# Design of a Socket Server

Weng Kai

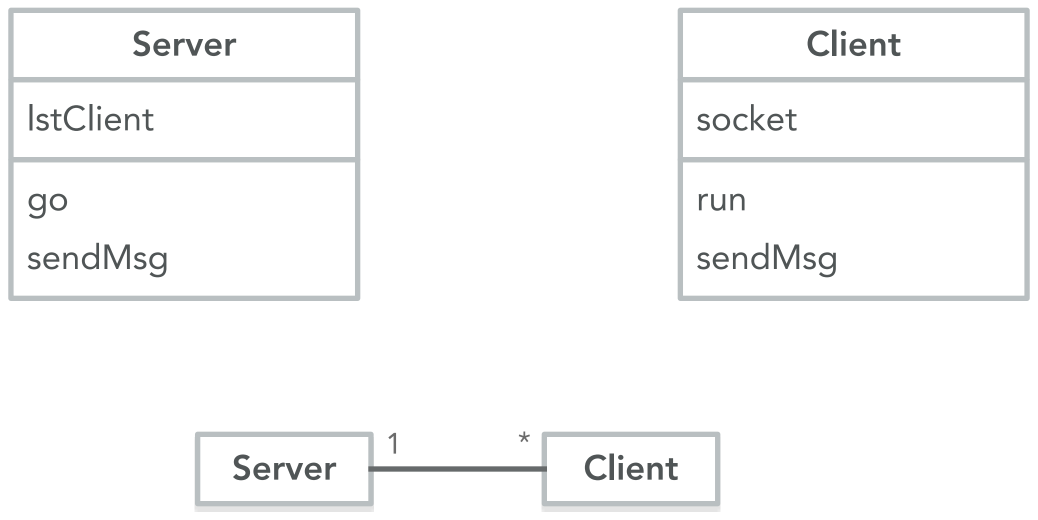
## Requirement

- 一个聊天服务器，多个客户端连接到一个服务器，客户端发送的文本消息被转发给所有的客户端

# Network Services

- Most have same basic structure:
  - Read request
  - Decode request
  - Process service
  - Encode reply
  - Send reply
- But differ in nature and cost of each step
  - XML parsing, File transfer, Web page generation, computational services, ...

# Classes



- Server knows all the clients connected
- During the run(), one client receives a message

## Client knows Server?

- Server is a singleton
- Client has a member variable points the server
- Client provides a listener interface that the server registers at
- Client provides a listener interface that all the other clients register at

# Who issues send?

- One client receives a message and issues a sending
  - The sending runs in the client’s receiving thread
- Should the sendMsg() be synchronized?
- Should there be a thread for sending in each client?

## Message Queue

- Put a message queue at the server
  - which is synchronized
  - Multi-write and single read
- Who runs the queue?
  - A thread at the server runs the queue
  - Need to be notified when new message arrives

# The Queue Runner

- wait() until a new message arrives and been notified
- Iterates the list of client and calls sendMsg() of each client
- What if sendMsg() of one client stalls?
  - Need thread to sendMsg() for each client

# How does client quit?

- Client can send a special message to quit, or
- the server knows that one client is gone
  - read() or write() throws exception
- Who knows one client is leaving?
  - read(): read thread at each client
  - write(): write thread at the server

# How does client quit?

- Server needs to get to know that one client is leaving, how?
- Server needs to remove the client from the list, how?
- The thread in client needs to quit itself, how?
  - read() fails —> quit directly
  - write() fails?

# Binary Message

- It is common that socket server deals with binary message instead of text

```
int cmd = in.read();
```

