Sockets and JDBC

Weng Kai

Identifying a machine

- 两种形式的地址：
  - The familiar DNS (Domain Name Service) form like www.zju.edu.cn.
  - The "dotted quad" form, which is four numbers separated by dots, such as 123.255.28.120.
- Case Study:WhoAmI.java

TCP vs. UDP

* TCP是有连接的协议
  - 在数据传送前要先建立端到端的连接，在数据传送过程中会校验数据
* UDP是无连接的
  - 数据传送前不需要建立连接，不管对方是否存在可以盲发

Port

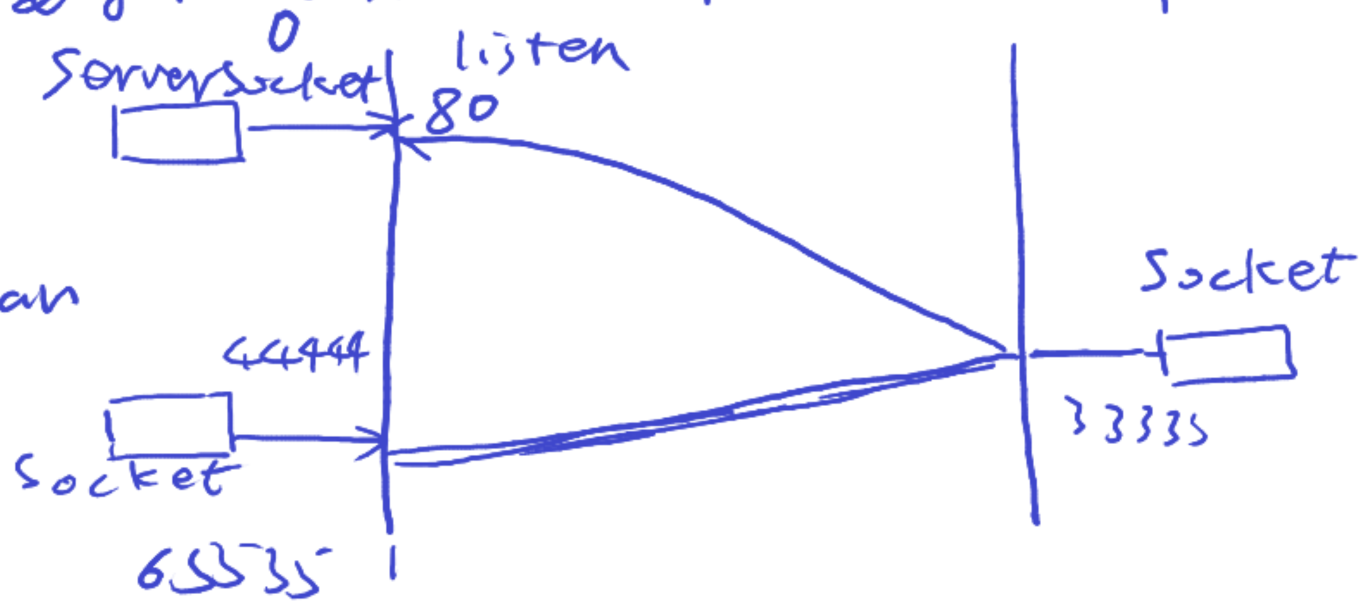
socket port

套接字. 64K 守听

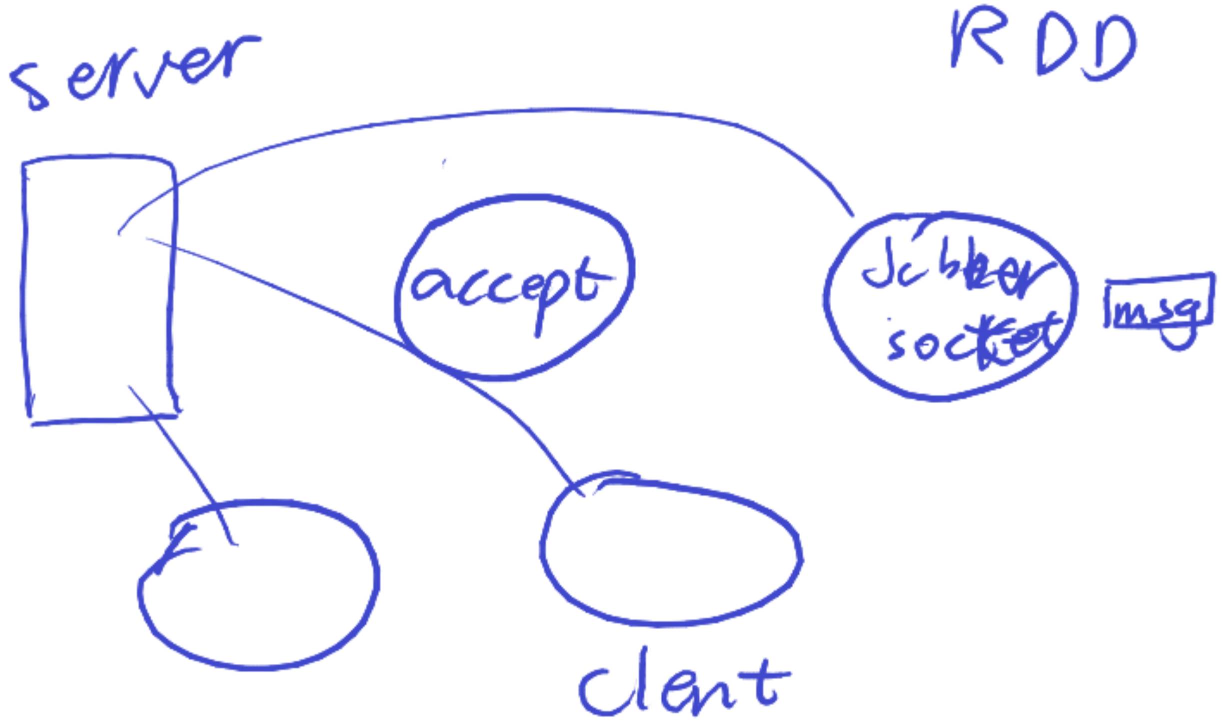
NAT

browser
http://

PDP-11
16-bit
big-endian



RDD



TCP

- 是一种Client/Server模式
- 端口
  - 每个IP设备最多可以有65536个端口，通信是在端口与端口之间进行的
- Server方在固定的端口守听，client方连接该端口，从而形成数据链路

TCP Sockets

- ServerSocket
  - 守听在固定端口，等待client连接的对象
- Socket
  1. Client方用来连接Server方的对象
  2. Server方用来和Client连接的对象

数据传输

- 在Socket对象里有InputStream和OutputStream，用来传输数据
  - nc -l 12345 —> listen
  - nc 12345 —> connect
- Case Study:
- 单服务
  - JabberServer.java
  - JabberClient.java
- 多服务
  - MutiJabberServer.java
  - MutiJabberClient.java

