# Network Applications: High-performance Server Design: Nonblocking Servers; Operational Analysis;

Qiao Xiang

https://qiaoxiang.me/courses/cnns-xmuf21/index.shtml

10/28/2021

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
  - Thread design
  - Asynchronous design
    - Overview
    - Multiplexed (selected), reactive programming
    - Asynchronous, proactive programming (asynchronous channel + future/completion handler)
  - Operational analysis
- Multi-servers

# Admin

- Lab assignment 2 due today

- Midterm exam date and location:
  - Nov. 11, 4:40-6:20pm, G-103

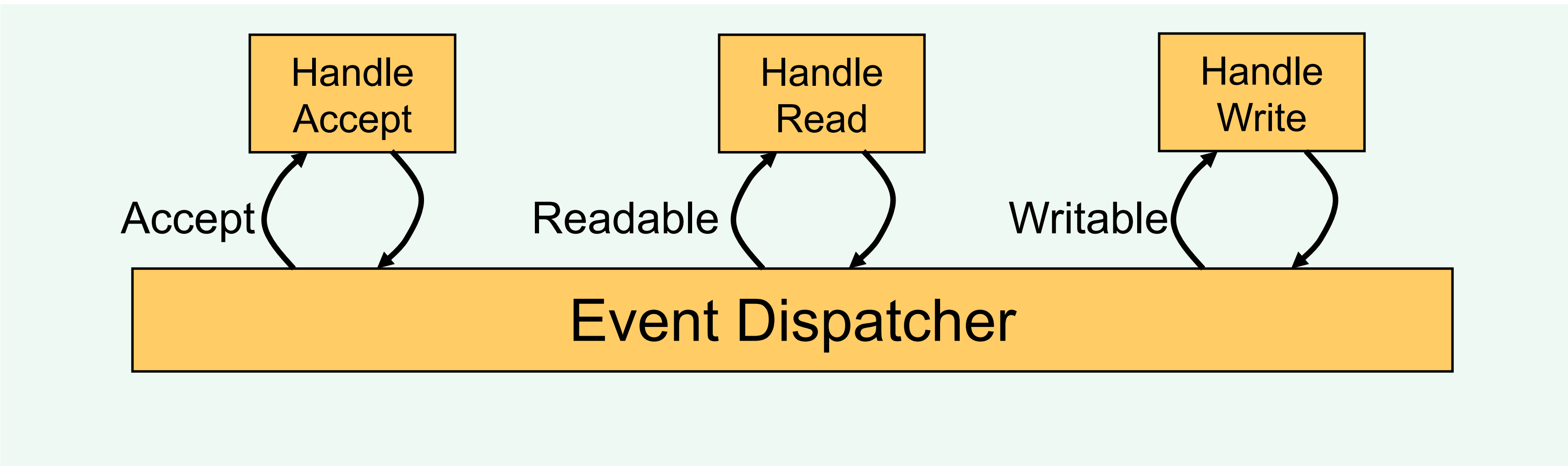
# Recap: Thread-Based Network Servers

- 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

# Recap: Program Correctness Analysis

- Safety
  - consistency
  - 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 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

# Recap: 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

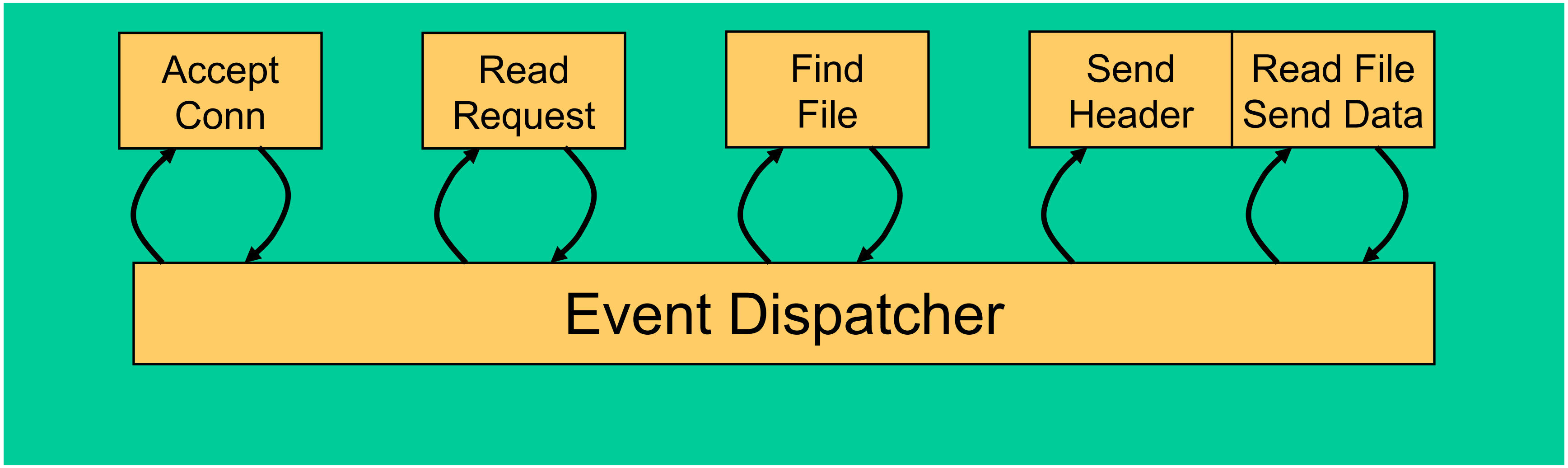
# Multiplexed (Selectable), Non-Blocking Channels

