

Scalable IO in Java

Doug Lea

State University of New York at Oswego

`dl@cs.oswego.edu`

`http://gee.cs.oswego.edu`

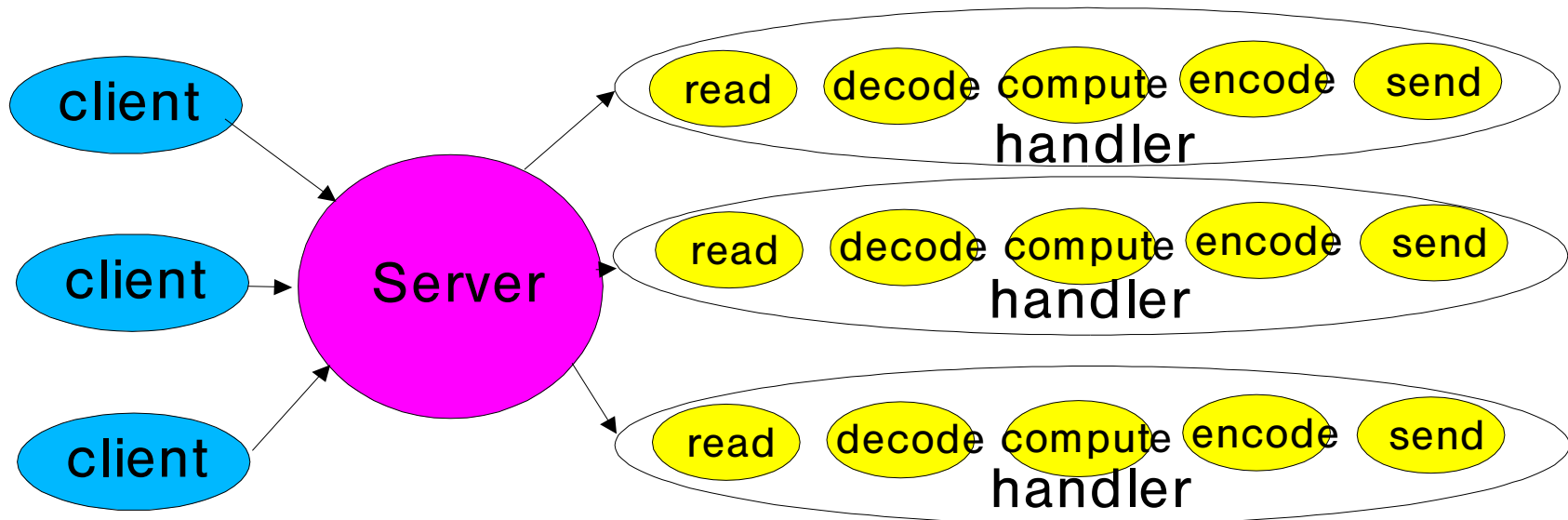
Outline

- Scalable network services
- Event-driven processing
- Reactor pattern
 - Basic version
 - Multithreaded versions
 - Other variants
- Walkthrough of java.nio nonblocking IO APIs

Network Services

- Web services, Distributed Objects, etc
- 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