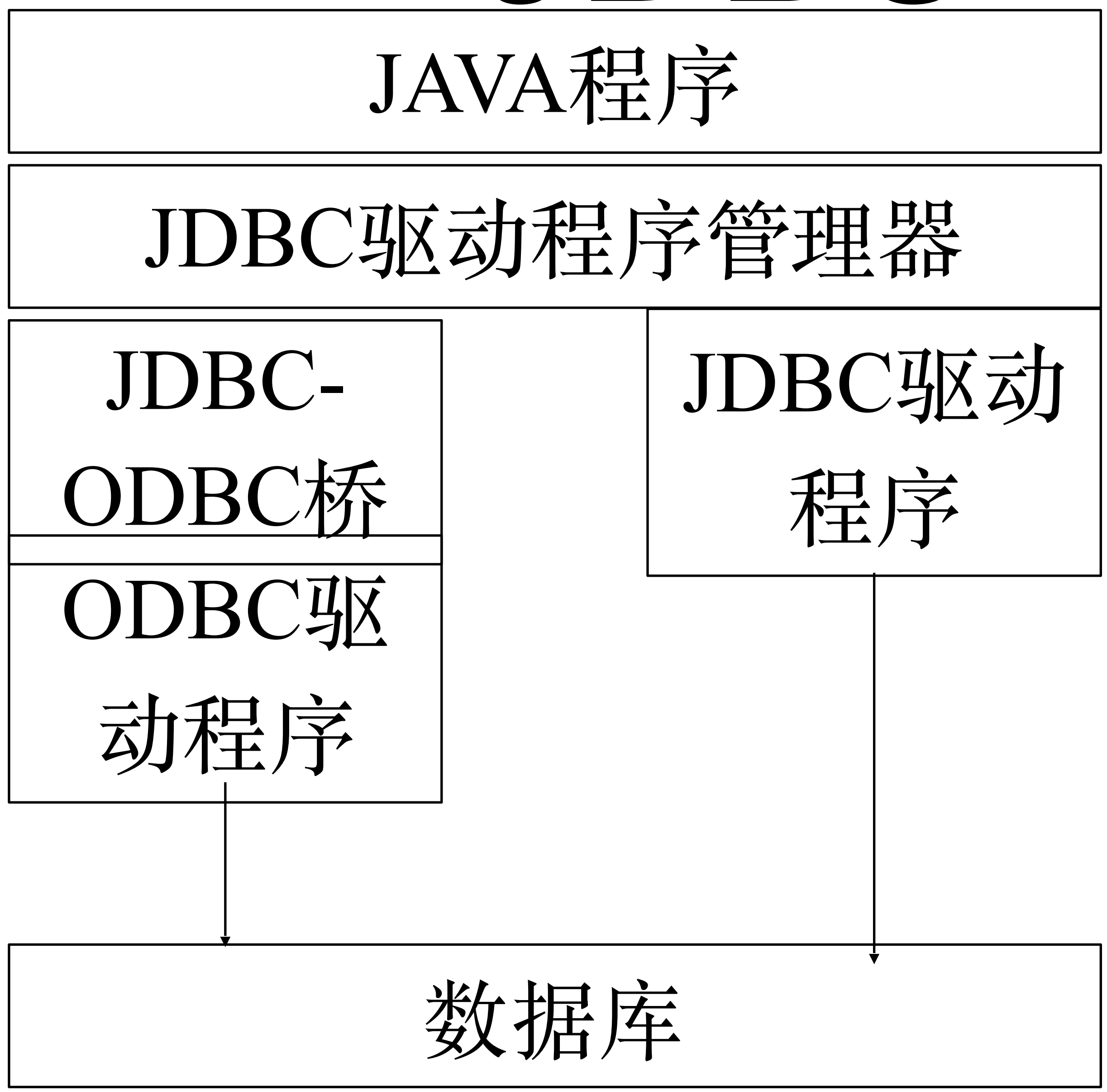
UDP

* DatagramSocket
  * UDP端口，收或发
* DatagramPacket
  * UDP数据
* Case:
  * DayBcast.java
  * DayWatch.java

RTTI in Socket Server

- Binary protocol usually use the first byte as type of messages
- Sub-classes should be used to deal with different types of messages
- Build a Hash-Map of sub-classes Class object to generate dealer of messages

JDBC



* Java DataBase Connection
* 遵循ODBC模型，但不是ODBC

JDBC三步曲

- DriverManager
- Connection
- Statement
- ResultSet
- Case Study: Lookup.java

更新数据库

* Statement.executeUpdate(String SQL);

PreparedStatement

* 由Connection产生PreparedStatement对象，其中的参数用?表示
* 用PreparedStatement的setXXX()函数对参数赋值
* 用PreparedStatement的execute()执行
* Case: Query DB.java

