Network Applications: High-performance Server Design

Qiao Xiang

https://qiaoxiang.me/courses/cnns-xmuf22/index.shtml

10/27/2022

This deck of slides are heavily based on CPSC 433/533 at Yale University, by courtesy of Dr. Y. Richard Yang.

Outline

- Admin and recap
- High performance servers
  - Threaded design
    - Per-request thread
    - Thread pool
      - Busy wait
      - Wait/notify
  - Asynchronous design

Admin

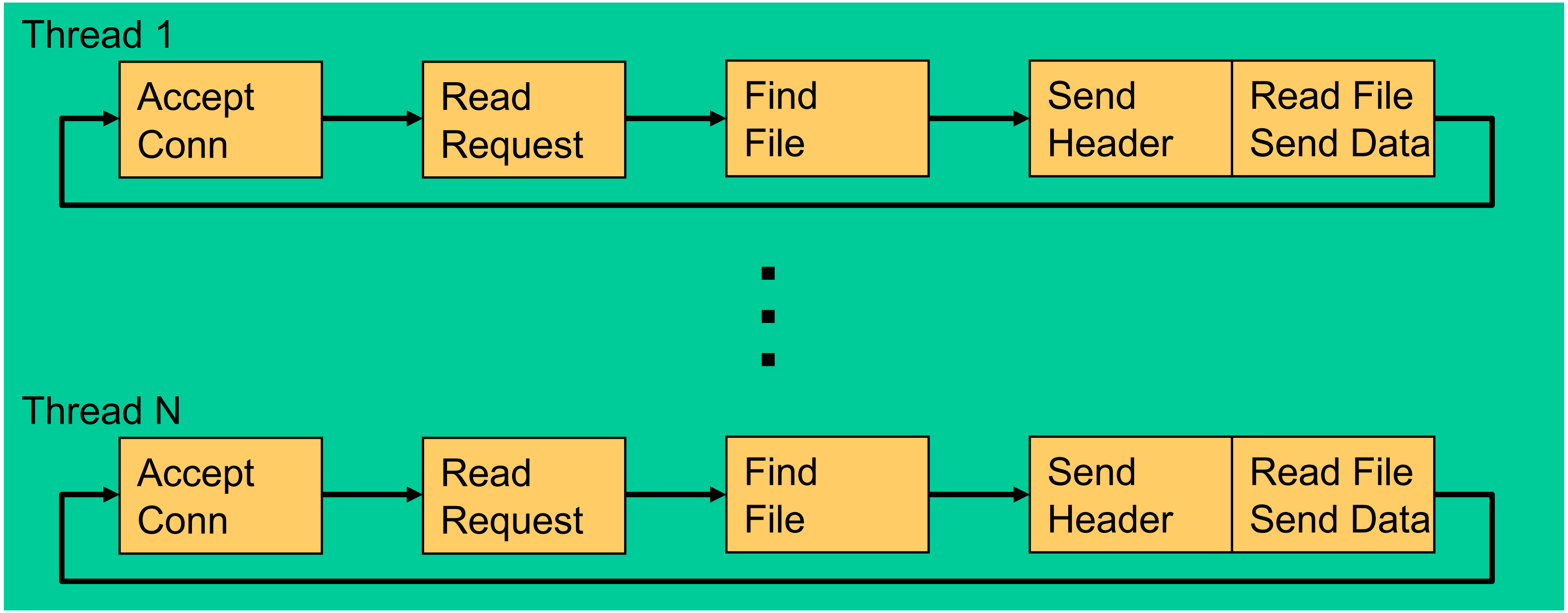
- Lab assignment 3 due on Nov. 8
- Date for exam 1?
  - Nov. 10 (2:30-4:10pm, lab class)

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 main thread coordinate on the shared queue
      - polling (busy wait)
      - suspension: wait/notify

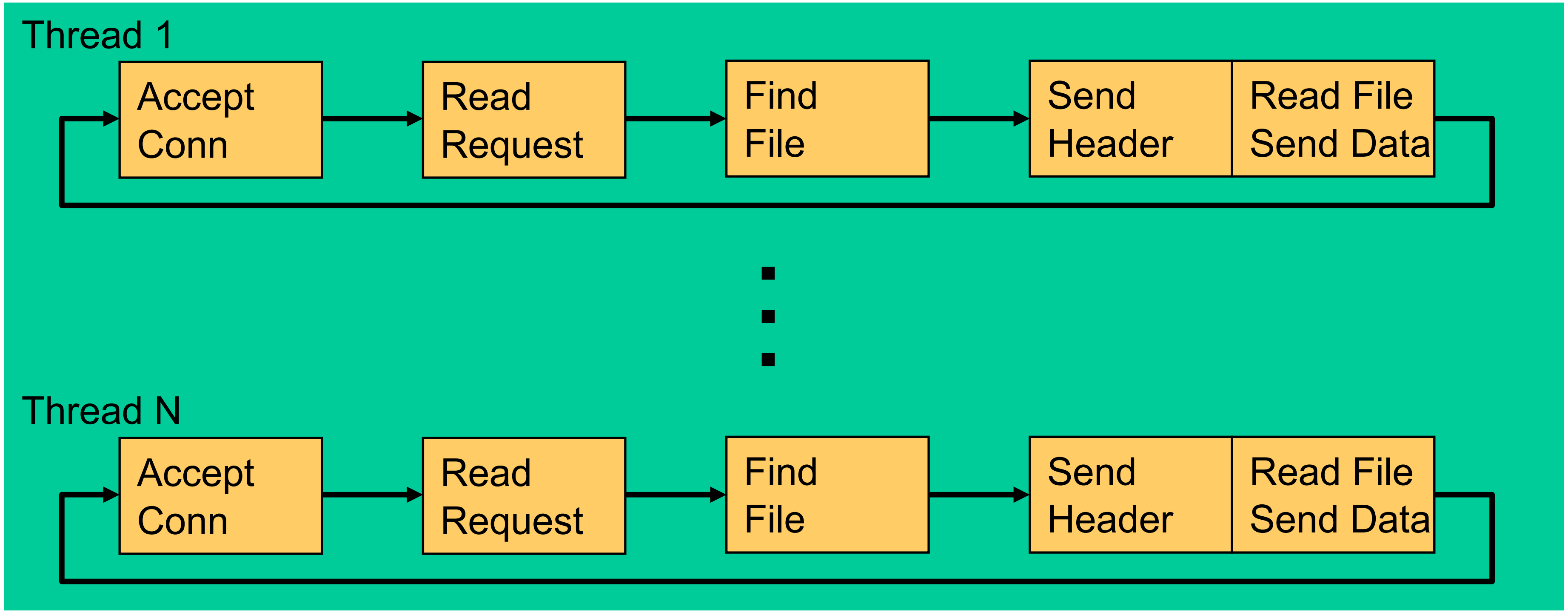
Summary: Thread-Based Network Server

- Multiple threads (execution sequences) offer multiple execution sequences => blocking causes only one thread being blocked
- Intuitive (sequential) programming model
- Shared address space simplifies optimizations



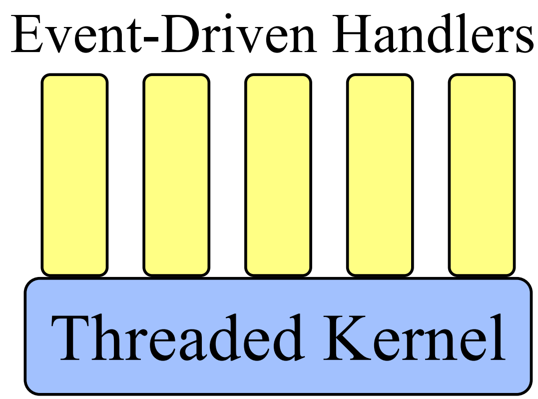
Summary: Thread-Based Network Server

- Thread creation overhead
- Thread synchronization overhead
  - Need to handle synchronization -> otherwise race condition
  - Handle synchronization -> Overhead, complexity (e.g., wait/notify, deadlock)
  - Thread size (how many threads) difficult to tune
- Still cannot handle well the large-number of long, idle connections problem (why?)