Each handler may be started in its own thread

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) { /* ... */ }
    }
}
```

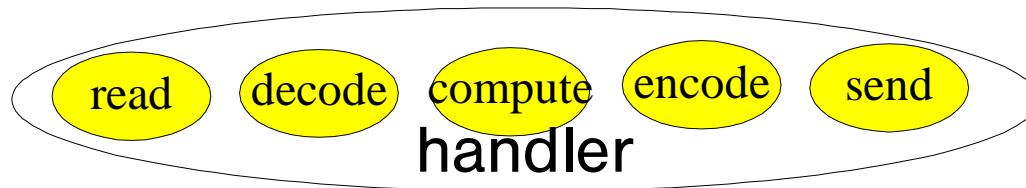
Note: most exception handling elided from code examples

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

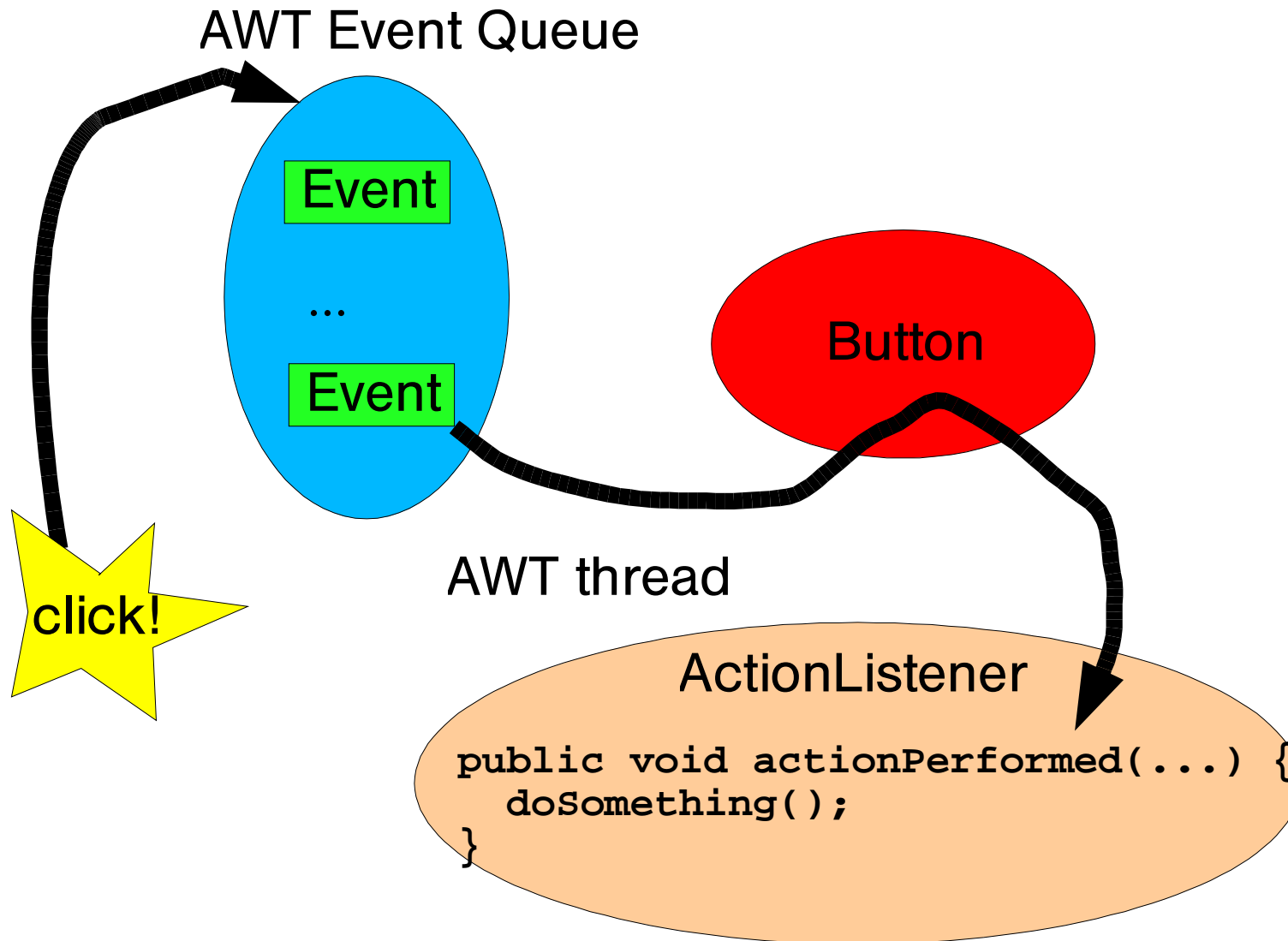


- Basic mechanisms supported in java.nio
 - **Non-blocking** reads and writes