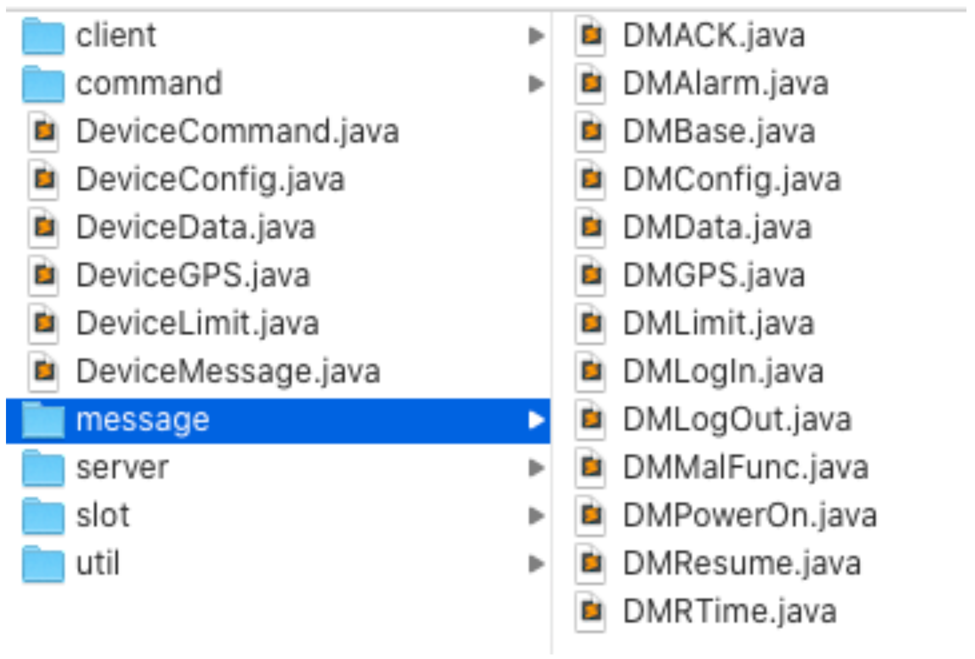
- Then what? switch_case on cmd?

```
enum {
    CMD_ACK         = 0x00,
    CMD_POWERON = 0x01,
    CMD_LOGON    = 0x11,
    CMD_LOGOUT   = 0x12,
    CMD_DATA        = 0x02,
    CMD_CONF        = 0x03,
    CMD_LIMIT       = 0x04,
    CMD_ALARM       = 0x05,
    CMD_RESUME   = 0x06,
    CMD_GPS         = 0x07,
    CMD_BASE        = 0x08,
    CMD_RTIME       = 0x09,
    CMD_MALFUNC = 0x5A,
    CMD_SACK        = 0x80,
    CMD_TIMESLOT= 0x88,
    CMD_LIMITSET= 0x84,
    CMD_PARAM       = 0x83,
    CMD_GUARD       = 0x85,
    CMD_MONI        = 0x86,
    CMD_TRACE       = 0x87,
    CMD_QUERY       = 0x82,
    CMD_STIME       = 0x89,
    CMD_RESTART = 0xA5,
    CMD_UPDATE   = 0xAA
};
```

```
typedef struct
_MsgQueueNode
{
    char cmd;
    char deviceSN[4];
    char msgSN[4];
    char time[7];
    char
content[MAX_SIZE];
    int length;
    char sentMin;
   struct _MsgQueueNode
*next;
} MsgQueueNode;
```

# Message Classes

- Define an abstract class DeviceMessage and classes derived from the DeviceMessage to deal with different messages



```
client                 ▶  DMACK.java
command                ▶  DMAlarm.java
DeviceCommand.java        DMBase.java
DeviceConfig.java         DMConfig.java
DeviceData.java           DMData.java
DeviceGPS.java            DMGPS.java
DeviceLimit.java          DMLimit.java
DeviceMessage.java        DMLogIn.java
message                ▶  DMLogOut.java
server                 ▶  DMMalFunc.java
slot                   ▶  DMPowerOn.java
util                   ▶  DMResume.java
                          DMRTime.java
```

```
public static DeviceMessage parse(DataInputStream in)
```

- A factory-ish methods generates object of the message based on the first byte

- But how?

```java
int cmd = in.read();
Class<? extends DeviceMessage> dpc = DPTypes.get((byte)cmd);

private static HashMap<Byte, Class<? extends DeviceMessage>>  DPTypes = new HashMap<>();

static {
        DPTypes.put((byte)0x00, DMACK.class);
        DPTypes.put((byte)0x01, DMPowerOn.class);
        DPTypes.put((byte)0x02, DMData.class);
        DPTypes.put((byte)0x03, DMConfig.class);
        DPTypes.put((byte)0x04, DMLimit.class);
        DPTypes.put((byte)0x05, DMAlarm.class);
        DPTypes.put((byte)0x06, DMResume.class);
        DPTypes.put((byte)0x07, DMGPS.class);
        DPTypes.put((byte)0x08, DMBase.class);
        DPTypes.put((byte)0x09, DMRTime.class);
        DPTypes.put((byte)0x11, DMLogIn.class);
        DPTypes.put((byte)0x12, DMLogOut.class);
        DPTypes.put((byte)0x5a, DMMalFunc.class);
}
```

# enum

```
public enum Operator {
    make(2) {
        public Value execute(Value[] args) {
            NameSpace.put(args[0].getValue(), args[1]);
            return null;
        }
    },
    print(1) {
        public Value execute(Value[] args) {
            System.out.println(args[0].getValue());
            return null;
        }
    },
    Operator(int num) { numOperand = num; }

    public Value execute(Value[] args) { return null; };

    public int numOperand() { return numOperand; }

    private int numOperand;
}
```