Should You Use Threads?

- Typically avoid threads for io
  - Use event-driven, not threads, for GUIs, servers, distributed systems.
- Use threads where true CPU concurrency is needed.
  - Where threads needed, isolate usage in threaded application kernel: keep most of code single-threaded.



[Ousterhout 1995]

Outline

- Admin and recap
- High performance servers
  - Threaded design
    - Per-request thread
    - Thread pool
      - Busy wait
      - Wait/notify
  - *Select-multiplexing server design*

Big Picture: Built on top of Lower-Layer OS Services/Abstractions

- Blocking IO
  - if not ready, block calling thread
  - get data, copy to user space;
- Non-blocking IO (set socket NON_BLOCK) stream
  - return error if not ready (EWOULDBLOCK)
  - after ready, call, OS copy
- Selector (channel) IO [Java NIO; Linux epoll; FreeBSD/Mac kqueue]
  - monitors multiple IO descriptors
- Async IO (Java 7 aio; Linux 2.5 first and then 2.6)
  - aio_read() // after copy to user space
- DMA based (later in course)

server

128.36.232.5

128.36.230.2

TCP socket space

state: listening
address: {*.**6789**, *:*}
completed connection queue: C1; C2
sendbuf:
recvbuf:

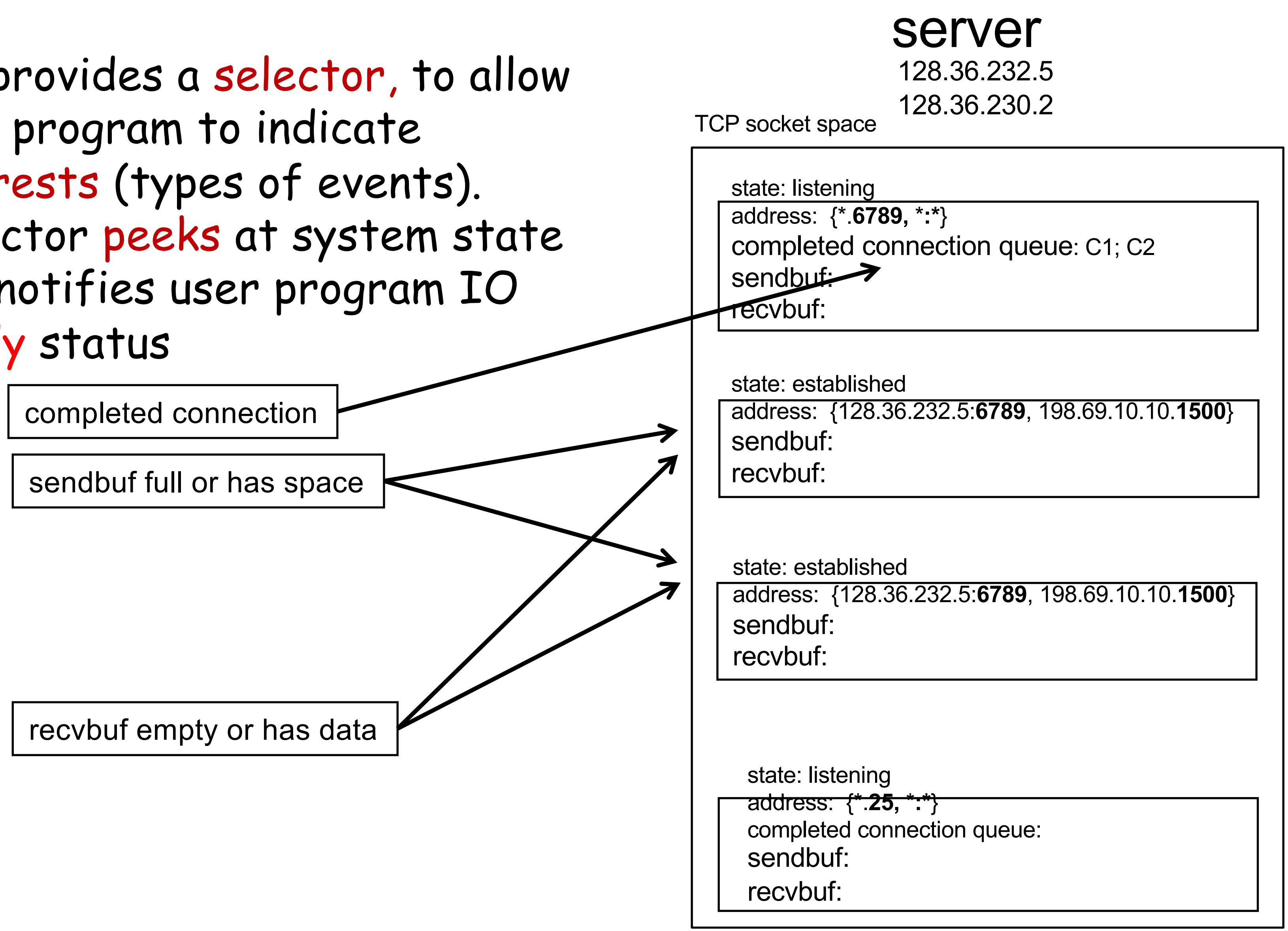
state: established
address: {128.36.232.5:**6789**, 198.69.10.10.**1500**}
sendbuf:
recvbuf:

state: established
address: {128.36.232.5:**6789**, 198.69.10.10.**1500**}
sendbuf:
recvbuf:

state: listening
address: {*.**25**, *:*}
completed connection queue:
sendbuf:
recvbuf:

Selector Multiplexing Basic Idea

- OS provides a selector, to allow user program to indicate interests (types of events). Selector peeks at system state and notifies user program IO ready status



Background: Linux epoll System Calls

- "… monitoring multiple files to see if IO is possible on any of them…" -- man 7 epoll
- Three basic system calls
  - epoll_create1(2) – create new epoll instance
  - epoll_ctl(2) – manage file descriptors regarding the interested-list
  - epoll_wait(2) – main workhorse, block tasks until IO becomes available
- See SelectEchoServer/epoll_examples.c

Core data structure

```
typedef union epoll_data {
    void        *ptr;
    int          fd;
    uint32_t     u32;
    uint64_t     u64;
} epoll_data_t;

struct epoll_event {
    uint32_t     events;      /* Epoll events */
    epoll_data_t data;        /* User data variable */
};
```

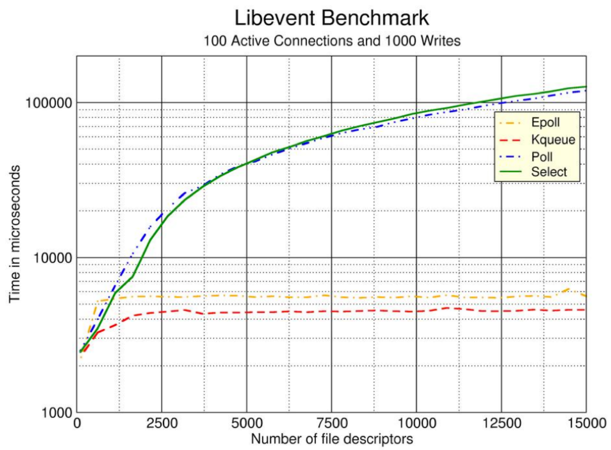
The *data* member of the *epoll_event* structure specifies data that the kernel should save and then return (via epoll_wait(2)) when this file descriptor becomes ready.

https://events19.linuxfoundation.org/wp-content/uploads/2018/07/dbueso-oss-japan19.pdf

