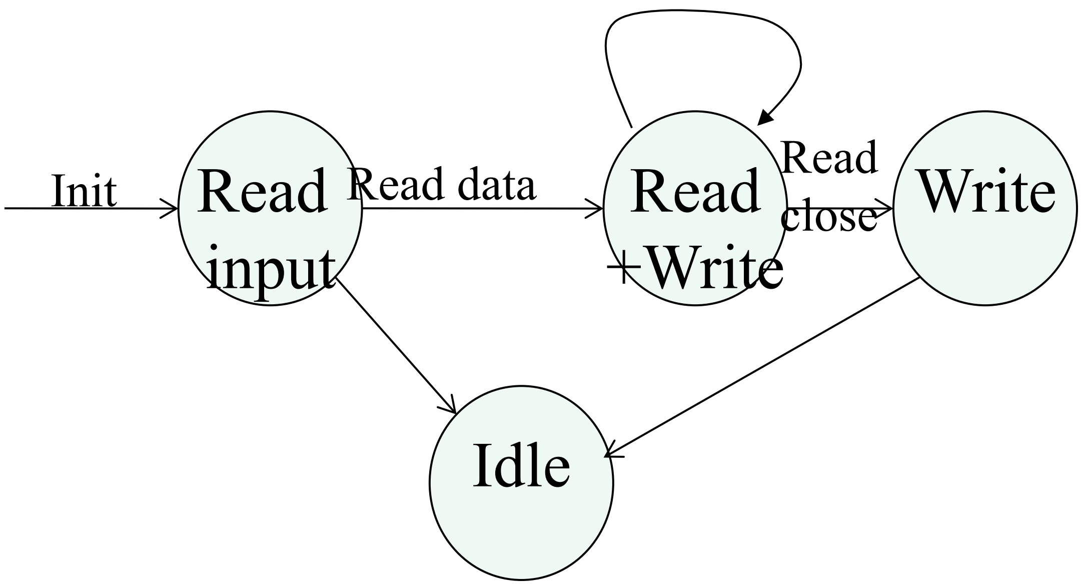
Finite-State Machine and Async Server

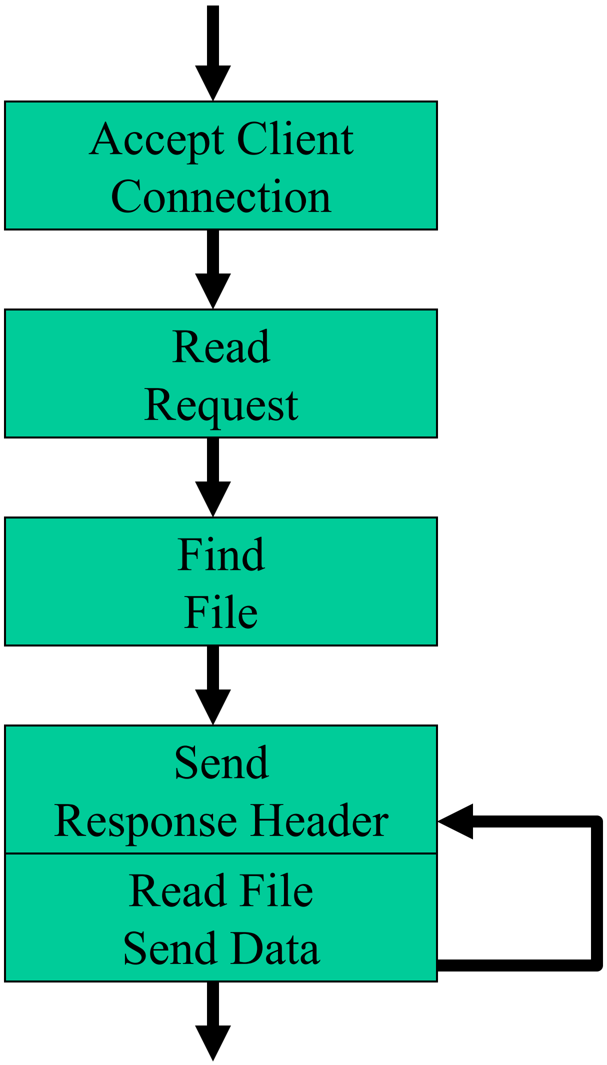


Not the most effective FSM, but an example.