## nio

# Client Threads?

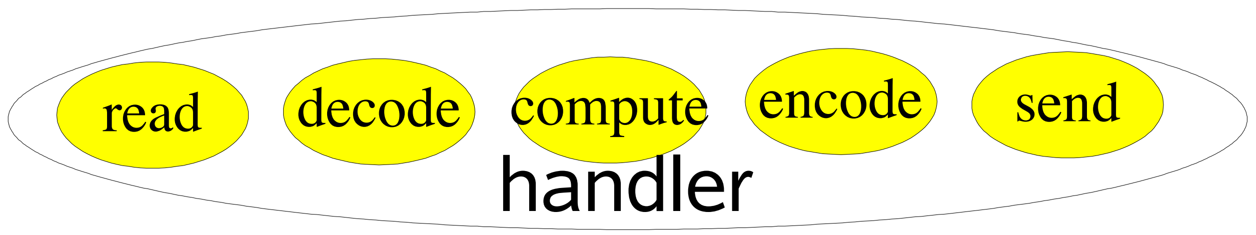
- One thread for one client and one thread for the ServerSocket and one thread for the message queue and one thread for each client to sendMsg
  - Will there be too much thread?
  - Most of the threads are waiting for new messages coming
- nio utilizes one thread that listens to many sockets

# Scalability Goals

- Graceful degradation under increasing load (more clients)
- Continuous improvement with increasing resources (CPU, memory, disk, bandwidth)
- Also meet availability and performance goals
  - Short latencies
  - Meeting peak demand
  - Tunable quality of service
- Divide-and-conquer is usually the best approach for achieving any scalability goal

# Divide and Conquer

- Divide processing into small tasks
    - Each task performs an action without blocking
- Execute each task when it is enabled
    - Here, an IO event usually serves as trigger