Background: Linux epoll Internal

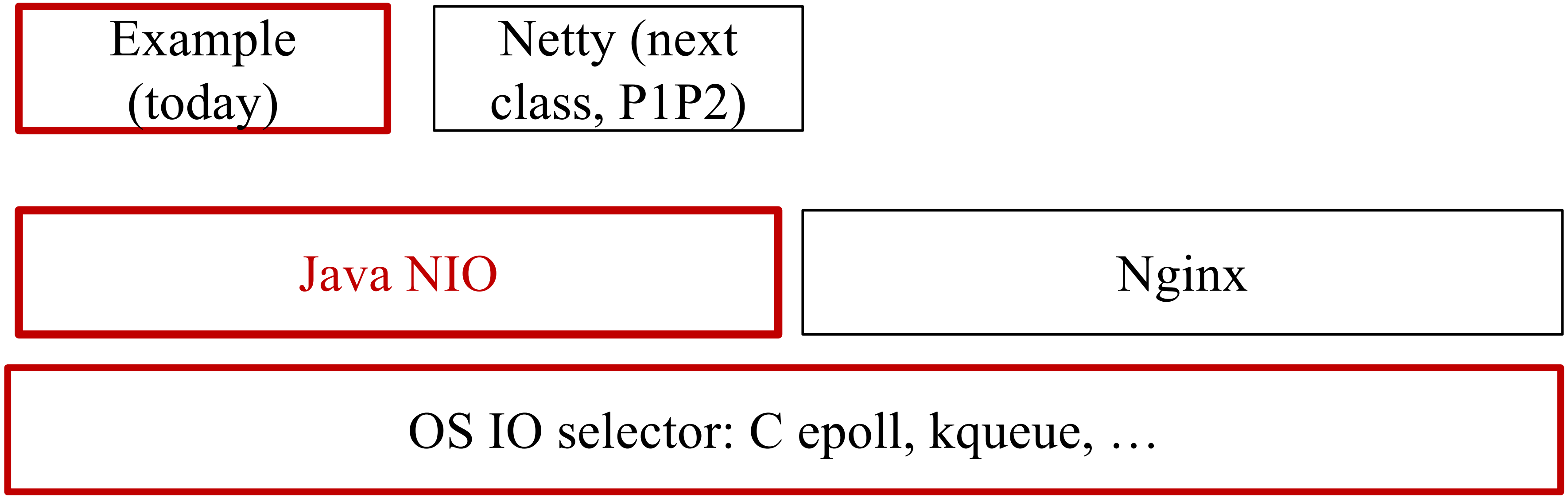
- Before epoll, select/poll is "stateless" but then need O(n) complexity; epoll separates setup and waiting phrases to reach O(n_ready)

- Details see: https://man7.org/linux/man-pages/man7/epoll.7.html



https://events19.linuxfoundation.org/wp-content/uploads/2018/07/dbueso-oss-japan19.pdf

Big Picture



Example (today)
Netty (next class, P1P2)
Java NIO
Nginx
OS IO selector: C epoll, kqueue, …

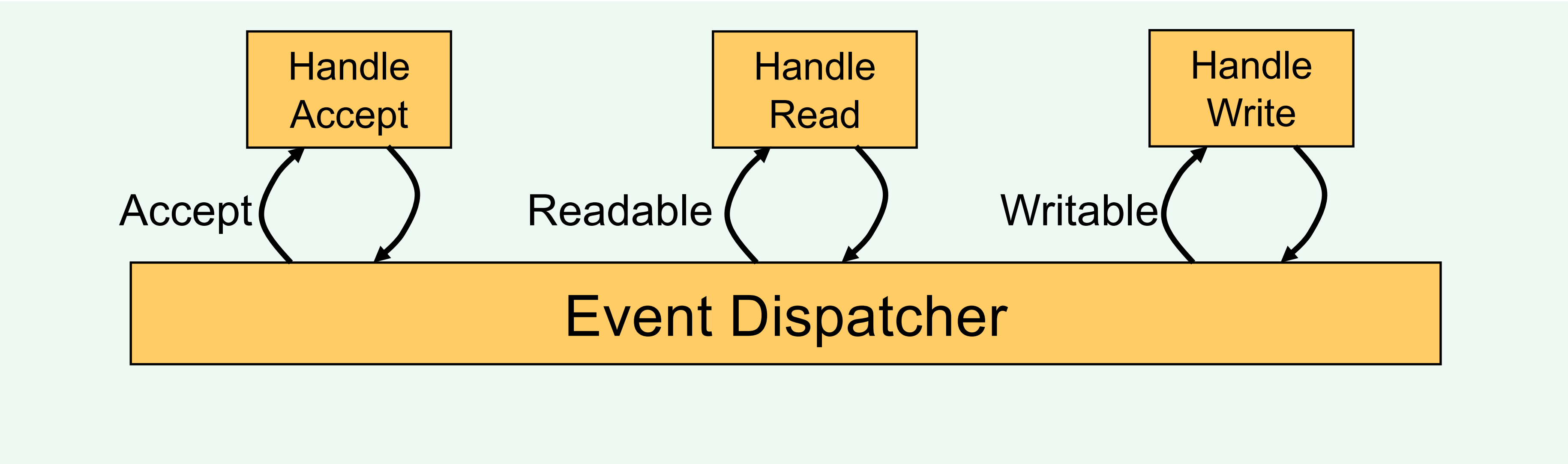
Basic Idea: Asynchronous Initiation and Callback

- 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

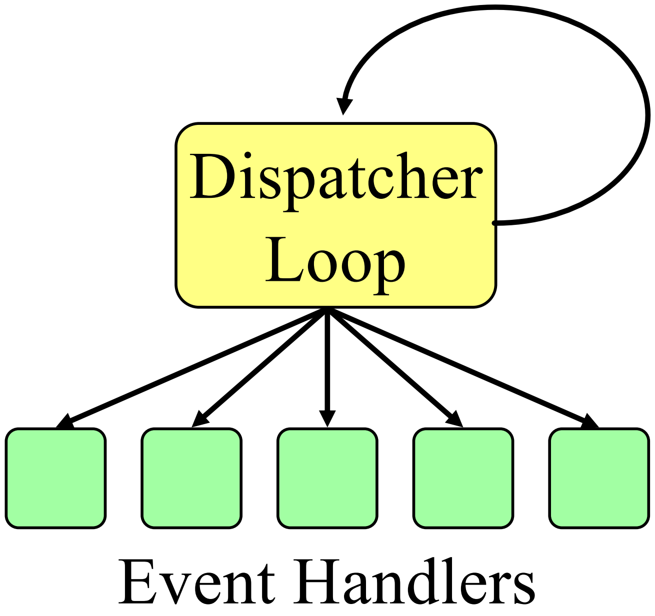
- We focus more on multiplexed, reactive design

Multiplexed, Reactive Server Architecture



- Program registers events (e.g., acceptable, readable, writable) to be monitored and a handler to call when an event is ready
- An infinite dispatcher loop:
  - Dispatcher asks OS to check if any ready event
  - Dispatcher calls (multiplexes) the registered handler of each ready event/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:
  - Channels: represent connections to entities capable of performing I/O operations;
  - Selectors and selection keys: selection facilities;
  - Buffers: containers for data.

- More details see https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html

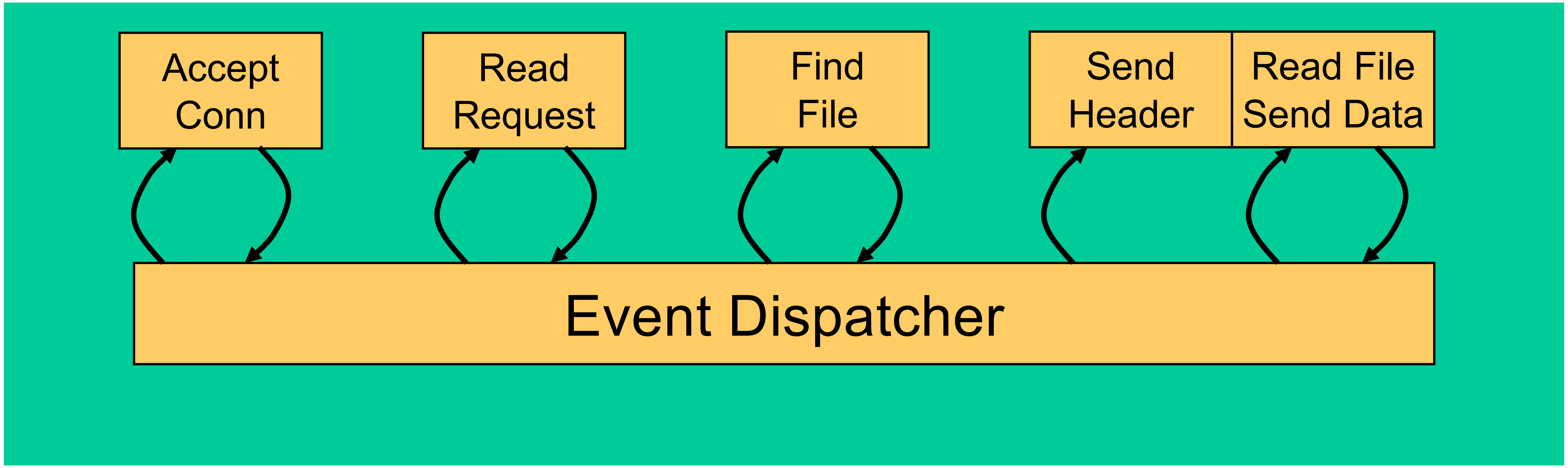
Multiplexed (Selectable), Non-Blocking Channels

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 (why?)