| | |
|---|---|
| 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

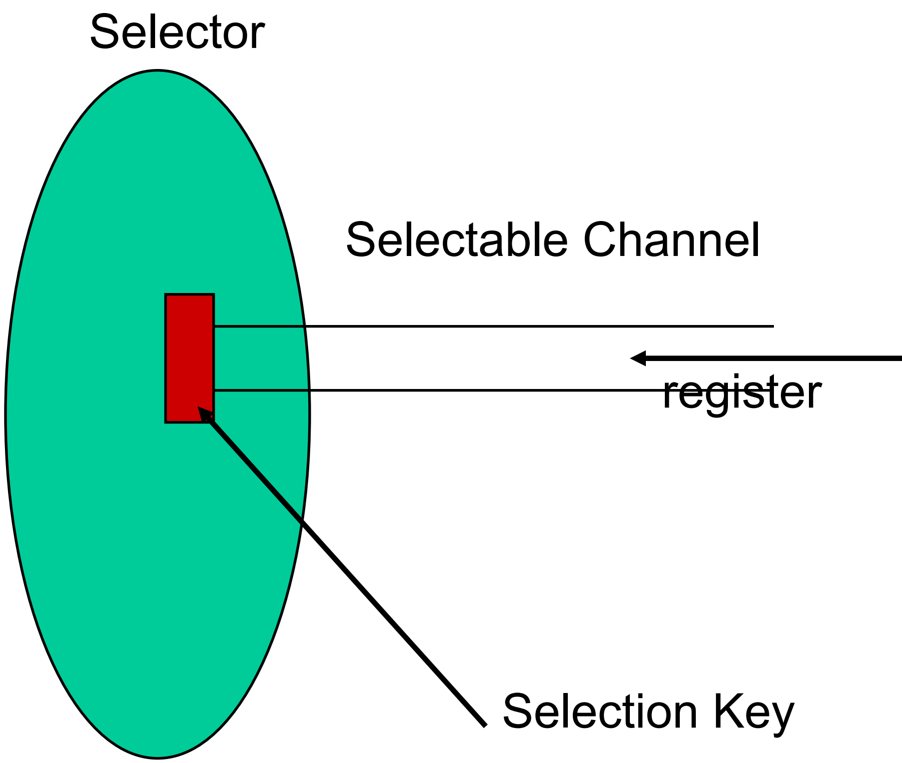
- The registration returns an object called a `SelectionKey`:

```
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

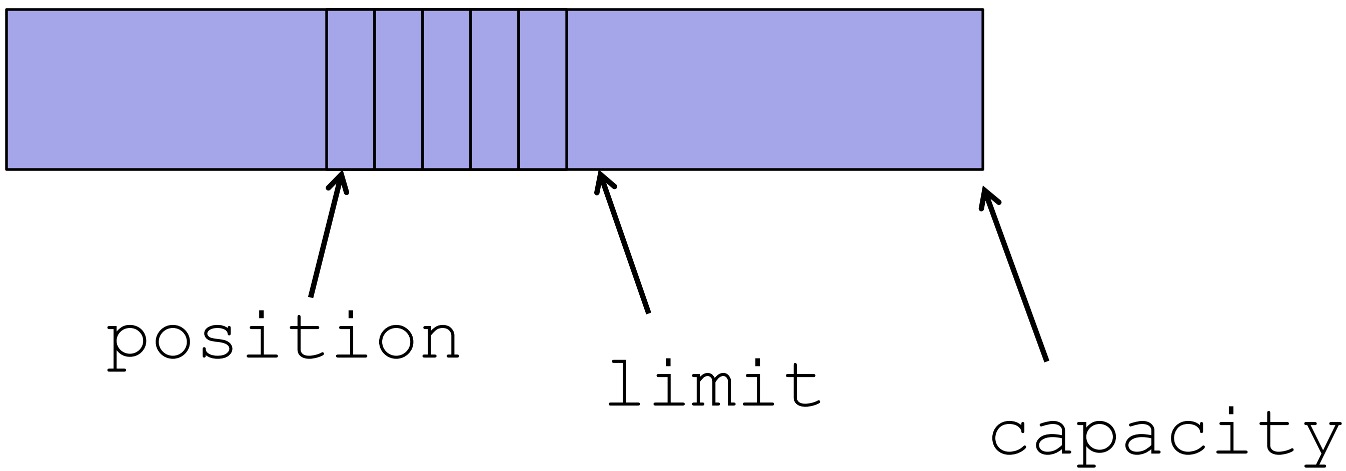
```
while (true)  {
  - selector.select()
  - Set readyKeys = selector.selectedKeys();

  - 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

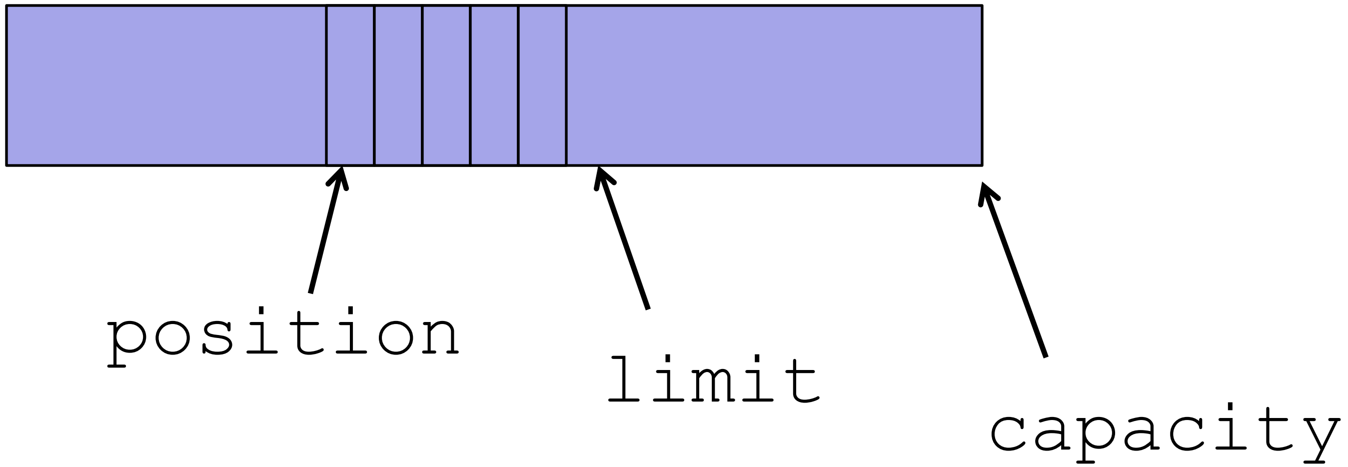
# Buffer (relative index)