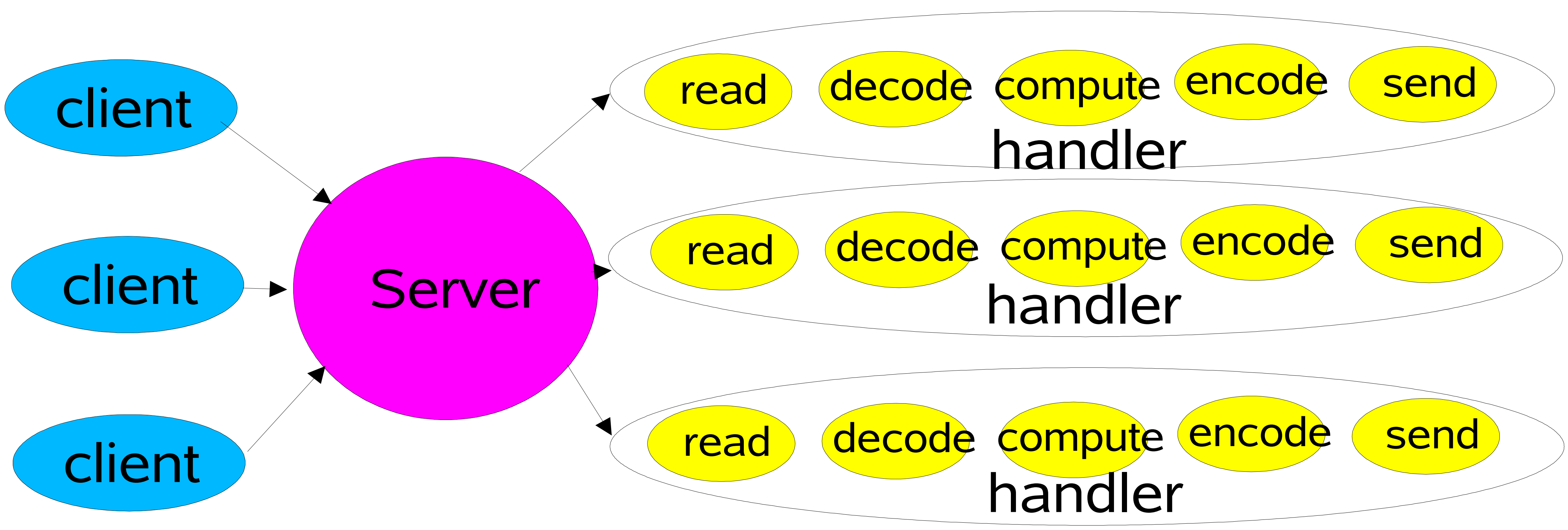
事务处理

* 在Connection上做事务处理
* setAutoCommit()
* commit()
* rollback()

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, ...

Classic Service Designs



Classic ServerSocket Loop

```
class Server implements Runnable {
  public void run() {
    try {
      ServerSocket ss = new ServerSocket(PORT);
      while (!Thread.interrupted())
        new Thread(new Handler(ss.accept())).start();
      // or, single-threaded, or a thread pool
    } catch (IOException ex) { /* ... */ }
  }

  static class Handler implements Runnable {
    final Socket socket;
    Handler(Socket s) { socket = s; }
    public void run() {
      try {
        byte[] input = new byte[MAX_INPUT];
        socket.getInputStream().read(input);
        byte[] output = process(input);
        socket.getOutputStream().write(output);
      } catch (IOException ex) { /* ... */ }
    }
    private byte[] process(byte[] cmd) { /* ... */ }
  }
}
```

多机聊天时的发送问题

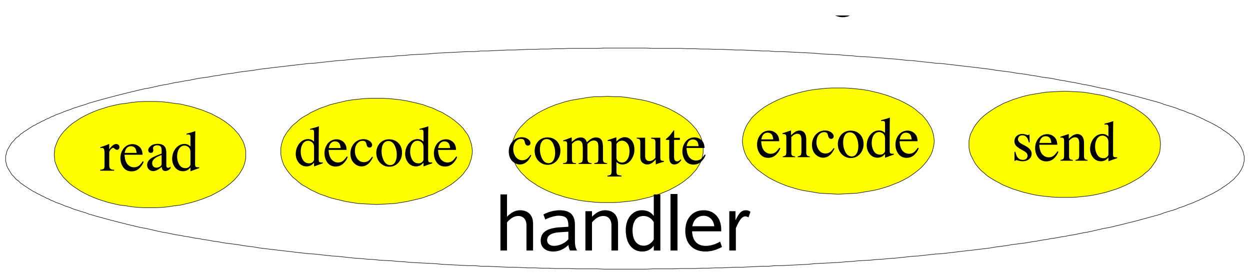
- 一个client发送上来的一条消息需要发送给所有的其他client（通常也包括这个client）
  - 每个client有一个发送线程（在接收线程之外）和一个发送队列，有一个中央机制遍历全部发送队列送消息
  - 有一个中央的发送线程和队列，遍历全部client发送消息
    - 可能被某个client堵住
    - 为每个client起一个线程发送一条消息