Selector and Registration

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

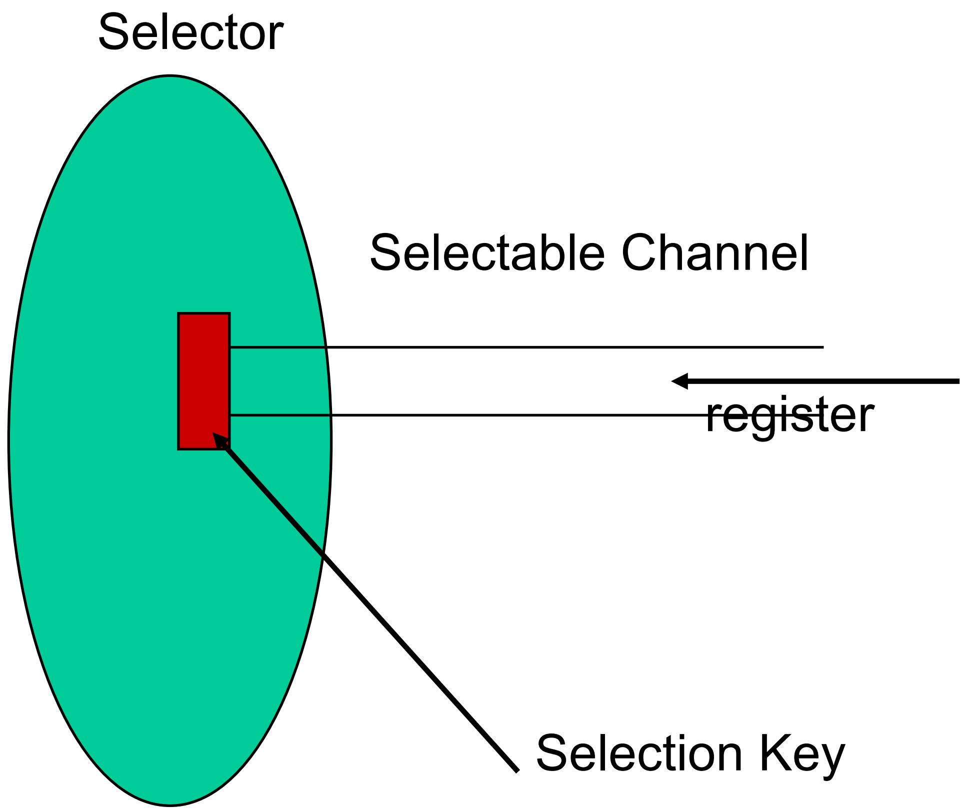
- The registration returns an object called a `SelectionKey`:

```
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

`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

Dispatcher using Select

```
while (true)  {
  - selector.select()
  - Set readyKeys = selector.selectedKeys();