 - **Dispatch** tasks associated with sensed IO events
- Endless variation possible
 - A family of event-driven designs

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

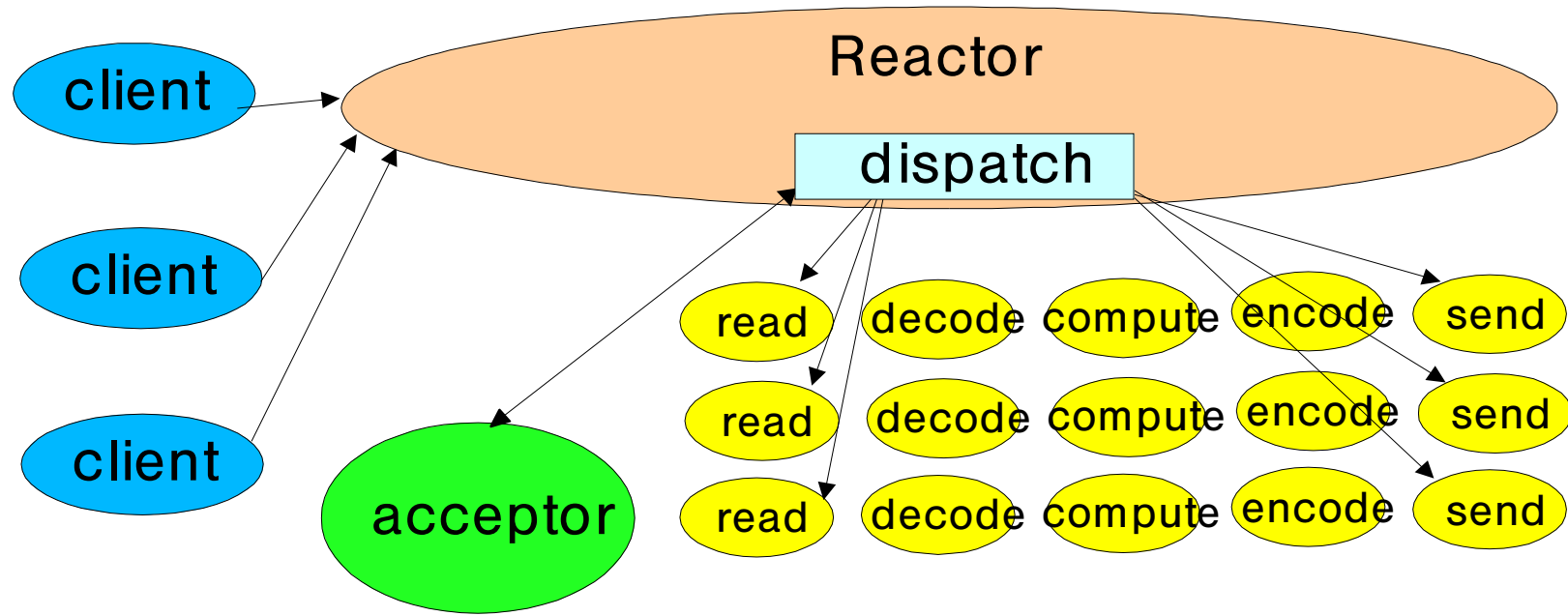


Event-driven IO uses similar ideas but in different designs

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.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();
    */
}
```