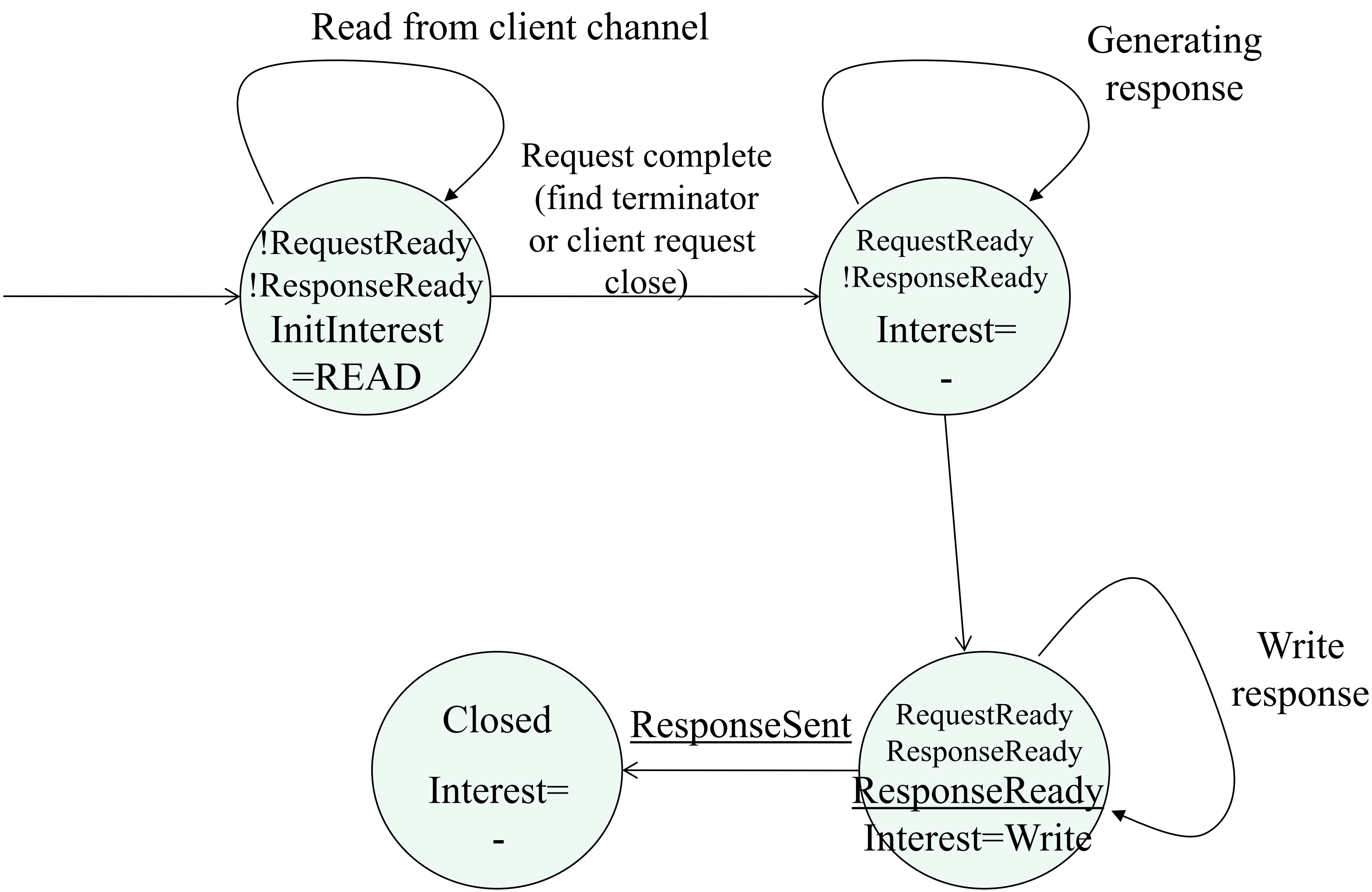
Finite-State Machine and Thread

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





A More Typical Finite State Machine



FSM and Reactive Programming

- Designing the FSM is key to non-blocking servers, and there can be multiple types of FSMs, to handle protocols correctly
  - 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