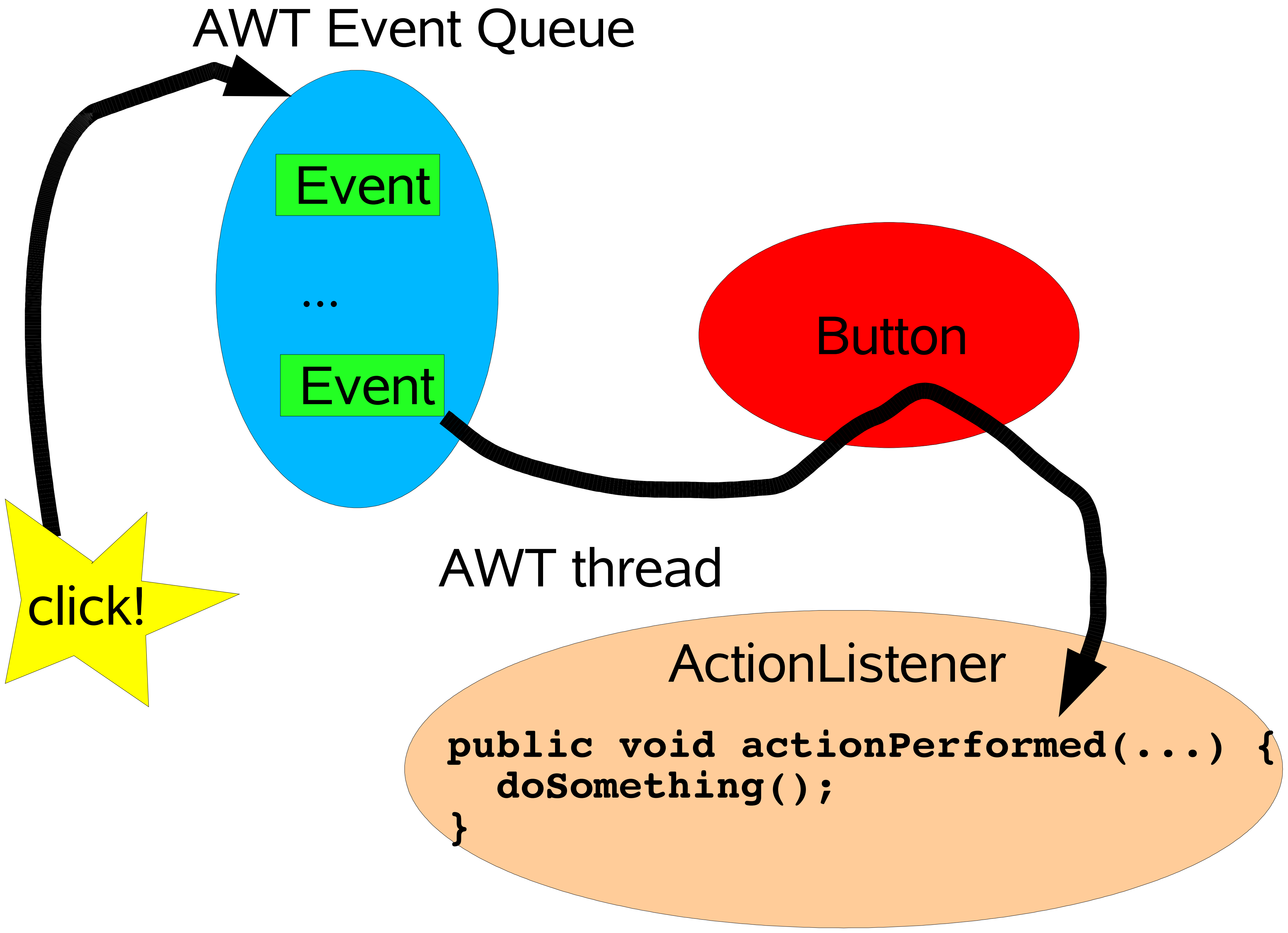


- Basic mechanisms supported in java.nio
    - Non-blocking reads and writes
    - Dispatch tasks associated with sensed IO events
- Endless variation possible
    - A family of event-driven designs

# Event-driven Designs

- Usually more efficient than alternatives
  - Fewer resources: Don't usually need a thread per client
  - Less overhead: Less context switching, often less locking
  - But dispatching can be slower: Must manually bind actions to events
- Usually harder to program
  - Must break up into simple non-blocking actions
    - Similar to GUI event-driven actions
    - Cannot eliminate all blocking: GC, page faults, etc
  - Must keep track of logical state of service

# Background: Events in AWT

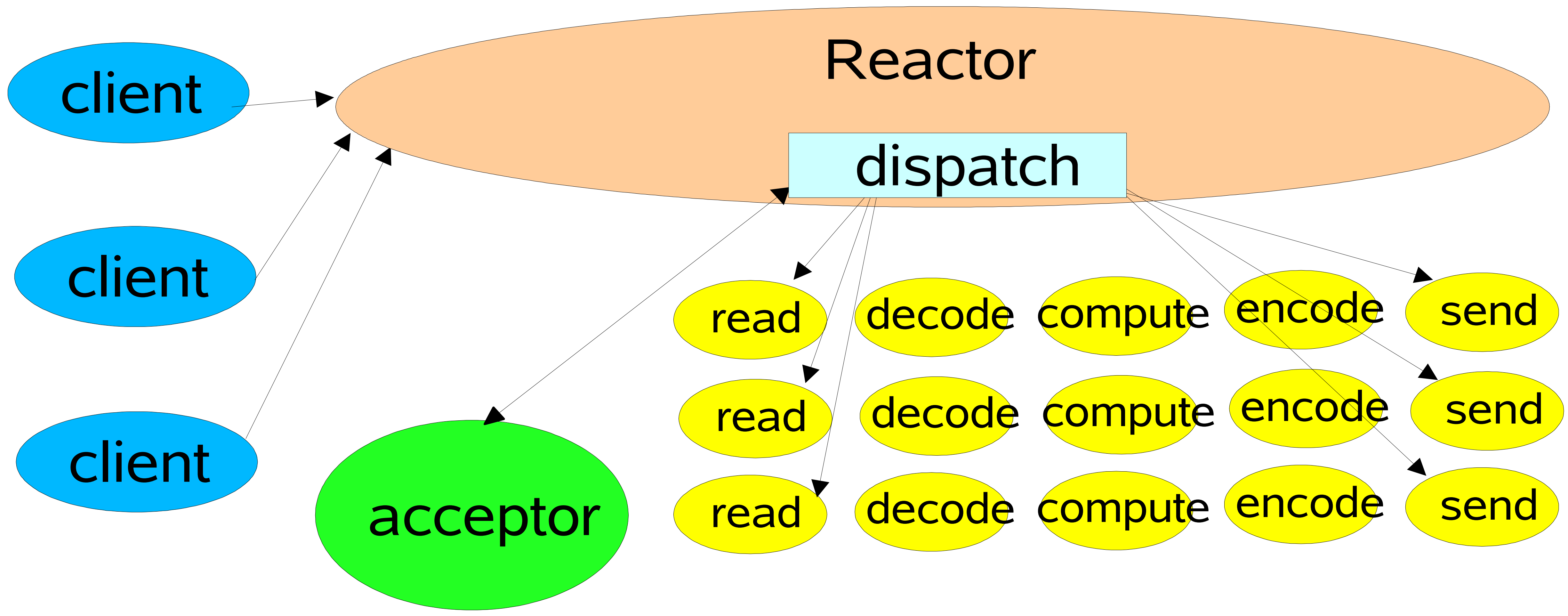


```
public void actionPerformed(...) {
  doSomething();
}
```