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

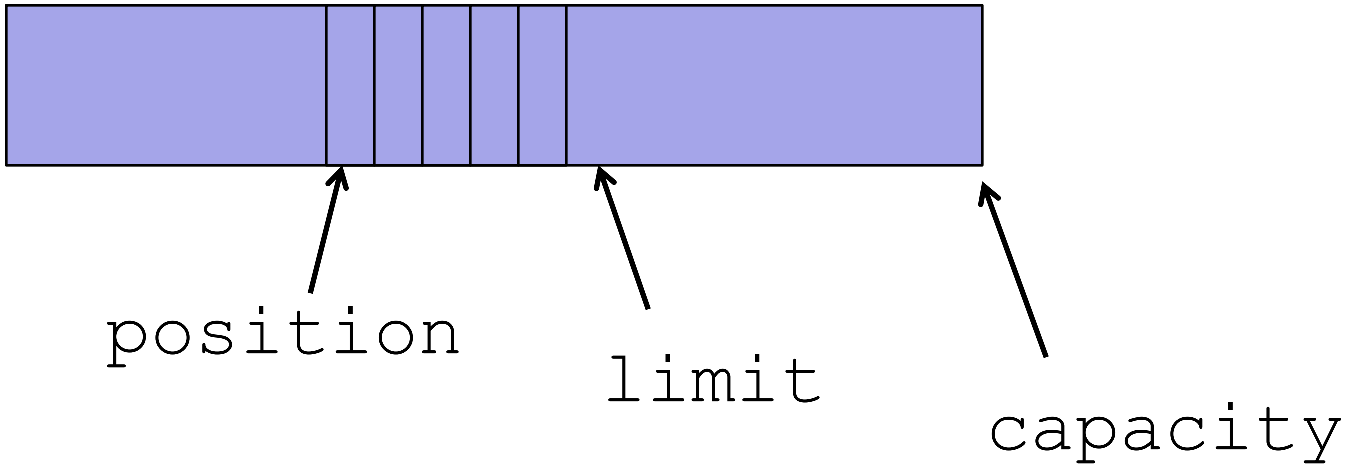
- `Buffer.clear(): position = 0; limit=capacity`