Toward More Generic Select Server Software Framework

- Non-blocking, select programming framework is among the more complex software systems, and we want to reuse the software as much as possible
  - E.g., consider a setting where a single server monitors multiple ports, with each port may run a different protocol
- Question: Which design of the EchoServer is not generic (i.e., reusable for different protocols)?

EchoServer Design Issues

- Fixed accept/read/write functions (handlers) are not general design
- PCB is customized for echo servers only

A More Extensible Dispatcher Design

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

```
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);
}
```

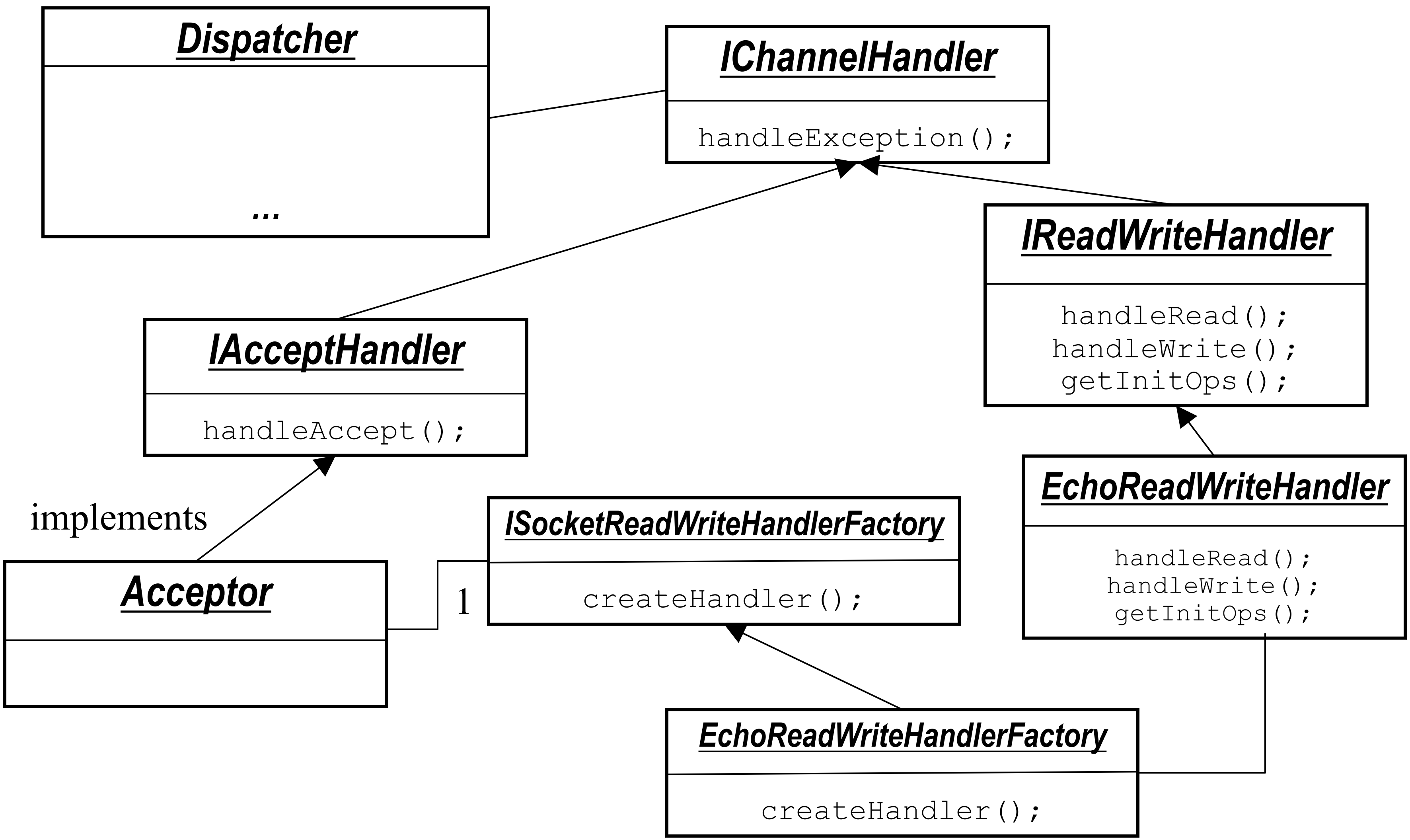
Handler Design: Acceptor

- What should an accept handler object know?
  - ServerSocketChannel (so that it can call accept)
    - Can be derived from SelectionKey in the call back
  - 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