# Reactor Pattern

- Reactor responds to IO events by dispatching the appropriate handler
    - Similar to AWT thread
- Handlers perform non-blocking actions
    - Similar to AWT ActionListeners
- Manage by binding handlers to events
    - Similar to AWT addActionListener
- See Schmidt et al, Pattern-Oriented Software Architecture, Volume 2 (POSA2)
    - Also Richard Stevens's networking books, Matt Welsh's SEDA framework, etc

# Basic Reactor Design



- Single threaded version

## java.nio Support

- Channels: Connections to files, sockets etc that support non-blocking reads
- Buffers: Array-like objects that can be directly read or written by Channels
- Selectors: Tell which of a set of Channels have IO events
- SelectionKeys: Maintain IO event status and bindings

# nio

- IO是面向流的处理，NIO是面向块(缓冲区)的处理
  - 面向流的I/O 系统一次一个字节地处理数据
  - 一个面向块(缓冲区)的I/O系统以块的形式处理数据
- NIO主要有三个核心部分组成：
  - buffer缓冲区
  - Channel管道
  - Selector选择器

# buffer缓冲区和Channel管道

- NIO以buffer缓冲区和Channel管道配合使用来处理数据
    - Channel管道比作成铁路，buffer缓冲区比作成火车(运载着货物)
- 常用的Buffer：ByteBuffer

# ByteBuffer

- A byte buffer is either direct or non-direct. Given a direct byte buffer, the Java virtual machine will make a best effort to perform native I/O operations directly upon it. That is, it will attempt to avoid copying the buffer's content to (or from) an intermediate buffer before (or after) each invocation of one of the underlying operating system's native I/O operations.
- A direct byte buffer may be created by invoking the allocateDirect factory method of this class.

**https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html**

## ByteBuffer

- Access to binary data

```
float  getFloat()
float  getFloat(int index)
void   putFloat(float f)
void   putFloat(int index, float f)
```

- Invocation chaining

```
bb.putInt(0xCAFEBABE).putShort(3).putShort(45);
```

```
ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
String s = "Java3y";
byteBuffer.put(s.getBytes());
```

# Reactor 1: Setup

```
class Reactor implements Runnable {
  final Selector selector;
  final ServerSocketChannel serverSocket;

  Reactor(int port) throws IOException {
    selector = Selector.open();
    serverSocket = ServerSocketChannel.open();
    serverSocket.socket().bind(
                    new InetSocketAddress(port));
    serverSocket.configureBlocking(false);
    SelectionKey sk =
      serverSocket.register(selector,

SelectionKey.OP_ACCEPT);
    sk.attach(new Acceptor());
  }

  /*
   Alternatively, use explicit SPI provider:
    SelectorProvider p = SelectorProvider.provider();
    selector = p.openSelector();
    serverSocket = p.openServerSocketChannel();
  */
```

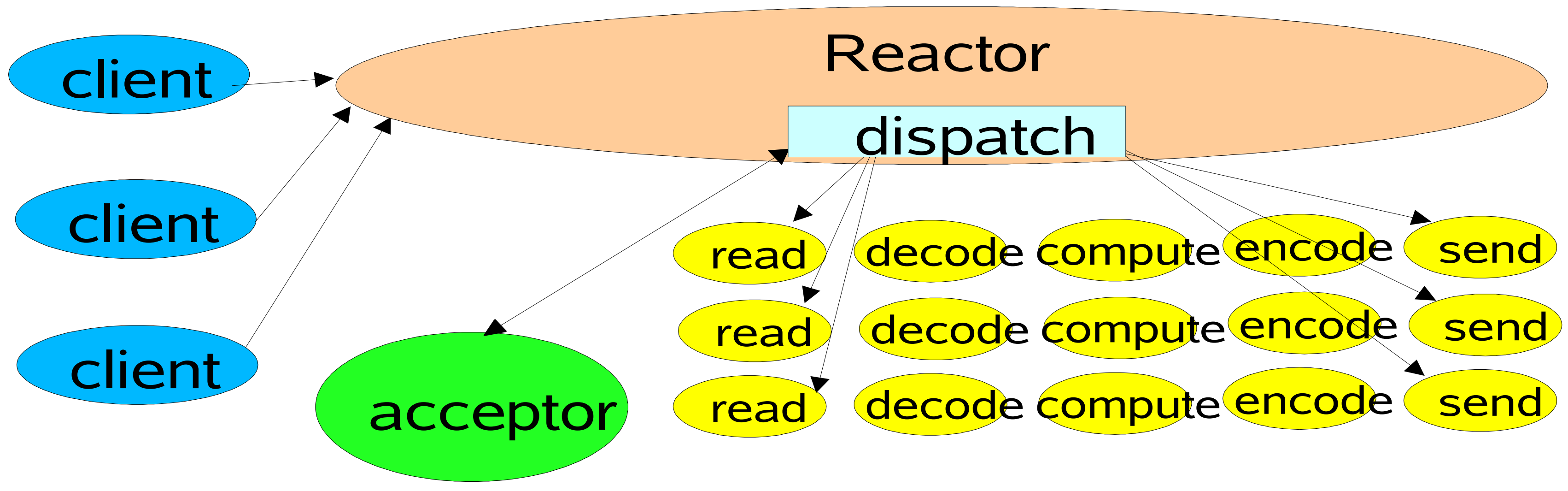
# Reactor 2: Dispatch Loop

```
// class Reactor continued
  public void run() {  // normally in a new
Thread
    try {
      while (!Thread.interrupted()) {
        selector.select();
        Set selected = selector.selectedKeys();
        Iterator it = selected.iterator();
        while (it.hasNext())
          dispatch((SelectionKey)(it.next());
        selected.clear();
      }
    } catch (IOException ex) { /* ... */ }
  }

  void dispatch(SelectionKey k) {
    Runnable r = (Runnable)(k.attachment());
    if (r != null)
      r.run();
  }
```