# channel.read(Buffer)



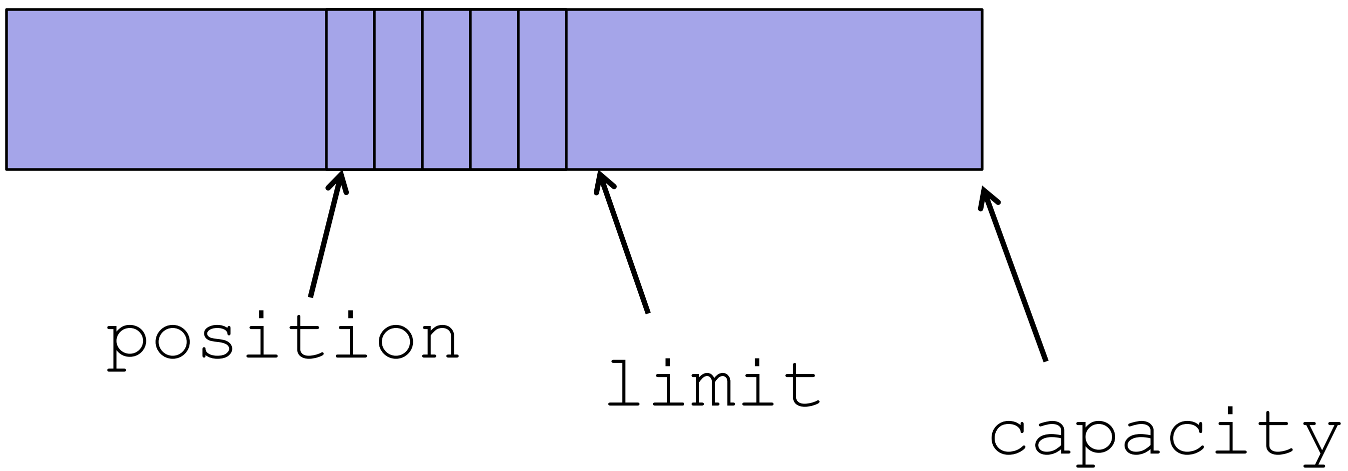

- Put data into Buffer, starting at `position`, not to reach `limit`

# channel.write(Buffer)



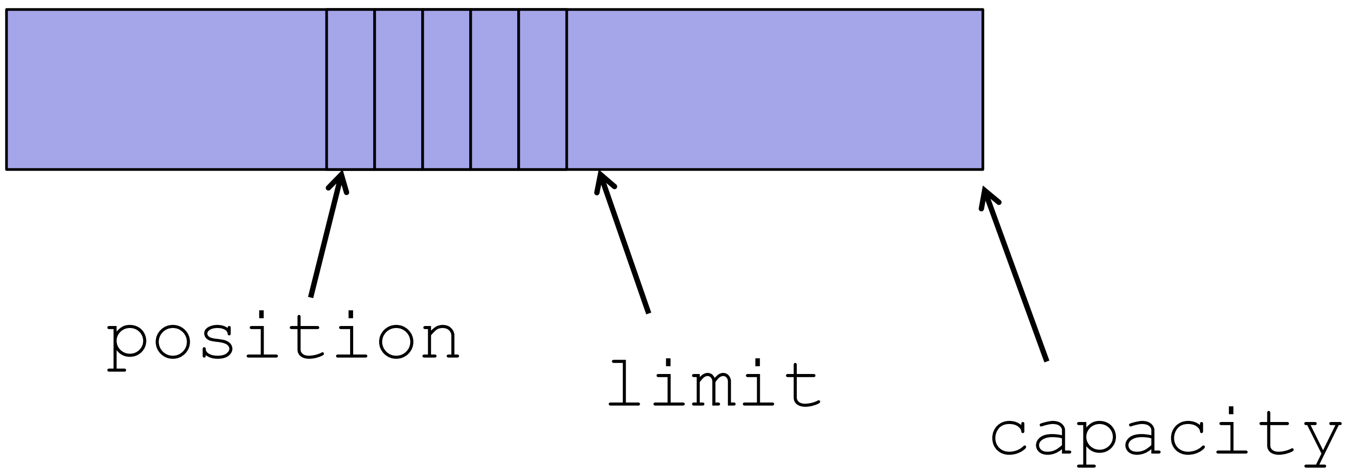

- Move data from Buffer to channel, starting at `position`, not to reach `limit`

# 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

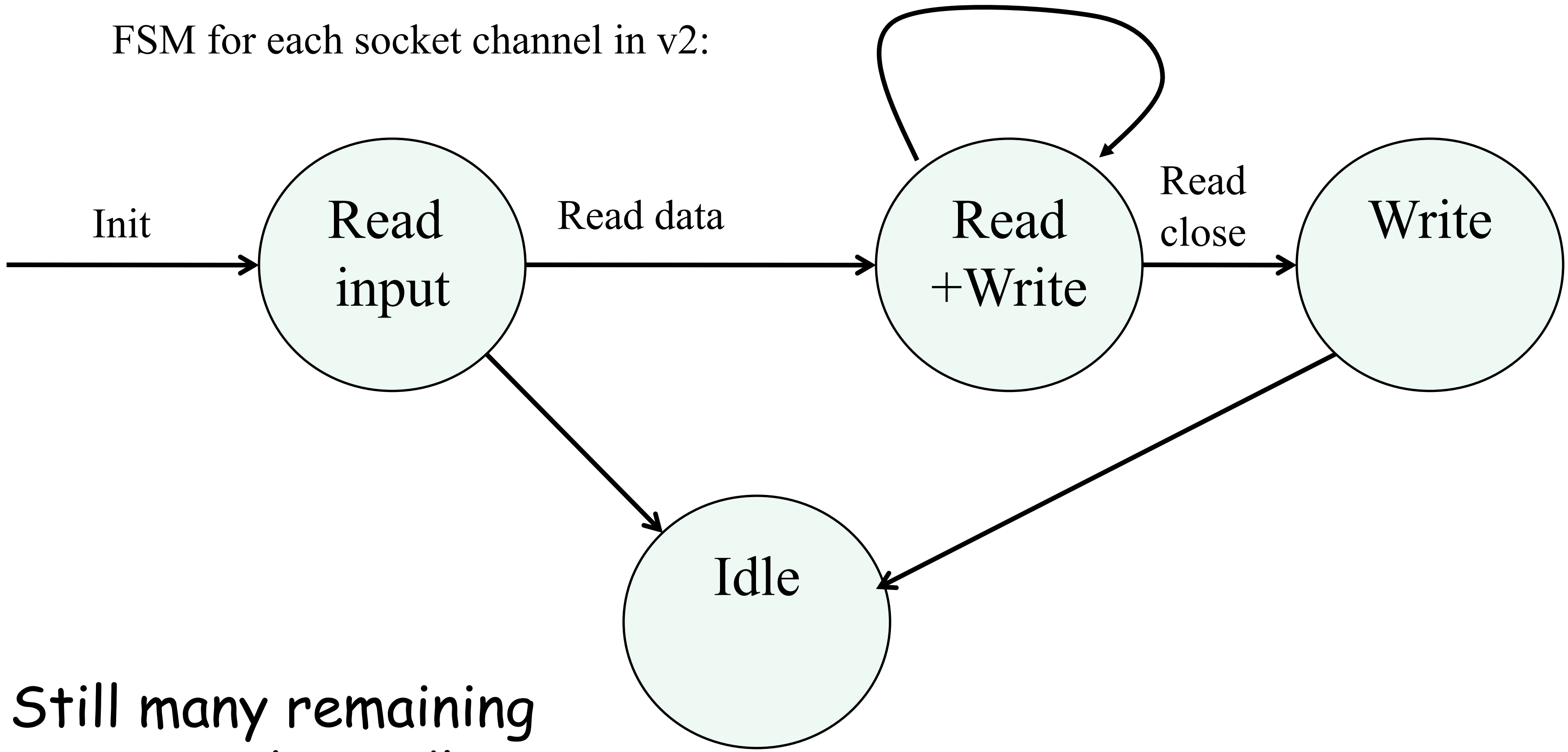
# Problems of Echo Server v1

- 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)

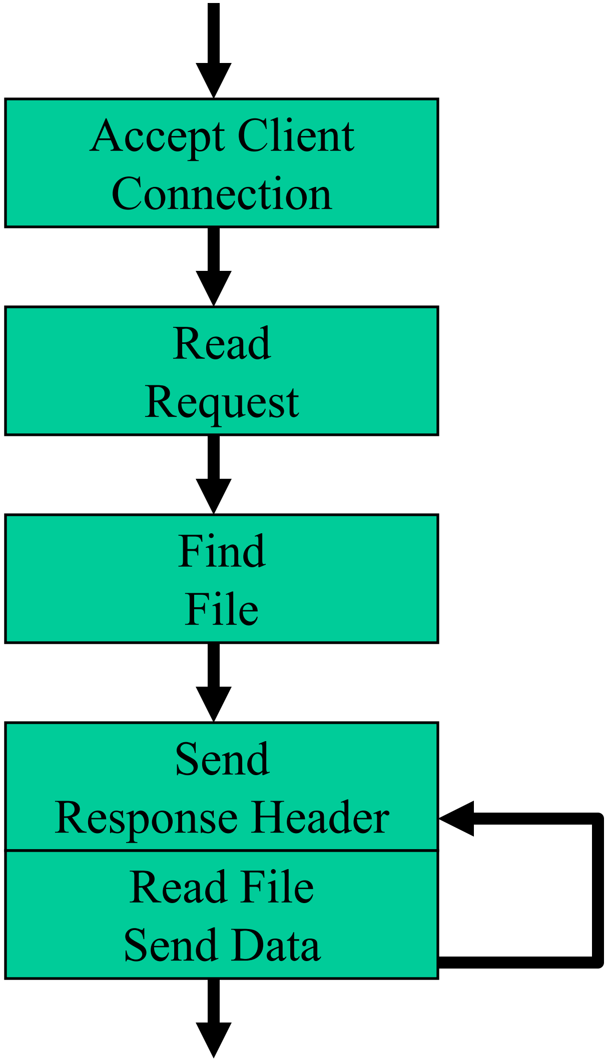
FSM for each socket channel in v2:



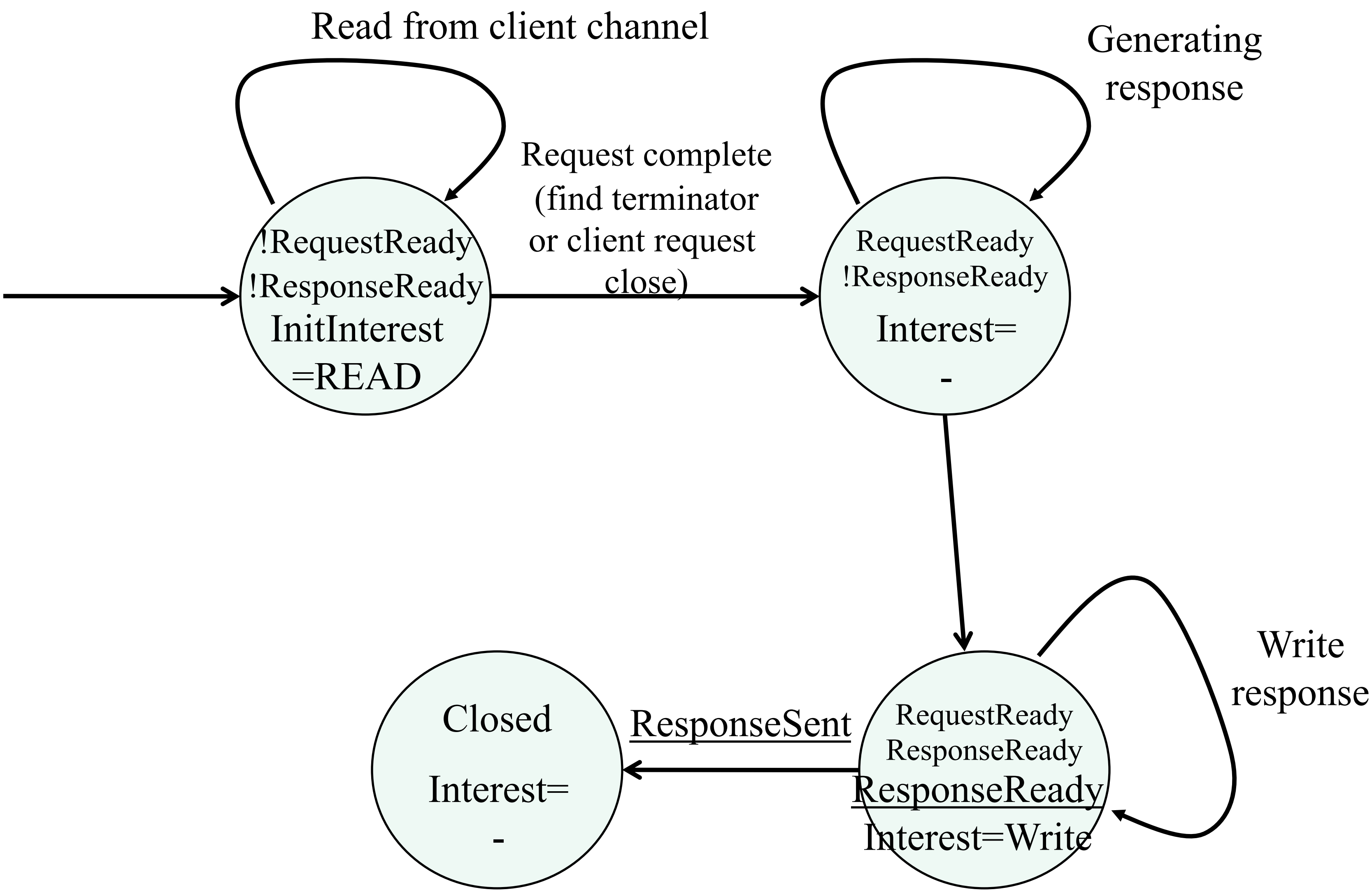

Still many remaining issues such as idle instead of close

# 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);
}
```

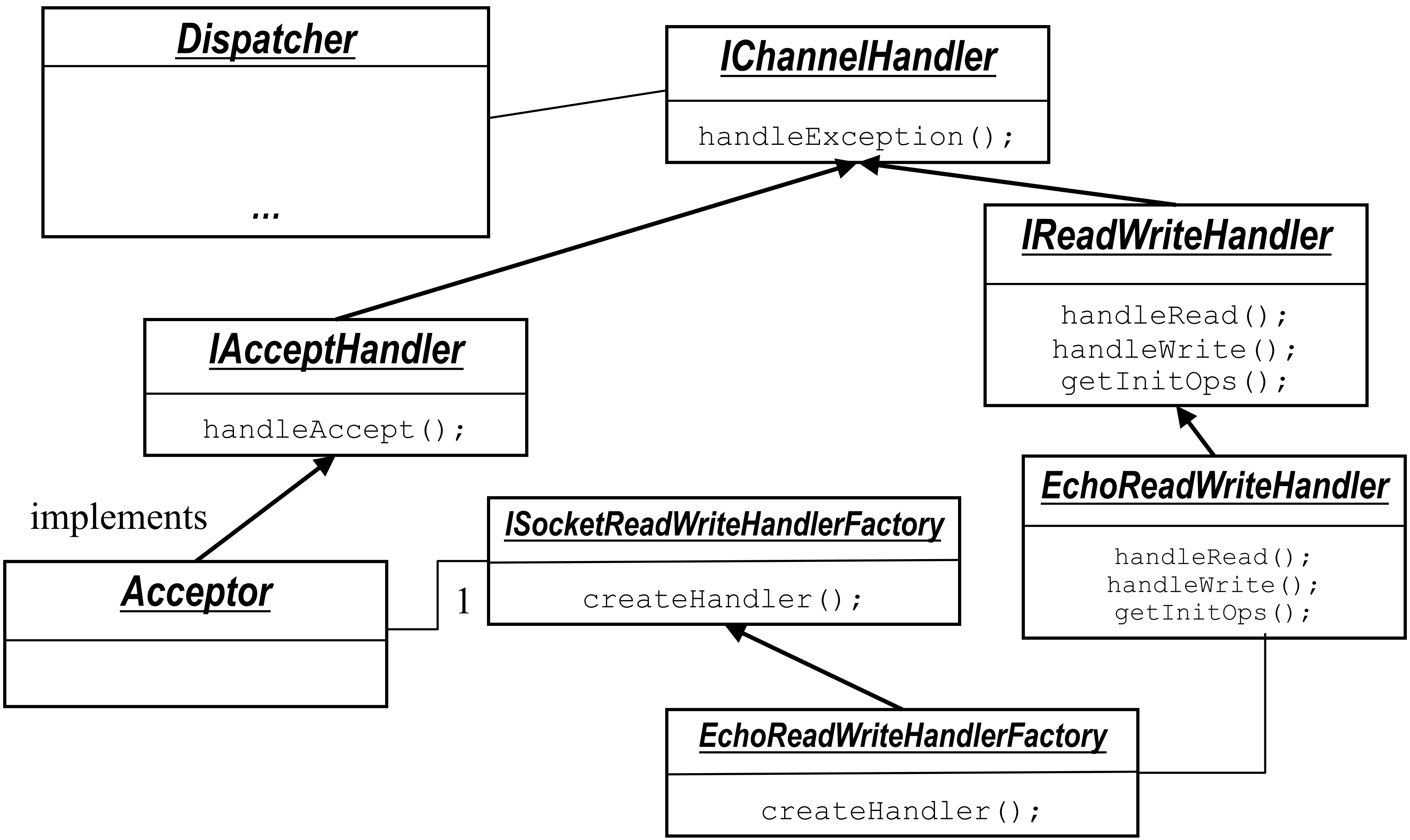
# Handler Design: Acceptor

- 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