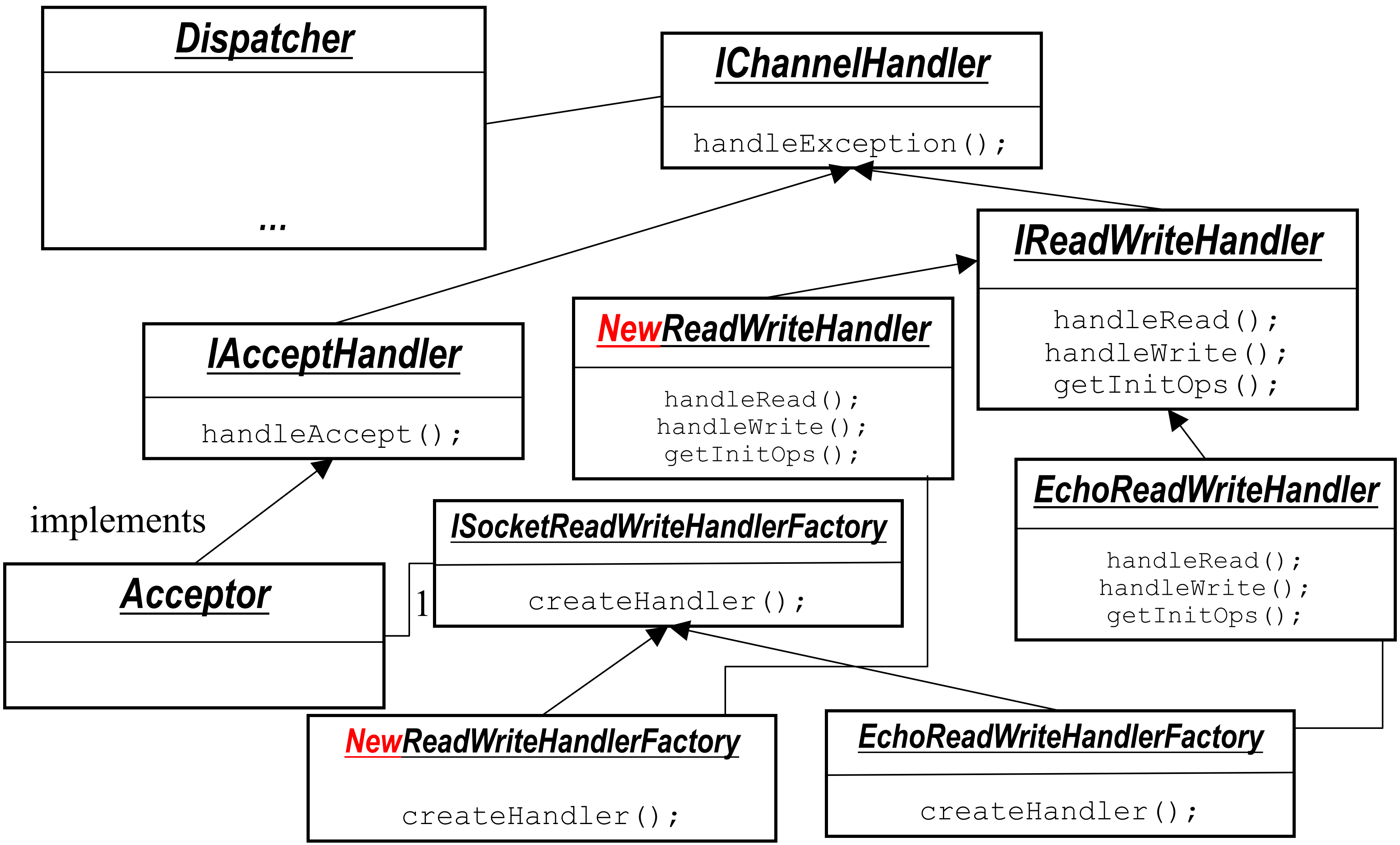
Handler Design: ReadWriteHandler

- 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



Class Diagram of SimpleNAIO



SimpleNAIO

- See SelectEchoServer/v2/*.java

Design Exercise

- In our current implementation (Server.java)

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

Can we simply switch 3 and 4?

Design Exercise to Understand Server Structure

- 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
}
```

Execute Commands by Dispatcher

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

Design Exercise to Understand Server Structure

- What if another thread wants to wait until a command is finished by the dispatcher thread?
  - AKA: How to block another thread until its command is executed 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.
  }
 }
```

Backup Slides

Another view