# Reactor 3: Acceptor

```
// class Reactor continued
   class Acceptor implements Runnable { // inner
      public void run() {
         try {
            SocketChannel c = serverSocket.accept();
            if (c != null)
               new Handler(selector, c);
         }
         catch(IOException ex) { /* ... */ }
      }
   }
}
```



# Reactor 4: Handler setup

```
final class Handler implements Runnable {
  final SocketChannel socket;
  final SelectionKey sk;
  ByteBuffer input = ByteBuffer.allocate(MAXIN);
  ByteBuffer output = ByteBuffer.allocate(MAXOUT);
  static final int READING = 0, SENDING = 1;
  int state = READING;

  Handler(Selector sel, SocketChannel c)
   throws IOException {
    socket = c; c.configureBlocking(false);
    // Optionally try first read now
    sk = socket.register(sel, 0);
    sk.attach(this);
    sk.interestOps(SelectionKey.OP_READ);
    sel.wakeup();
  }

  boolean inputIsComplete()  { /* ... */ }
  boolean outputIsComplete() { /* ... */ }
  void process()             { /* ... */ }
```

# Reactor 5: Request handling

```
// class Handler continued
   public void run() {
    try {
      if      (state == READING) read();
      else if (state == SENDING) send();
    } catch (IOException ex) { /* ... */ }
  }
  void read() throws IOException {
    socket.read(input);
    if (inputIsComplete()) {
       process();
       state = SENDING;
       // Normally also do first write now
       sk.interestOps(SelectionKey.OP_WRITE);
    }
  }
  void send() throws IOException {
    socket.write(output);
    if (outputIsComplete()) sk.cancel();
  }
}
```

# Per-State Handlers

```
class Handler  { // ...

  public void run() { // initial state is reader
    socket.read(input);
    if (inputIsComplete()) {
      process();
      sk.attach(new Sender());
      sk.interest(SelectionKey.OP_WRITE);
      sk.selector().wakeup();
    }
  }

  class Sender implements Runnable {
    public void run(){ // ...
      socket.write(output);
      if (outputIsComplete()) sk.cancel();
    }
  }
}
```

- A simple use of GoF State-Object pattern
  - Rebind appropriate handler as attachment