  - foreach key in readyKeys {
       switch event type of key:
        accept: call accept handler
        readable: call read handler
        writable: call write handler
    }
}
```

I/O in Java: ByteBuffer

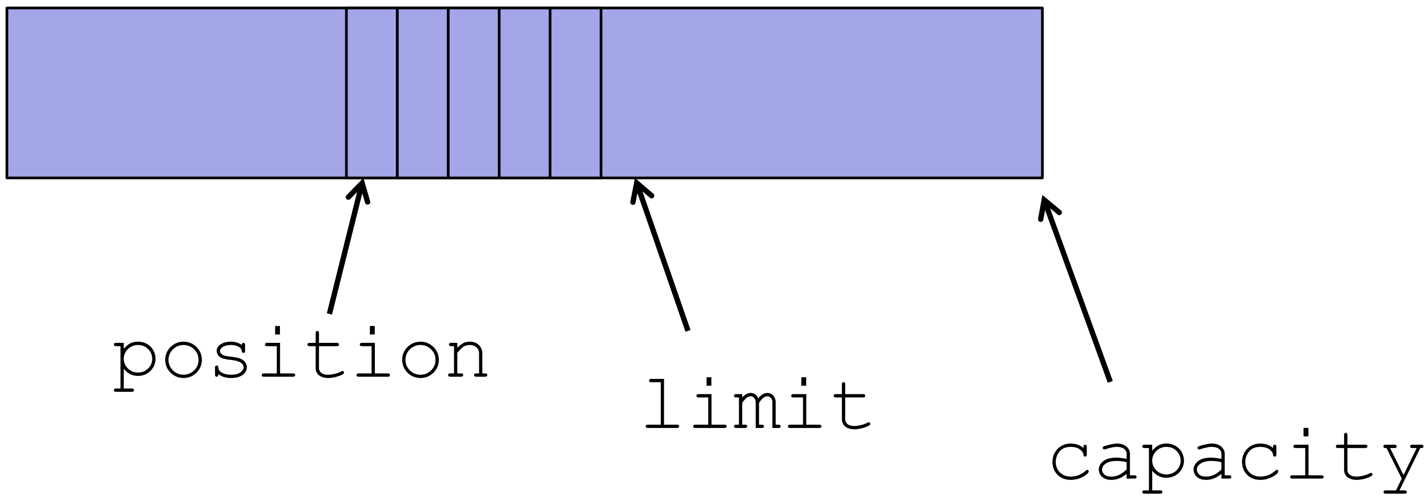
- Java SelectableChannels typically use ByteBuffer for read and write
  - channel.read(byteBuffer);
  - channel.write(byteBuffer);

- ByteBuffer is a powerful class that can be used for both read and write
- It is derived from the class Buffer
- All reasonable network server design should have a good buffer design

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	' '

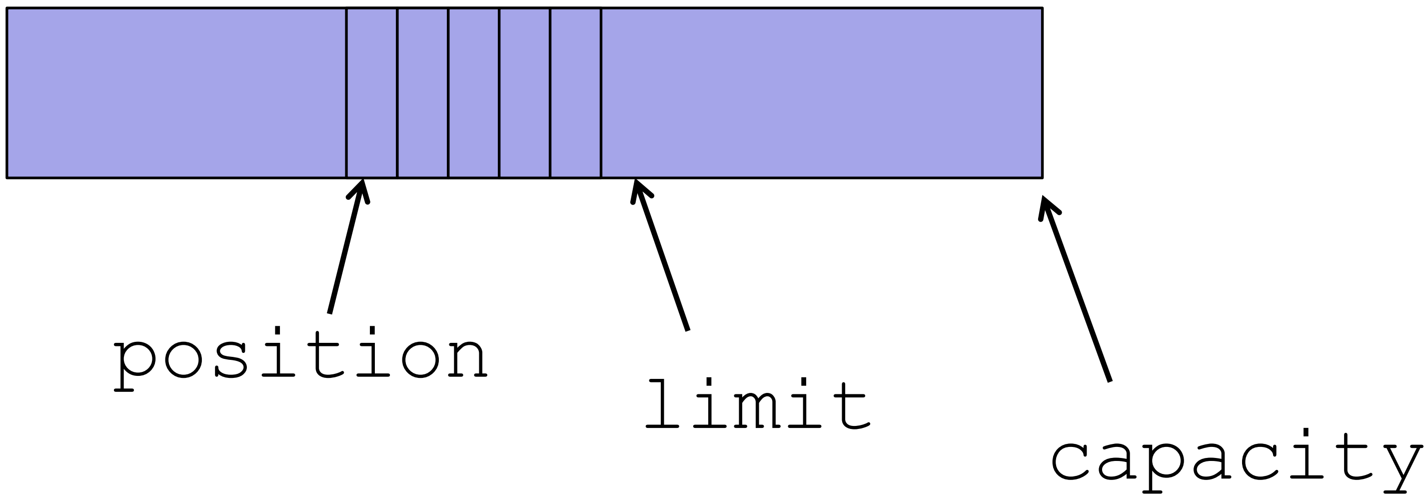
Buffer (relative index)



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

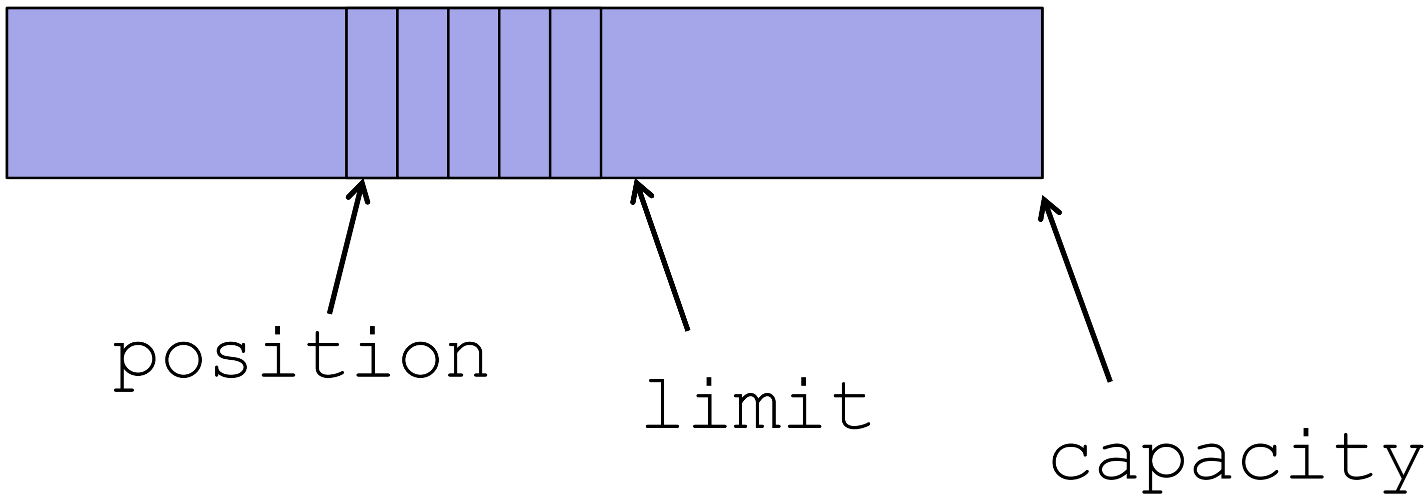
- `Buffer.clear(): position = 0; limit=capacity`

channel.read(Buffer)



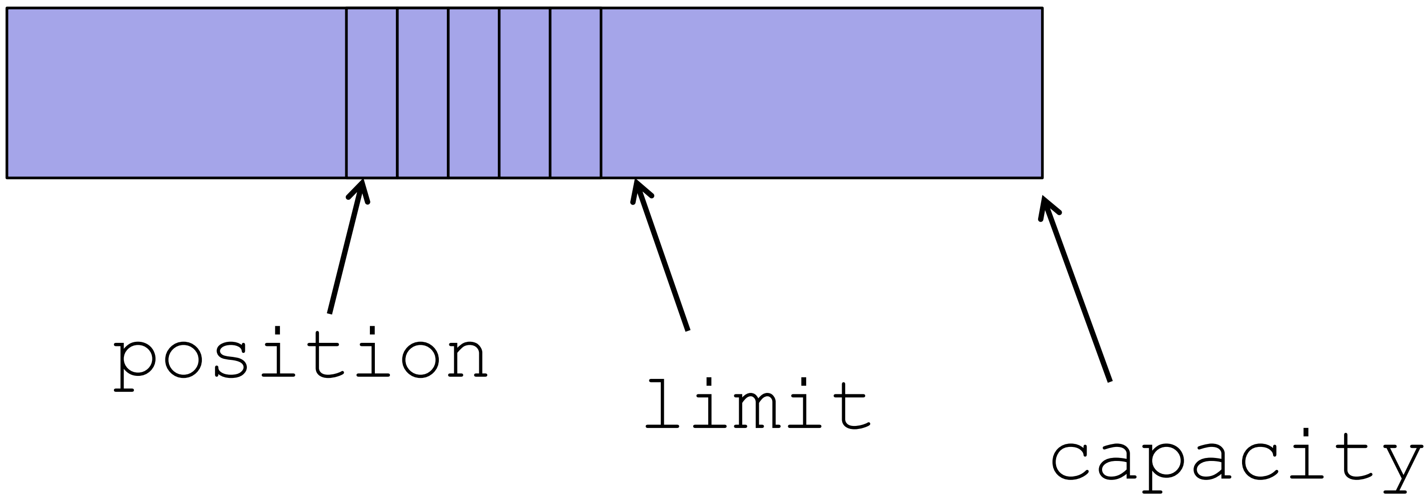
- Put data into Buffer, starting at `position`, not to reach `limit`

channel.write(Buffer)



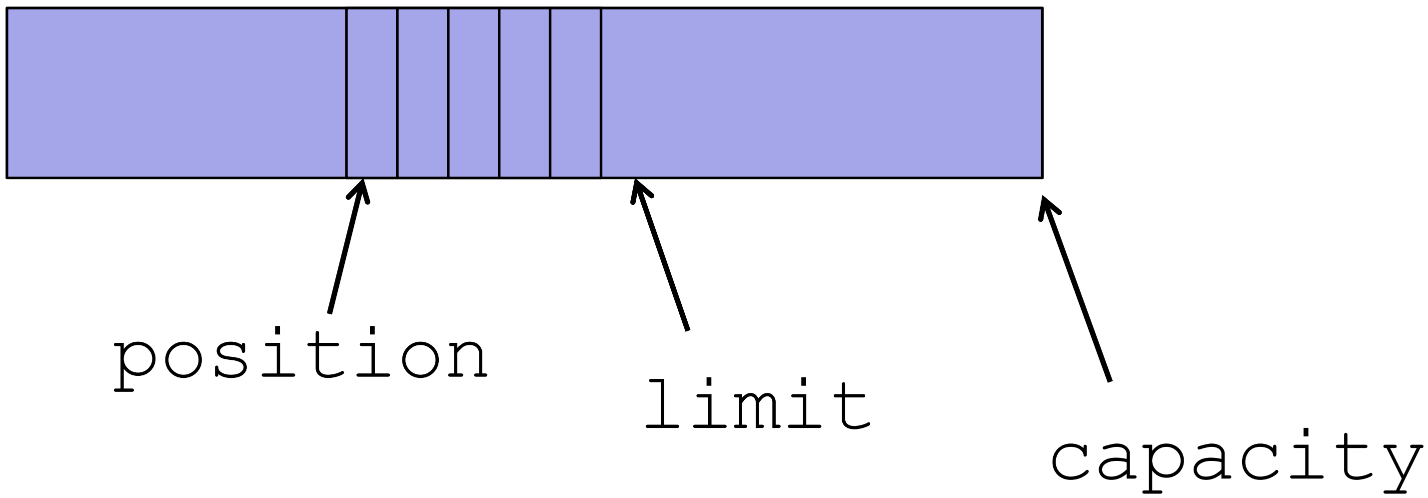
- Move data from Buffer to channel, starting at `position`, not to reach `limit`

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

- See SelectEchoServer/v1-2/SelectEchoServer.java

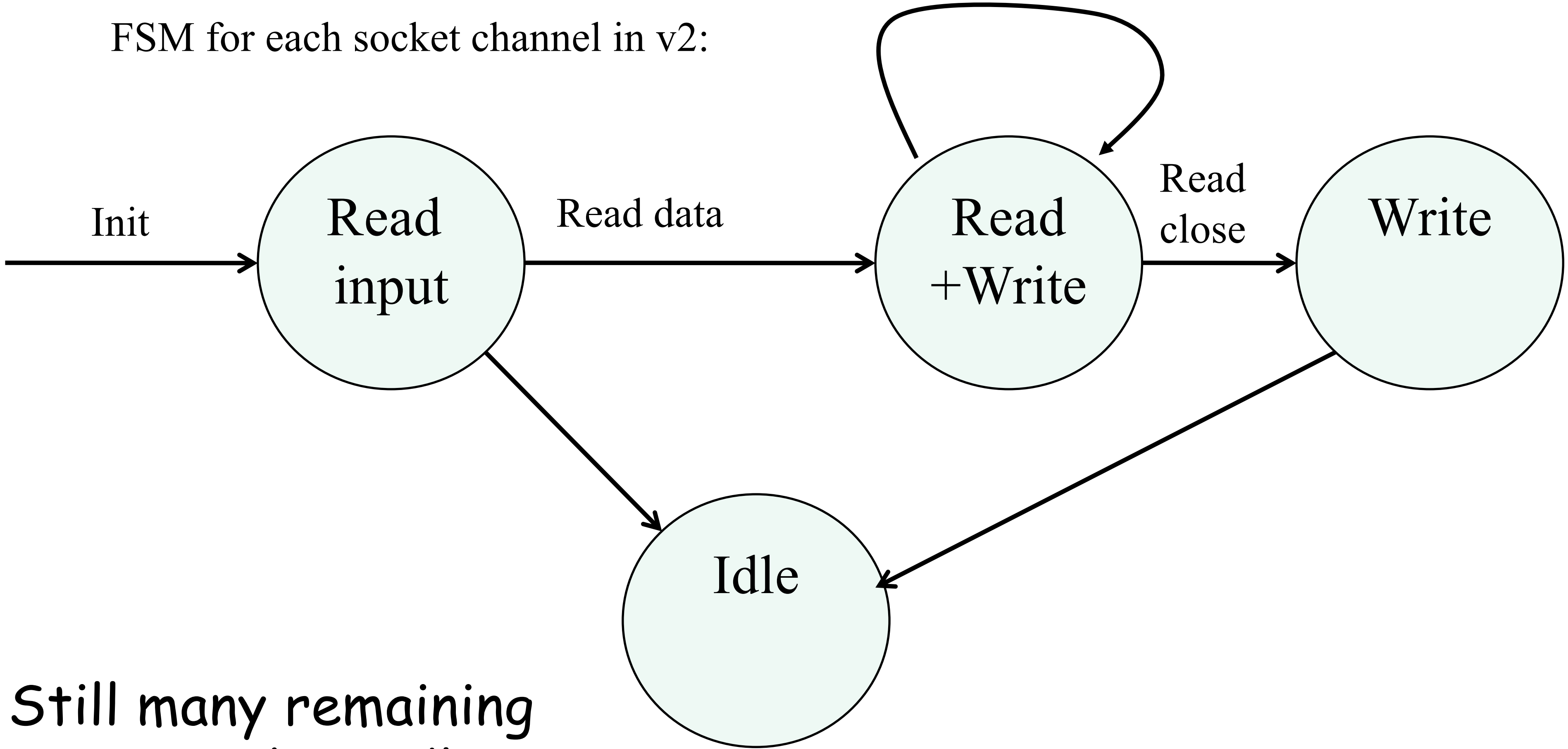
Problems of Echo Server v1

- Empty write: Callback to `handleWrite()` is unnecessary when nothing to write