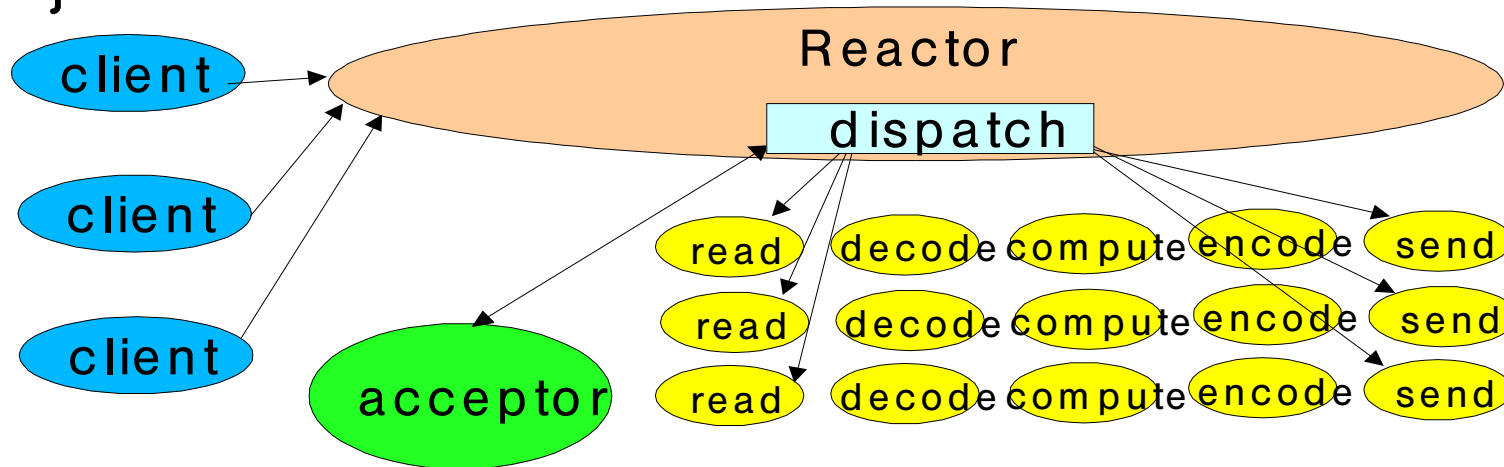
Reactor 2: Dispatch Loop

```
// 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;
        // 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

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