- Events obscure control flow
  - For programmers *and* tools

Threads

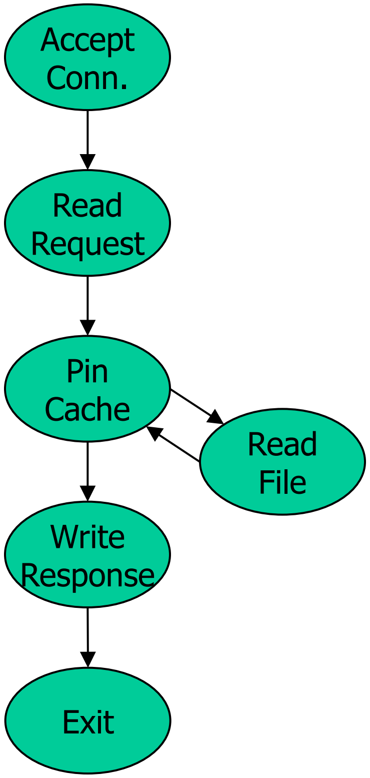
```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    read_request(&s);
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

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);
}
. . .
ExitHandlerr(struct session *s) {
    …;  unpin(&s);  free_session(s);  }
```

Web Server



[von Behren]

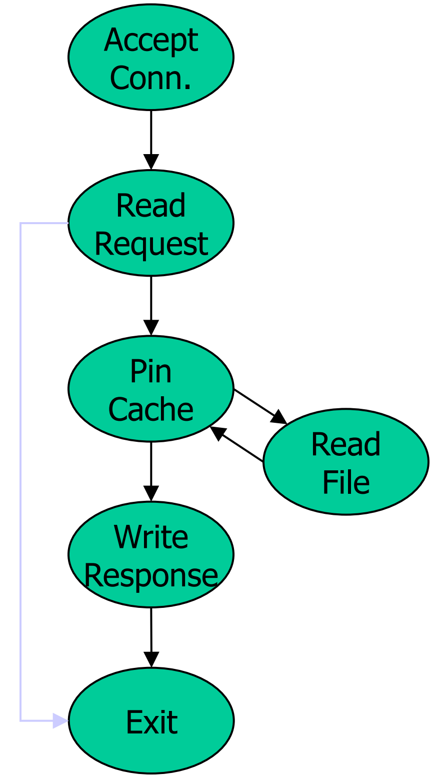
State Management

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

Threads	*Events*

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);
}
	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);
	}
	. . .
	ExitHandlerr(struct session *s) {
	    …; unpin(&s); free_session(s);
	}
	AcceptHandler(event e) {
	    struct session *s = new_session(e);
	    RequestHandler.enqueue(s); }

Web Server



[von Behren]