  - Imagine empty write with 10,000 sockets
  - Solution: initially read only, later allow write

- `handleRead()` still reads after the client closes
  - Solution: after reading end of stream (read returns -1), deregister read interest for the channel

(Partial) Finite State Machine (FSM)

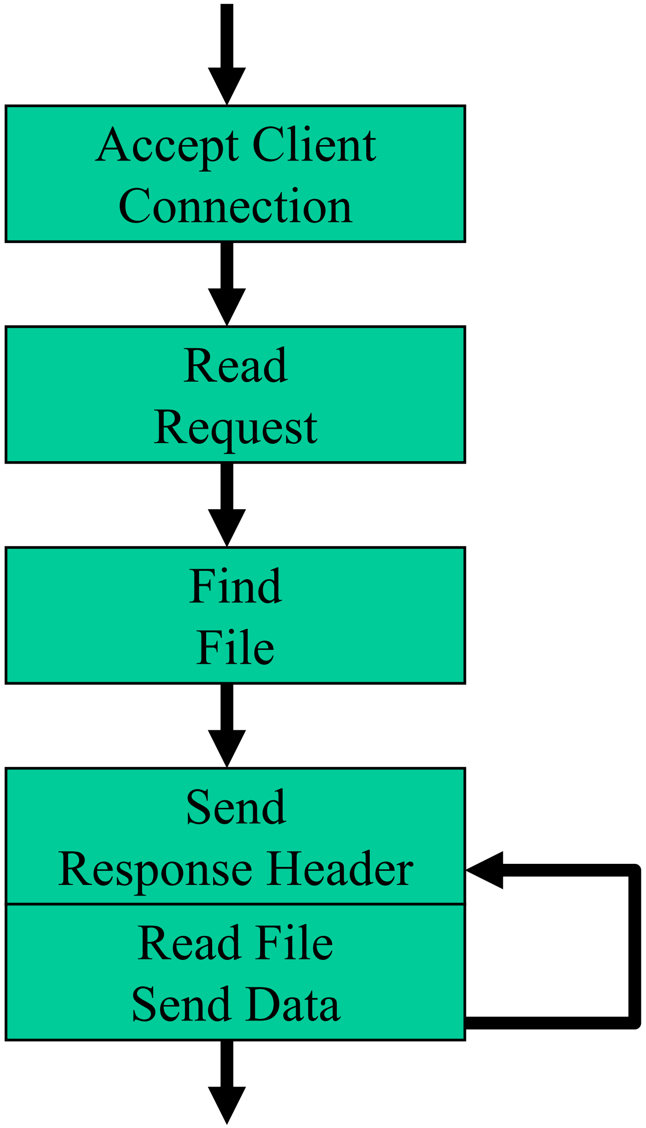
FSM for each socket channel in v2:



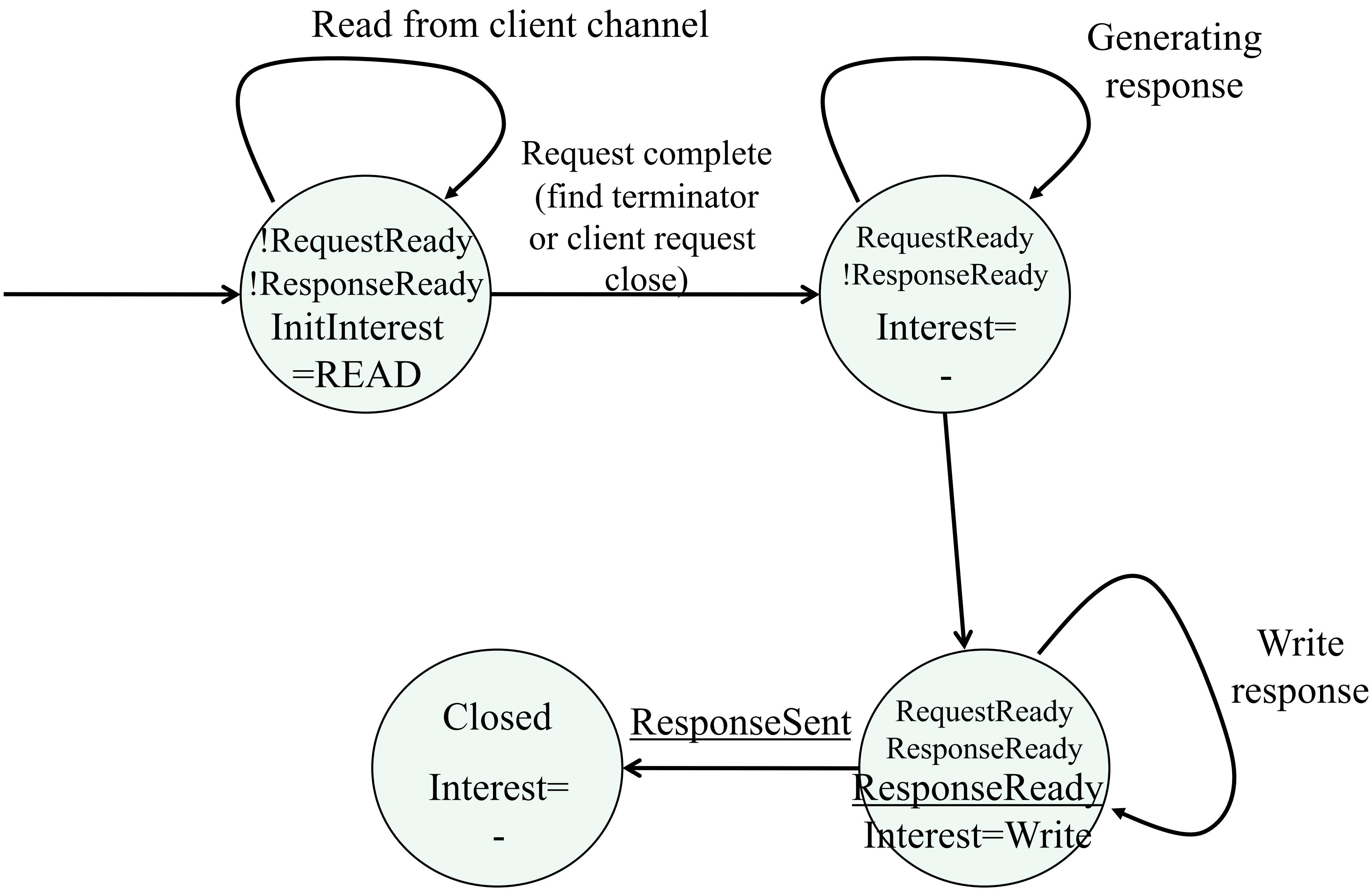
Still many remaining issues such as idle instead of close

Finite-State Machine and Thread

- Why no need to introduce FSM for a thread version?
- One perspective
  - A selector io program turns a sequential thread program into a parallel program, with each instruction block being able to run in parallel
  - Thread releases each block only when it reaches the instruction
  - Selector FSM releases all blocks by default and hence need FSM to control



A More Typical Finite State Machine



FSM and Reactive Programming

- 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 General Server Framework

- Our example EchoServer is for a specific protocol

- A general non-blocking, reactive programming framework tries to introduce structure to allow substantial program reuse
  - Non-blocking 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
  - 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