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

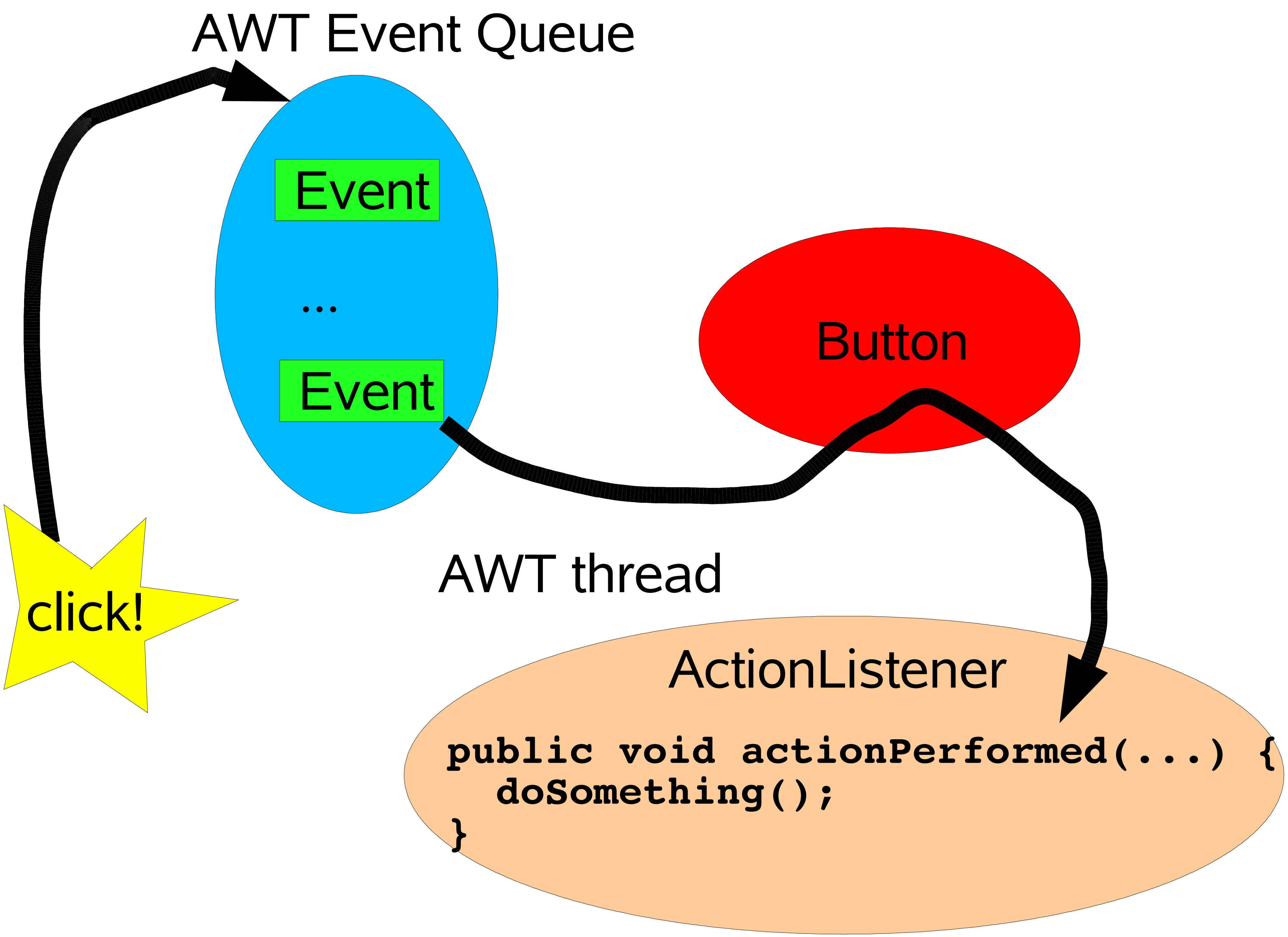


- Basic mechanisms supported in java.nio
  - Non-blocking reads and writes
  - Dispatch tasks associated with sensed IO events
- Endless variation possible
  - A family of event-driven designs