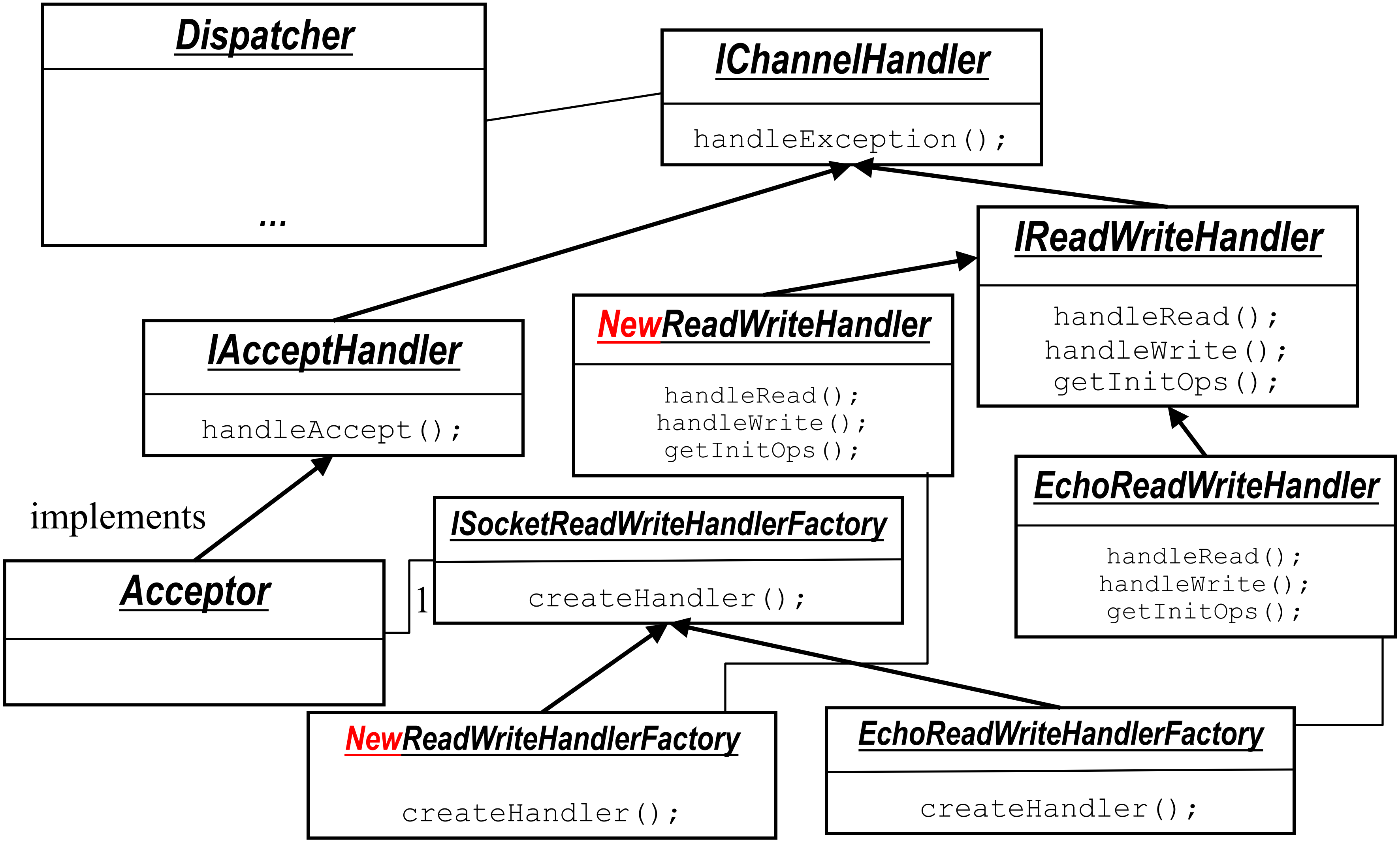
# Handler Design: ReadWriteHandler

- 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

# 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

```
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?

```java
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);
```