```
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);
}
```

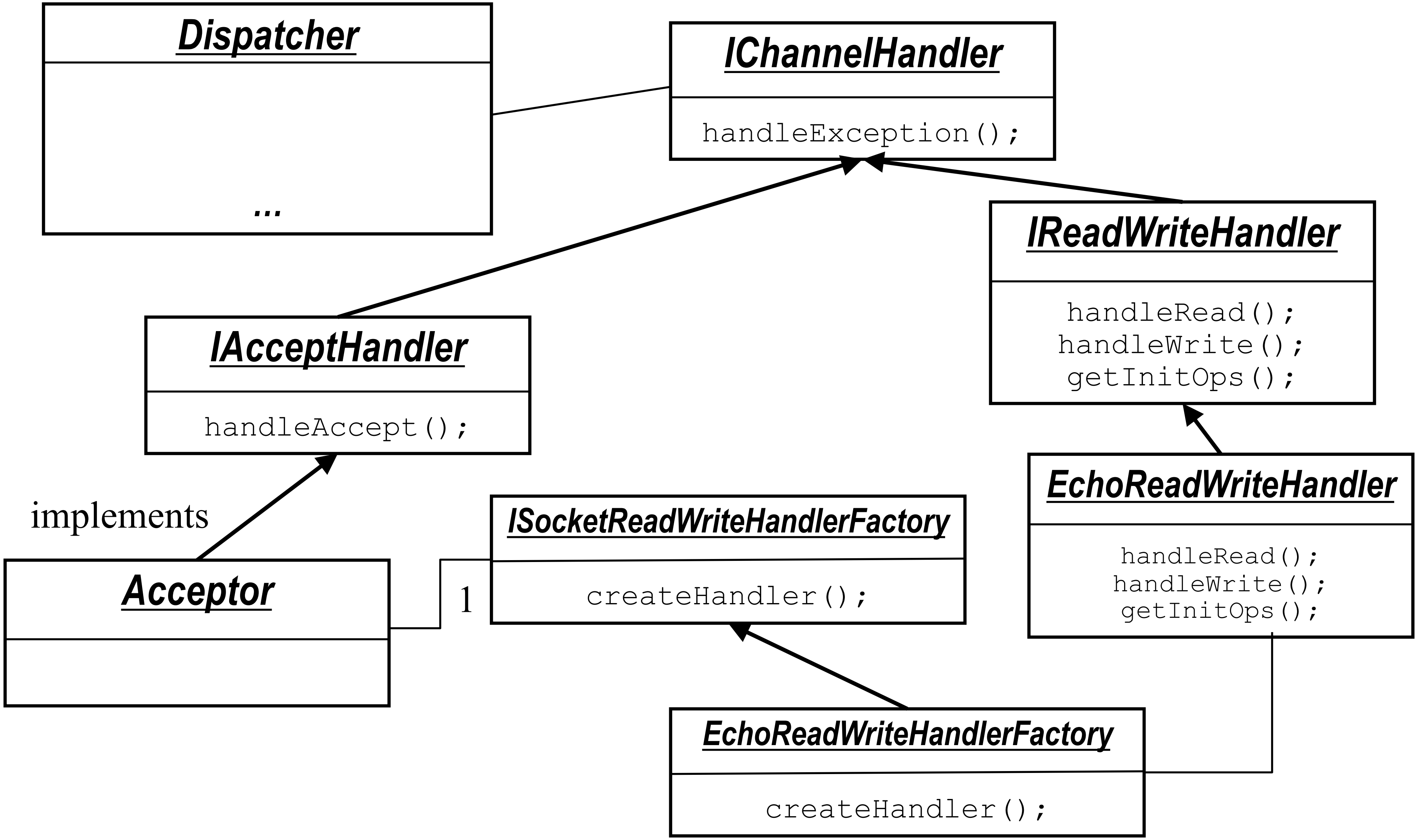
Handler Design: Acceptor

- 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

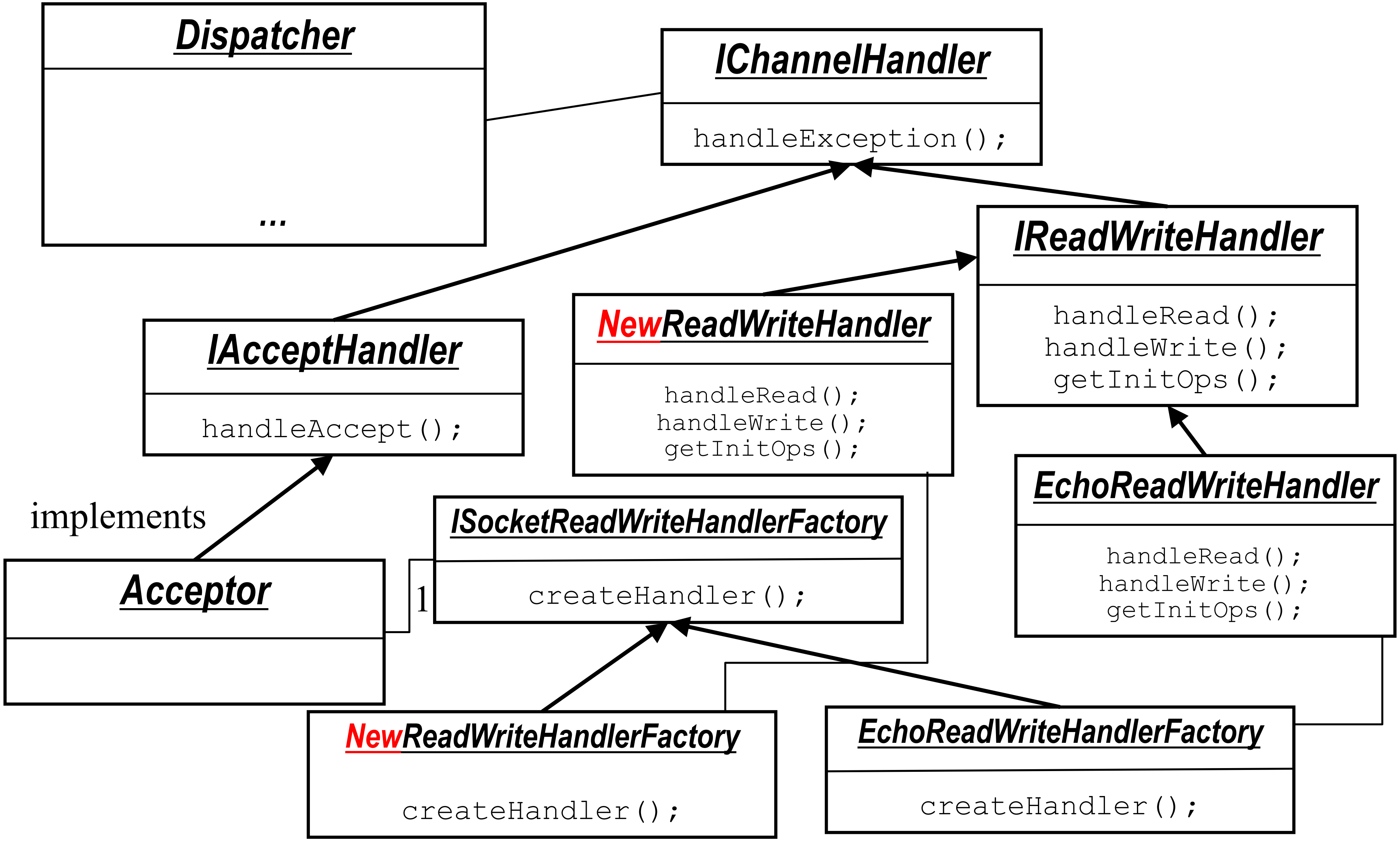
Handler Design: ReadWriteHandler

- 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/v3/*.java

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