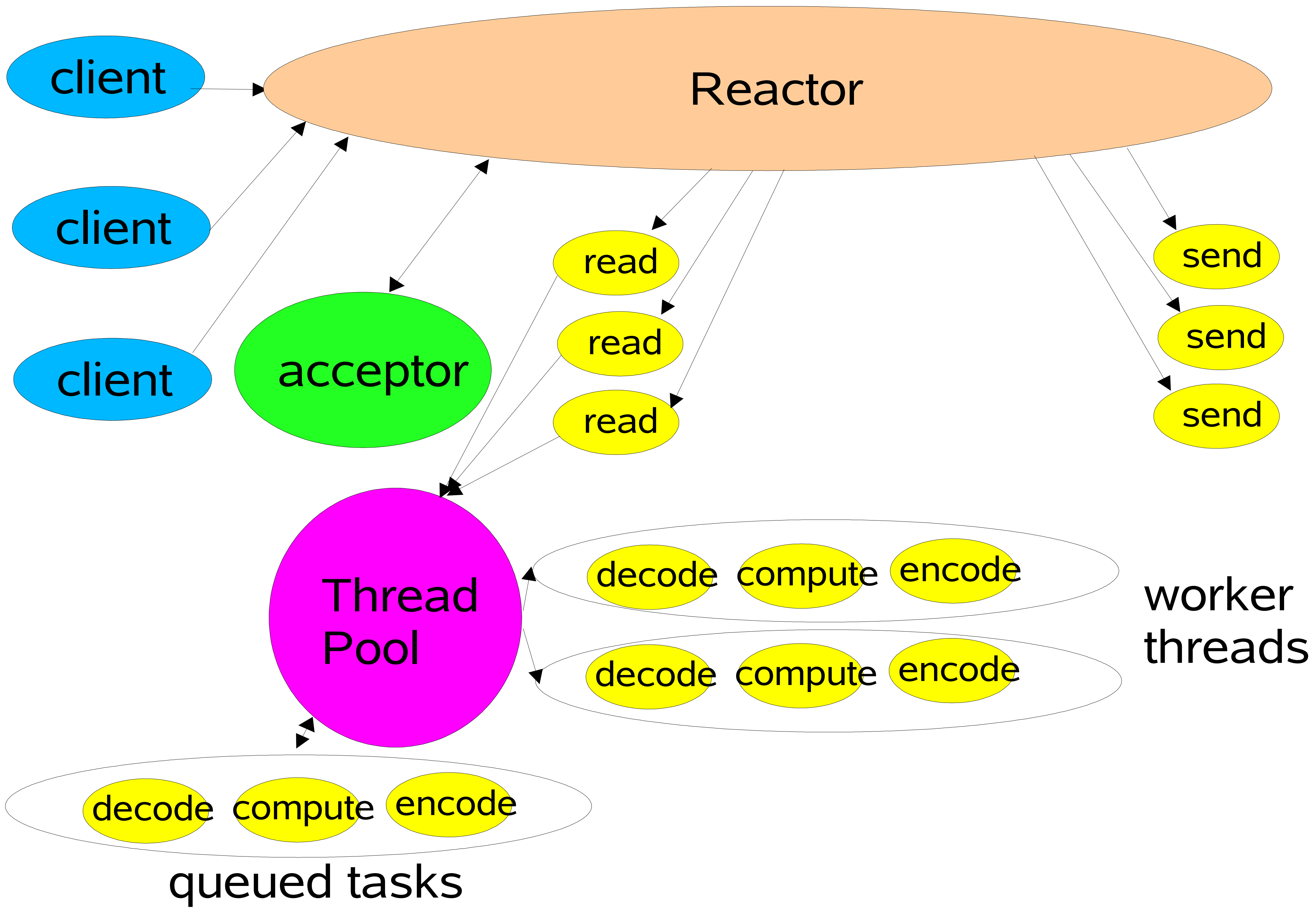
# Multithreaded Designs

- Strategically add threads for scalability: Mainly applicable to multiprocessors
- Worker Threads
  - Reactors should quickly trigger handlers
    - Handler processing slows down Reactor
  - Offload non-IO processing to other threads
- Multiple Reactor Threads
  - Reactor threads can saturate doing IO
  - Distribute load to other reactors
    - Load-balance to match CPU and IO rates

# Worker Threads

- Offload non-IO processing to speed up Reactor thread
  - Similar to POSA2 Proactor designs
- Simpler than reworking compute-bound processing into event-driven form
  - Should still be pure nonblocking computation
  - Enough processing to outweigh overhead
- But harder to overlap processing with IO
  - Best when can first read all input into a buffer
- Use thread pool so can tune and control
  - Normally need many fewer threads than clients

# Worker Thread Pools



# Handler with Thread Pool

```
class Handler implements Runnable {
  // uses util.concurrent thread pool
  static PooledExecutor pool = new PooledExecutor(...);
  static final int PROCESSING = 3;
  // ...

  synchronized void read() { // ...
    socket.read(input);
    if (inputIsComplete()) {
      state = PROCESSING;
      pool.execute(new Processer());
    }
  }

  synchronized void processAndHandOff() {
    process();
    state = SENDING; // or rebind attachment
    sk.interest(SelectionKey.OP_WRITE);
  }

  class Processer implements Runnable {
    public void run() { processAndHandOff(); }
  }
}
```

# Coordinating Tasks

- Handoffs
  - Each task enables, triggers, or calls next one
  - Usually fastest but can be brittle
- Callbacks to per-handler dispatcher
  - Sets state, attachment, etc
  - A variant of GoF Mediator pattern
- Queues: For example, passing buffers across stages
- Futures
  - When each task produces a result
  - Coordination layered on top of join or wait/notify