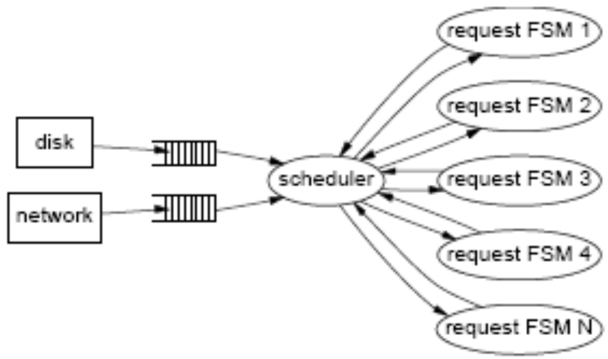
## CompletionHandler

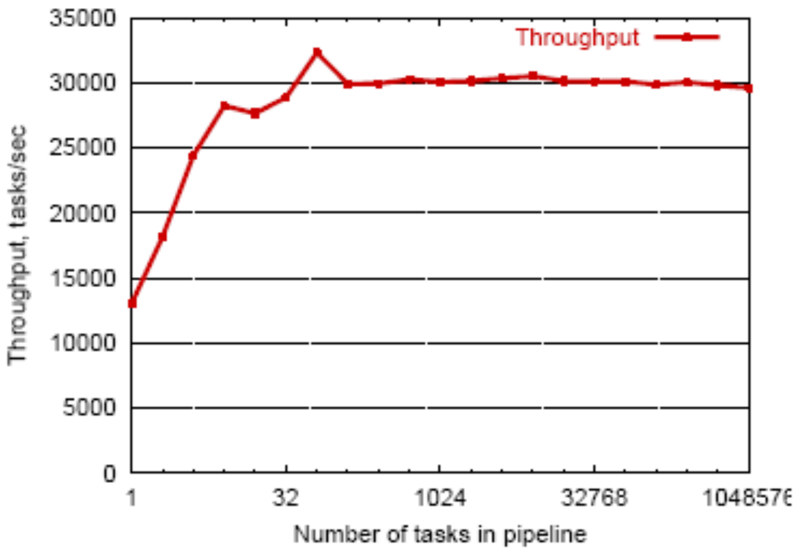
```
class LineHandler implements
CompletionHandler<Integer, ByteBuffer> {

  @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

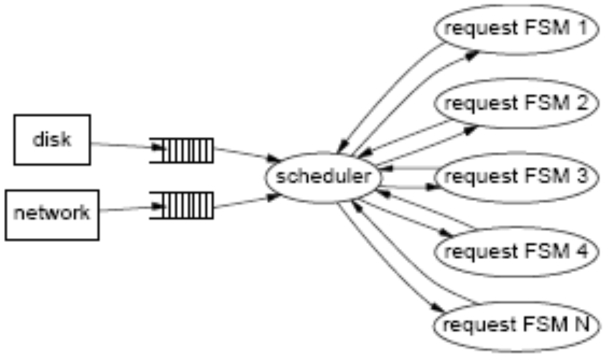



- Advantages
  - Single address space for ease of sharing
  - No synchronization/thread overhead
- Many examples: Click router, Flash web server, TP Monitors, NOX controller, Google Chrome (libevent), Dropbox (libevent), ...

# 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

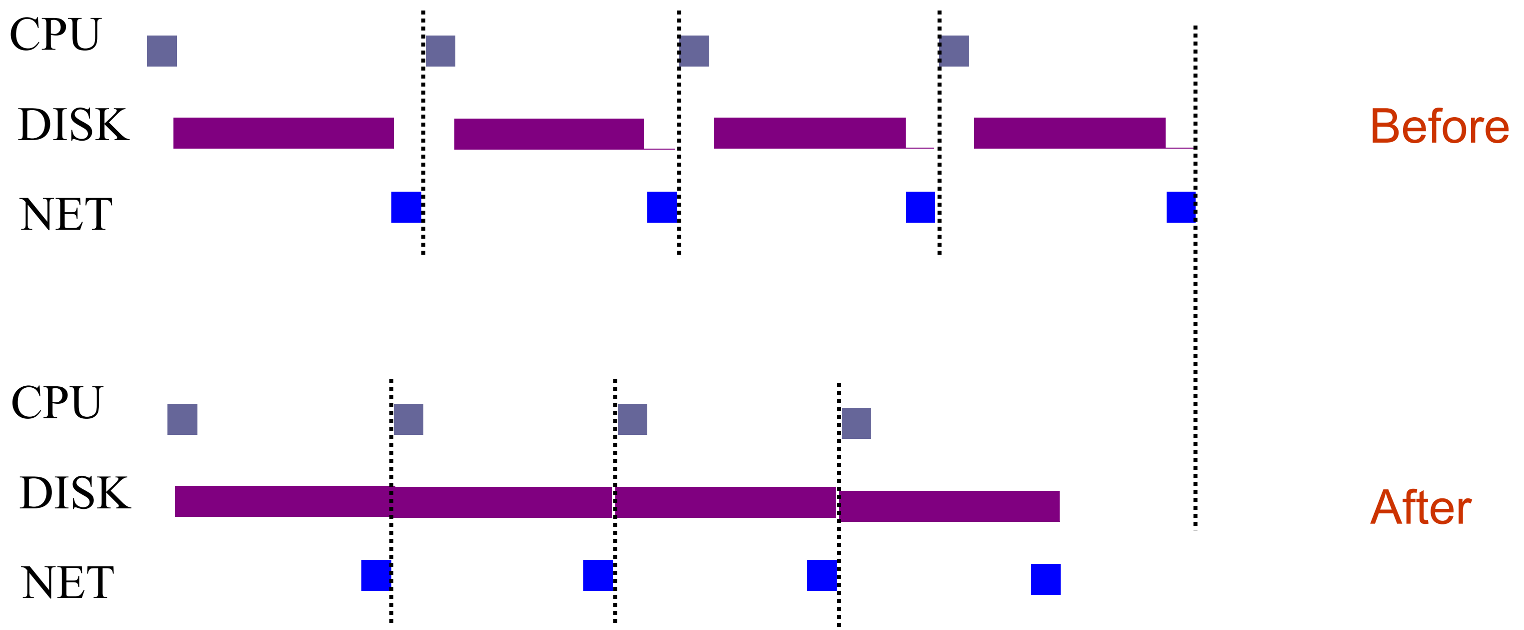
- Avoid blocking (so that we can reach bottleneck throughput)
  - Introduce threads
- Limit unlimited thread overhead
  - Thread pool, async io
- Coordinating data access
  - synchronization (lock, synchronized)
- Coordinating behavior: avoid busy-wait
  - wait/notify; select FSM, Future/Listener
- Extensibility/robustness
  - language support/design for interfaces

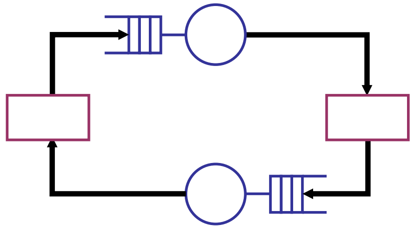
# Beyond Class: Design Patterns

- We have see 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

# Recap: Best Server Design Limited Only by Resource 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

# Outline

- Admin and recap
- High performance servers
  - Thread design
  - Asynchronous design
  - *Operational analysis*

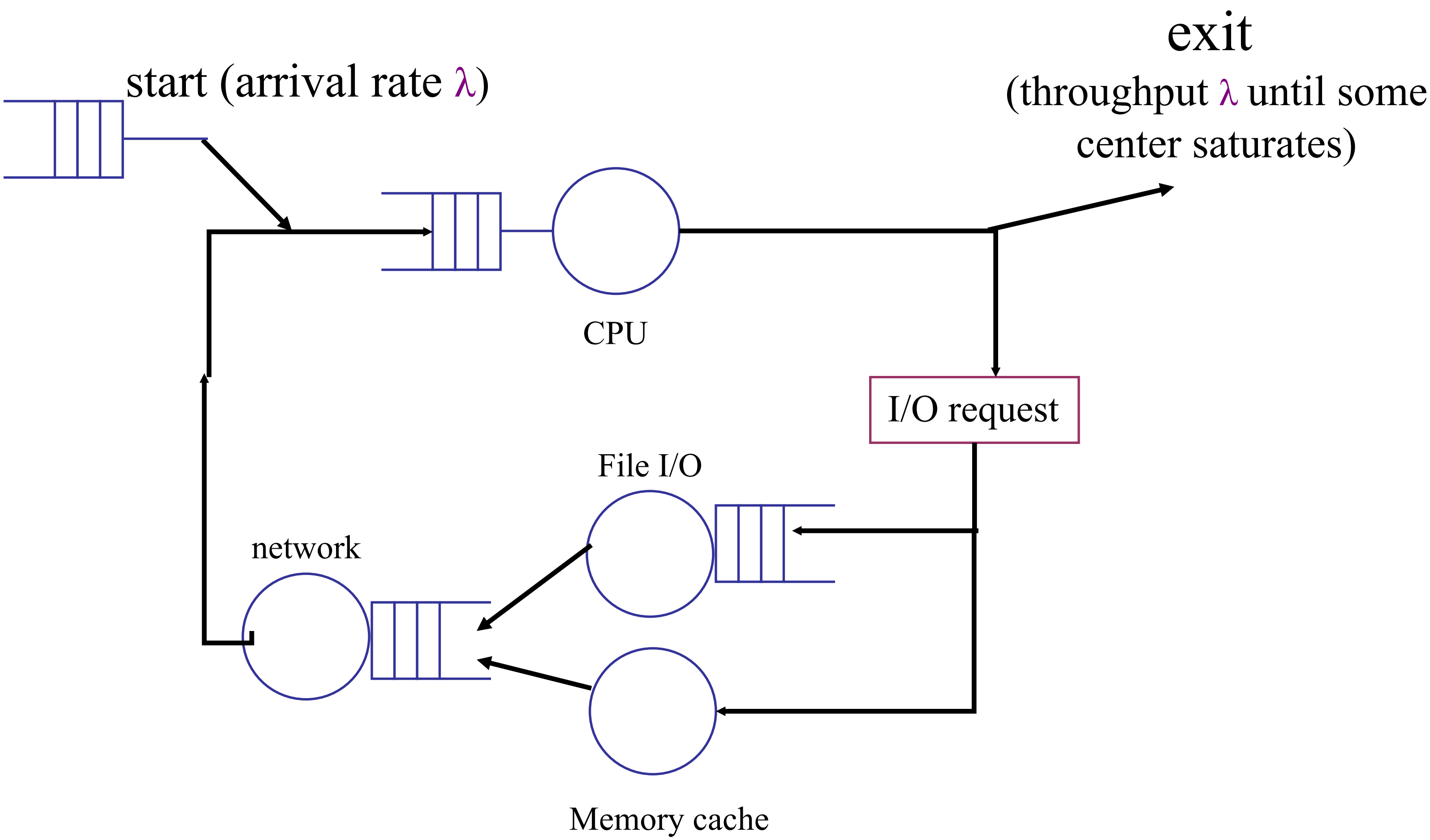
# Operational Analysis

- Relationships that do not require any assumptions about the distribution of service times or inter-arrival times
  - Hence focus on measurements

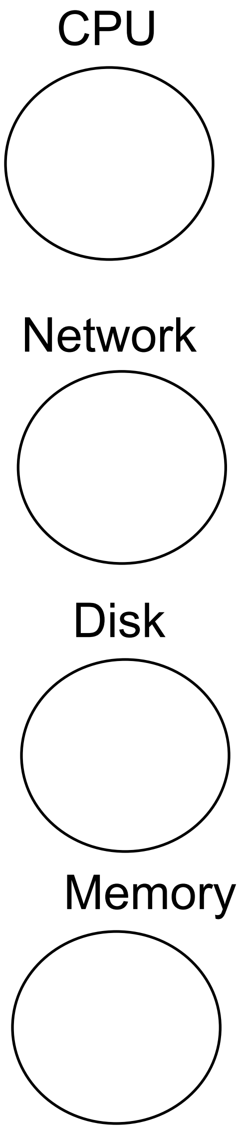
- Identified originally by Buzen (1976) and later extended by Denning and Buzen (1978).

- We touch only some techniques/results