```
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?

```
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.
  }
 }
```

Asynchronous Initiation and Callback: Basic Idea

- 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

Outline

- Admin and recap
- High performance servers
  - Thread design
  - Asynchronous design
    - Overview
    - Multiplexed (selected), reactive programming
    - *Asynchronous, proactive programming (asynchronous channel + future/completion handler)*

Asynchronous Channel using Future/Completion Handler

- Java 7 introduces ASynchronousServerSocketChannel and ASynchornousSocketChannel 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

Asynchronous I/O

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

- https://docs.oracle.com/javase/8/docs/api/java/nio/channels/package-summary.html

Example Async Calls

abstract Future <AsynchronousSocketChannel>	accept(): Accepts a connection.
abstract <A> void	accept(A attachment, CompletionHandler<AsynchronousSocketChannel,? super A> handler): Accepts a connection.

abstract Future<Integer>	read(ByteBuffer dst): Reads a sequence of bytes from this channel into the given buffer.
abstract <A> void	read(ByteBuffer[] dsts, int offset, int length, long timeout, TimeUnit unit, A attachment, CompletionHandler<Long,? super A> handler): Reads a sequence of bytes from this channel into a subsequence of the given buffers.

https://docs.oracle.com/javase/8/docs/api/java/nio/channels/AsynchronousServerSocketChannel.html

Future

```
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);
```

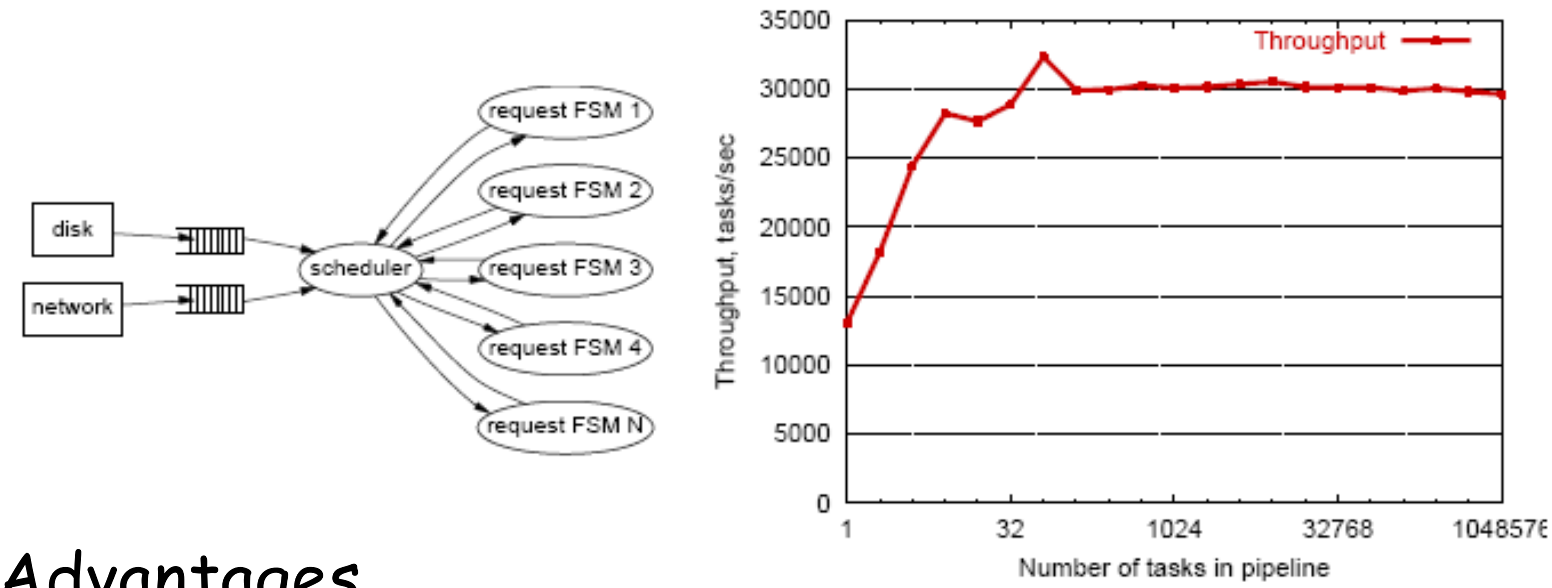
CompletionHandler

```
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



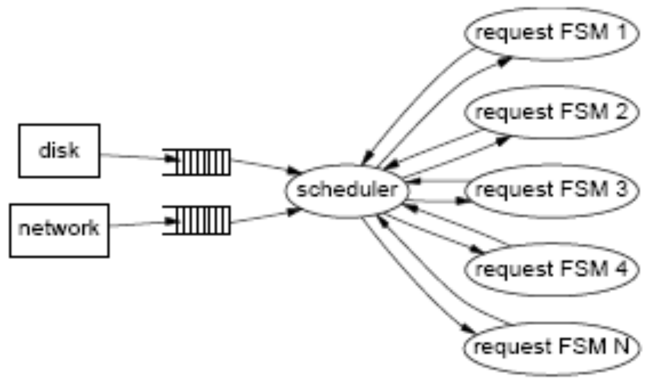
- Advantages
  - Single address space for ease of sharing
  - No synchronization/thread overhead
- Many examples: Click router, Flash web server, TP Monitors, NOX controller, Google Chrome (libevent), Dropbox (libevent), …
- Link https://javadoop.com/post/nio-and-aio provides a good introduction to Java NIO and AIO (in Chinese)

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



Summary: Architecture

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

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:
  - http://www.cs.wustl.edu/~schmidt/PDF/OOCP-tutorial4.pdf
  - http://www.stal.de/Downloads/ADC2004/pra03.pdf