```
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();
        }
    }
}
```

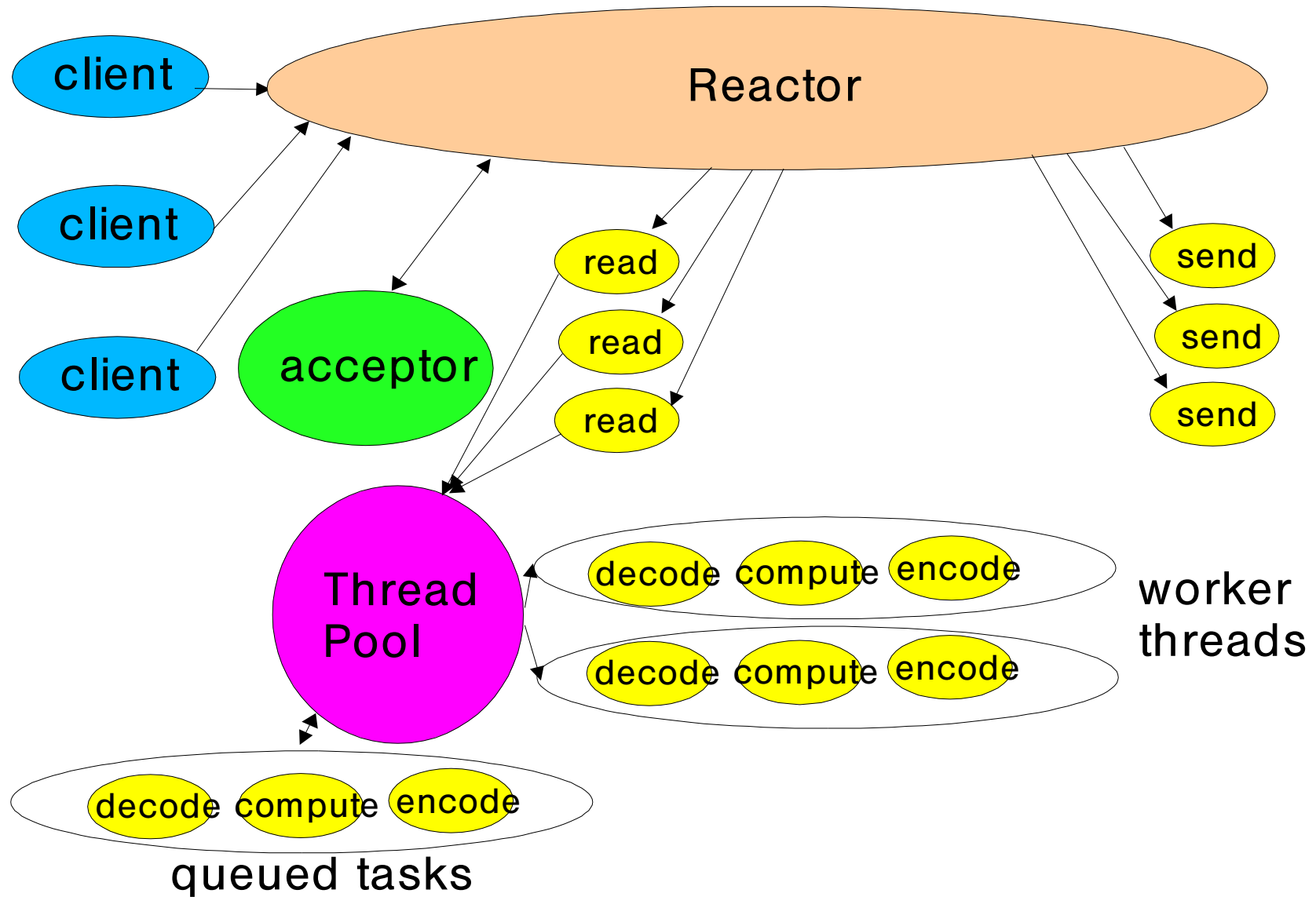
Multithreaded Designs

- 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 POA2 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

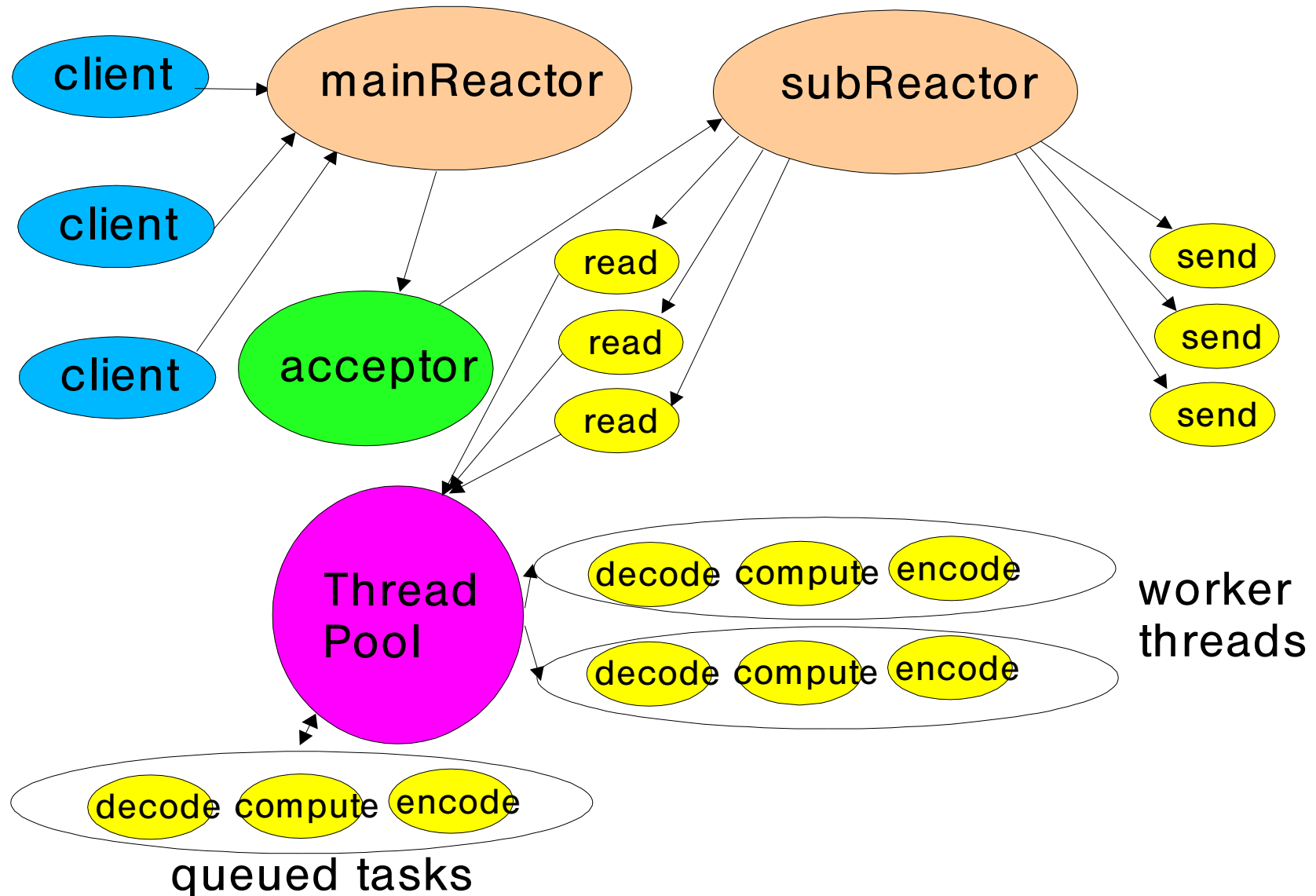
- A tunable worker thread pool
- Main method **execute(Runnable r)**
- Controls for:
 - The kind of task queue (any Channel)
 - Maximum number of threads
 - Minimum number of threads
 - "Warm" versus on-demand threads
 - Keep-alive interval until idle threads die
 - to be later replaced by new ones if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



Using other java.nio features

- Multiple Selectors per Reactor
 - To bind different handlers to different IO events
 - May need careful synchronization to coordinate
- File transfer
 - Automated file-to-net or net-to-file copying