  - In particular, bottleneck analysis
- More details see linked reading

# Under the Hood (An example FSM)



# Operational Analysis: Resource Demand of a Request




$V_{CPU}$ visits for $S_{CPU}$ units of resource time per visit

$V_{Net}$ visits for $S_{Net}$ units of resource time per visit

$V_{Disk}$ visits for $S_{Disk}$ units of resource time per visit

$V_{Mem}$ visits for $S_{Mem}$ units of resource time per visit

# Operational Quantities

- T: observation interval
- Bi: busy time of device i
- i = 0 denotes system

Ai: # arrivals to device i

Ci: # completions at device i

$$\text{arrival rate } \lambda_i = \frac{A_i}{T}$$

$$\text{Throughput } \mathrm{X}_i = \frac{C_i}{T}$$

$$\text{Utilization } \mathrm{U}_i = \frac{B_i}{T}$$

$$\text{Mean service time } \mathrm{S}_i = \frac{B_i}{C_i}$$

# Utilization Law

$$\text{Utilization } U_i = \frac{B_i}{T}$$

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

$$= X_i S_i$$

- 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

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

# Forced Flow Law

- Assume each request visits device i Vi times

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

$$= \frac{C_i}{C_0}\frac{C_0}{T}$$

$$= V_i X$$

# Bottleneck Device

$$\text{Utilization } U_i = X_i S_i$$
$$= V_i X S_i$$
$$= X V_i S_i$$

- Define Di = Vi Si as the total demand of a request on device i
- The device with the highest Di has the highest utilization, and thus is called the bottleneck

# Bottleneck vs System Throughput

$$\text{Utilization } \mathrm{U}_i = X V_i S_i \leq 1$$

$$\rightarrow \; X \leq \frac{1}{D_{\max}}$$

# 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:
  - Ddisk = 0.5 * 1 ms * 1M/8K = 62.5 ms (e.q. = 16 requests/s)