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

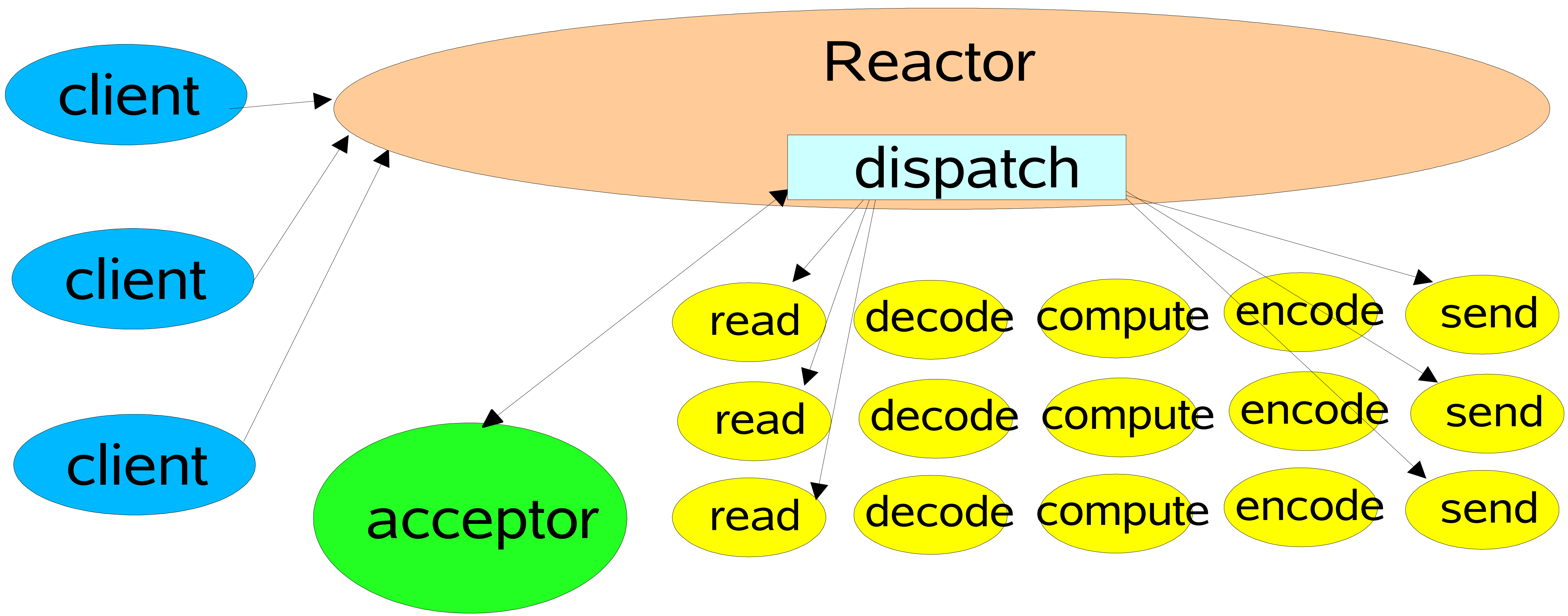
Background: Events in AWT



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

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();
  */
```

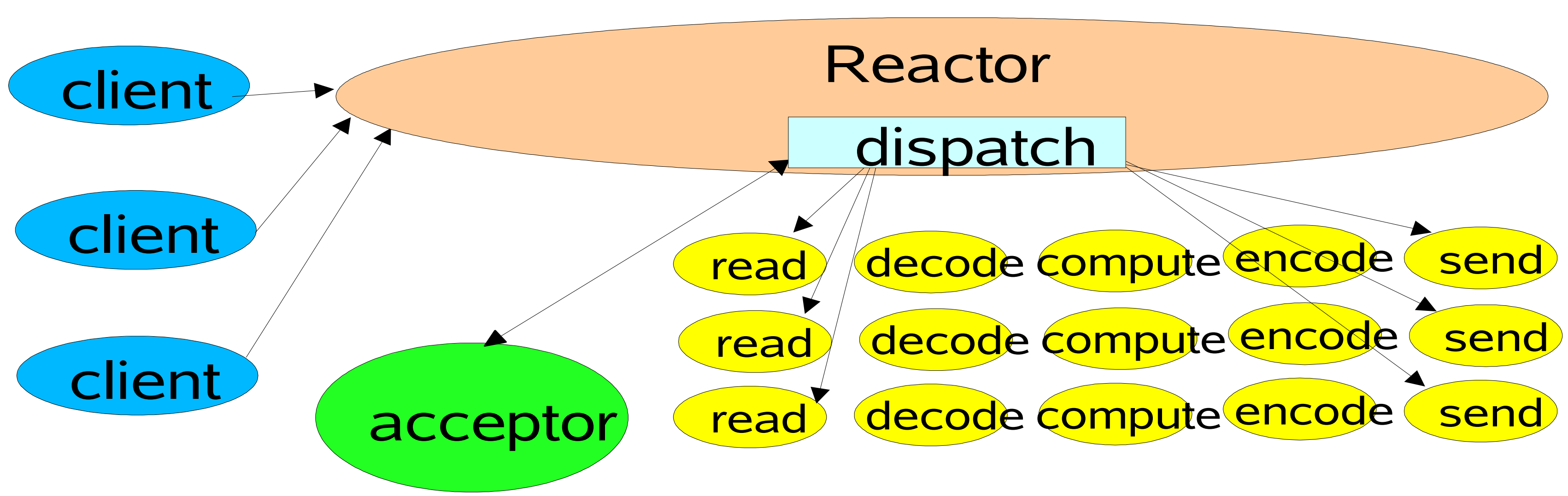
Reactor 2: Dispatch Loop