- Memory-mapped files
 - Access files via buffers
- Direct buffers
 - Can sometimes achieve zero-copy transfer
 - But have setup and finalization overhead
 - Best for applications with long-lived connections

Connection-Based Extensions

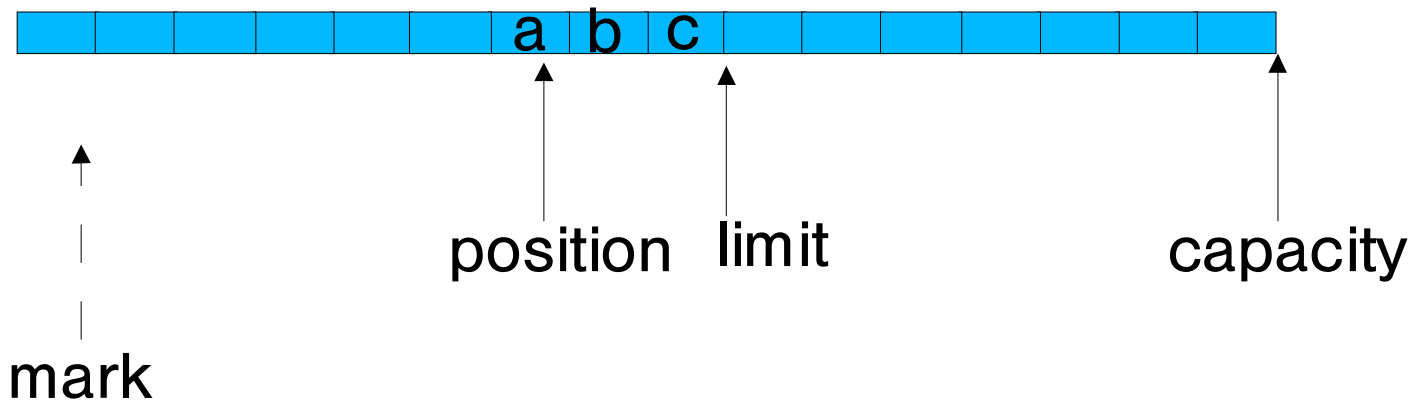
- Instead of a single service request,
 - Client connects
 - Client sends a series of messages/requests
 - Client disconnects
- Examples
 - Databases and Transaction monitors
 - Multi-participant games, chat, etc
- Can extend basic network service patterns
 - Handle many relatively long-lived clients
 - Track client and session state (including drops)
 - Distribute services across multiple hosts

API Walkthrough

- Buffer
- ByteBuffer
 - (CharBuffer, LongBuffer, etc not shown.)
- Channel
- SelectableChannel
- SocketChannel
- ServerSocketChannel
- FileChannel
- Selector
- SelectionKey

Buffer