## Using PooledExecutor

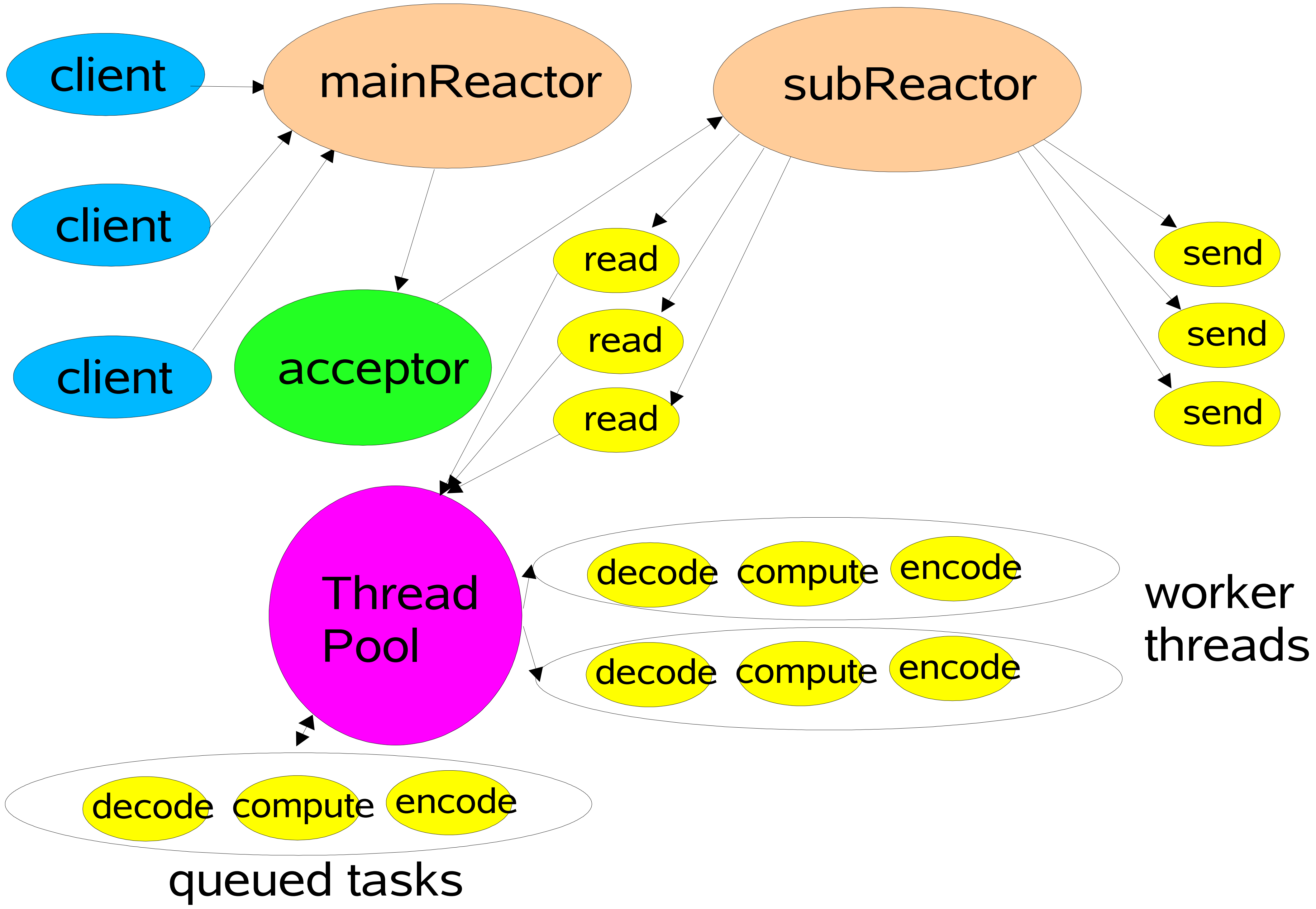
- A tunable worker thread pool
- Main method execute(Runnable r)
- Controls for:
  - The kind of task queue (any Channel)
  - Maximum number of threads
  - Minimum number of threads
  - "Warm" versus on-demand threads
  - Keep-alive interval until idle threads die
    - to be later replaced by new ones if necessary
  - Saturation policy
    - block, drop, producer-runs, etc

# Multiple Reactor Threads

- Using Reactor Pools
  - Use to match CPU and IO rates
  - Static or dynamic construction
    - Each with own Selector, Thread, dispatch loop
  - Main acceptor distributes to other reactors

```
Selector[] selectors; // also create threads
int next = 0;
class Acceptor { // ...
  public synchronized void run() { ...
    Socket connection = serverSocket.accept();
    if (connection != null)
      new Handler(selectors[next], connection);
    if (++next == selectors.length) next = 0;
  }
}
```

# Using Multiple Reactors


client
client
client
mainReactor
acceptor
subReactor
read
read
read
send
send
send
Thread Pool
decode
compute
encode
decode
compute
encode
worker threads
decode
compute
encode
queued tasks