```
// 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

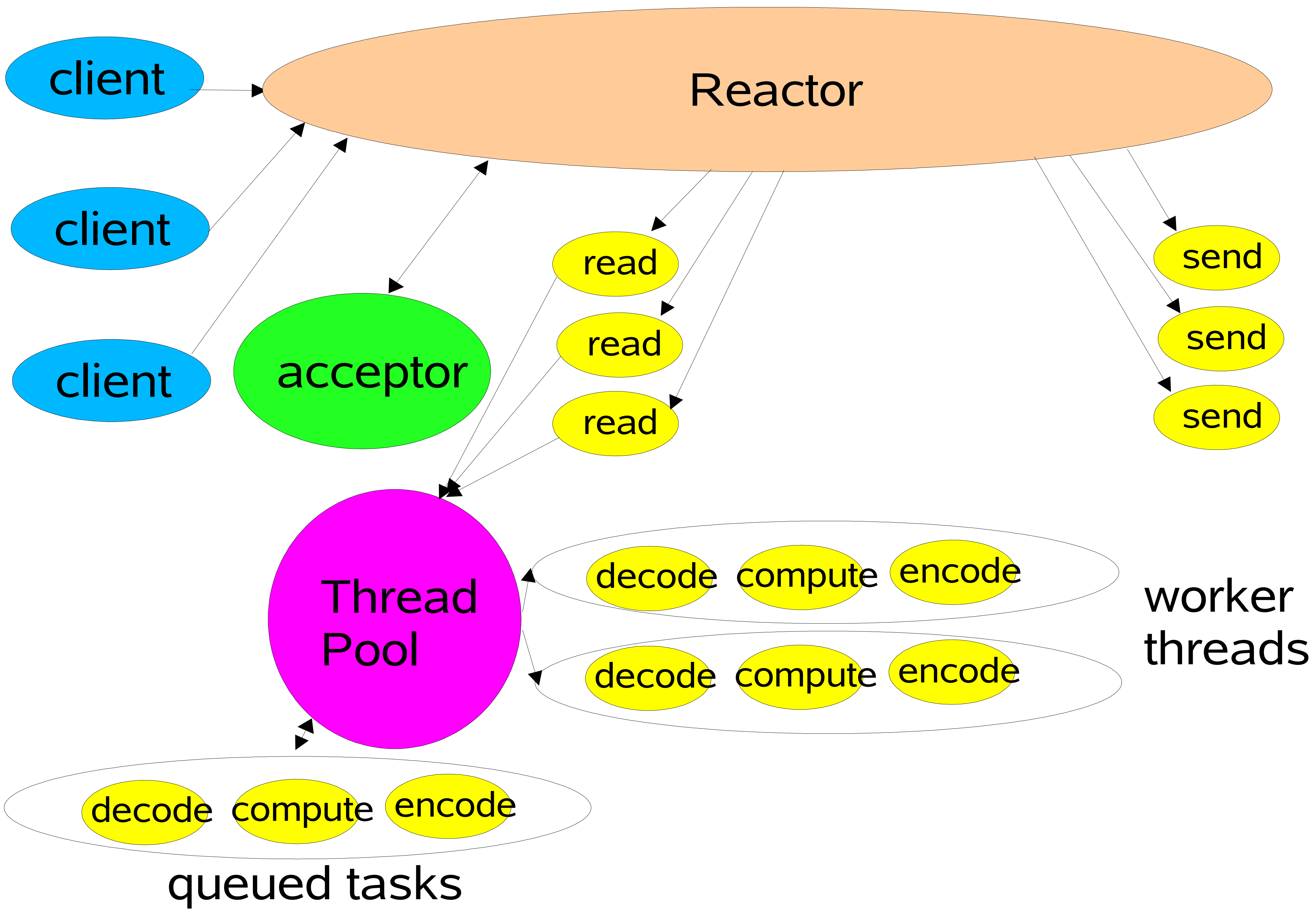
Multithreaded Designs

- 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

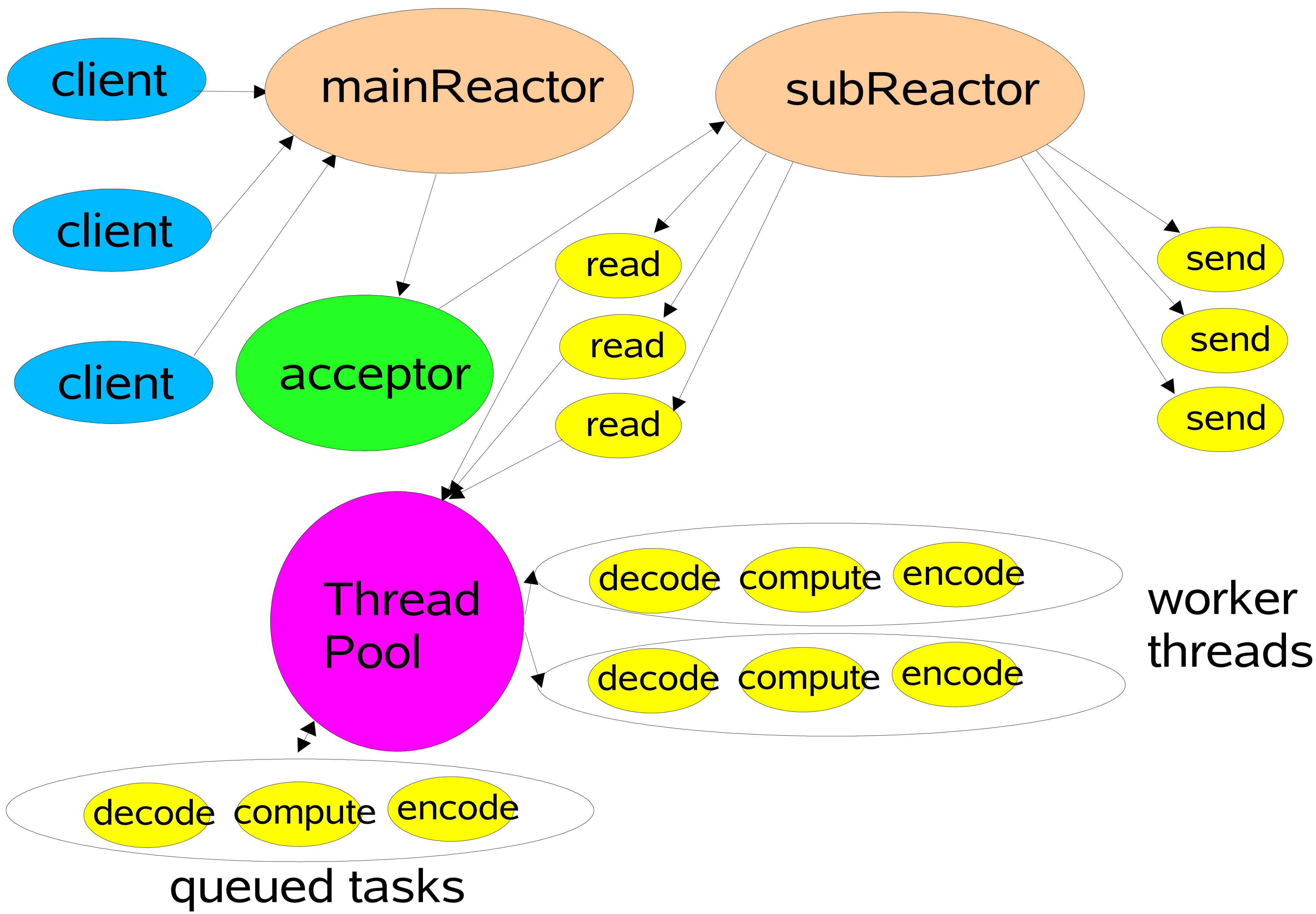
- A tunable worker thread pool
- Main method execute(Runnable r)
- Controls for:
  - The kind of task queue (any Channel)
  - Maximum number of threads
  - Minimum number of threads
  - "Warm" versus on-demand threads
  - Keep-alive interval until idle threads die
    - to be later replaced by new ones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



Why JNI?

* 使用现有的成熟的代码
* 访问系统特性或设备
* 代码运行速度很关键时

* 为函数保留一个代码位置

```
public class HelloNative {
 public native static void greeting();
 public static void main(String[] args) {
     greeting();
 }
}
```

* 编译该Java程序HelloNative.java

* 使用javah HelloNative来产生C语言头文件

* 编写C程序

* 编译生成DLL

vcvars32

cl –Ic:\jdk\include –Ic:\jdk\include\win32 –LD
HelloNative.c –FeHelloNative.dll

* 在Java程序中装载DLL

```
static {
    System.loadLibrary("HelloNative");
}
```

传递基本参数

boolean	jboolean	1
byte	jbyte	1
char	jchar	2
short	jshort	2
int	jint	4
long	jlong	8
float	jfloat	4
double	jdouble	8

传递String

- jstring
- jstring NewStringUTF(JNIEnv*, const char[]);
- jsize GetStringUTFLength(JNIEnv*, jstring)
- const jbyte* GetStringUTFChars(JNIEnv*, jstring string, jboolean*)