```
abstract class Buffer {  
    int      capacity();  
    int      position();  
    Buffer    position(int newPosition);  
    int      limit();  
    Buffer    limit(int newLimit);  
    Buffer    mark();  
    Buffer    reset();  
    Buffer    clear();  
    Buffer    flip();  
    Buffer    rewind();  
    int      remaining();  
    boolean  hasRemaining();  
    boolean  isReadOnly();  
}
```



ByteBuffer (1)

```
abstract class ByteBuffer extends Buffer {
    static ByteBuffer allocateDirect(int capacity);
    static ByteBuffer allocate(int capacity);
    static ByteBuffer wrap(byte[] src, int offset, int len);
    static ByteBuffer wrap(byte[] src);

    boolean            isDirect();
    ByteOrder          order();
    ByteBuffer          order(ByteOrder bo);
    ByteBuffer          slice();
    ByteBuffer          duplicate();
    ByteBuffer          compact();
    ByteBuffer          asReadOnlyBuffer();
    byte               get();
    byte               get(int index);
    ByteBuffer          get(byte[] dst, int offset, int length);
    ByteBuffer          get(byte[] dst);
    ByteBuffer          put(byte b);
    ByteBuffer          put(int index, byte b);
    ByteBuffer          put(byte[] src, int offset, int length);
    ByteBuffer          put(ByteBuffer src);
    ByteBuffer          put(byte[] src);
    char               getChar();
    char               getChar(int index);
    ByteBuffer          putChar(char value);
    ByteBuffer          putChar(int index, char value);
    CharBuffer          asCharBuffer();
```

ByteBuffer (2)

```
short      getShort();
short      getShort(int index);
ByteBuffer putShort(short value);
ByteBuffer putShort(int index, short value);
ShortBuffer asShortBuffer();
int        getInt();
int        getInt(int index);
ByteBuffer putInt(int value);
ByteBuffer putInt(int index, int value);
IntBuffer  asIntBuffer();
long       getLong();
long       getLong(int index);
ByteBuffer putLong(long value);
ByteBuffer putLong(int index, long value);
LongBuffer asLongBuffer();
float      getFloat();
float      getFloat(int index);
ByteBuffer putFloat(float value);
ByteBuffer putFloat(int index, float value);
FloatBuffer asFloatBuffer();
double     getDouble();
double     getDouble(int index);
ByteBuffer putDouble(double value);
ByteBuffer putDouble(int index, double value);
DoubleBuffer asDoubleBuffer();
}
```


Channel

```
interface Channel {
    boolean isOpen();
    void close() throws IOException;
}

interface ReadableByteChannel extends Channel {
    int read(ByteBuffer dst) throws IOException;
}

interface WritableByteChannel extends Channel {
    int write(ByteBuffer src) throws IOException;
}

interface ScatteringByteChannel extends ReadableByteChannel {
    int read(ByteBuffer[] dsts, int offset, int length)
        throws IOException;
    int read(ByteBuffer[] dsts) throws IOException;
}

interface GatheringByteChannel extends WritableByteChannel {
    int write(ByteBuffer[] srcs, int offset, int length)
        throws IOException;
    int write(ByteBuffer[] srcs) throws IOException;
}
```

SelectableChannel

```
abstract class SelectableChannel implements Channel {
    int          validOps();
    boolean      isRegistered();

    SelectionKey keyFor(Selector sel);
    SelectionKey register(Selector sel, int ops)
        throws ClosedChannelException;

    void          configureBlocking(boolean block)
        throws IOException;

    boolean      isBlocking();
    Object       blockingLock();
}
```

SocketChannel

```
abstract class SocketChannel implements ByteChannel ... {
    static SocketChannel open() throws IOException;

    Socket    socket();
    int       validOps();
    boolean   isConnected();
    boolean   isConnectionPending();
    boolean   isInputOpen();
    boolean   isOutputOpen();

    boolean   connect(SocketAddress remote) throws IOException;
    boolean   finishConnect() throws IOException;
    void      shutdownInput() throws IOException;
    void      shutdownOutput() throws IOException;

    int       read(ByteBuffer dst) throws IOException;
    int       read(ByteBuffer[] dsts, int offset, int length)
                throws IOException;
    int       read(ByteBuffer[] dsts) throws IOException;

    int       write(ByteBuffer src) throws IOException;
    int       write(ByteBuffer[] srcs, int offset, int length)
                throws IOException;
    int       write(ByteBuffer[] srcs) throws IOException;
}
```

ServerSocketChannel

```
abstract class ServerSocketChannel extends ... {
    static ServerSocketChannel open() throws IOException;

    int          validOps();
    ServerSocket socket();
    SocketChannel accept() throws IOException;
}
```

FileChannel

```
abstract class FileChannel implements ... {
    int  read(ByteBuffer dst);
    int  read(ByteBuffer dst, long position);
    int  read(ByteBuffer[] dsts, int offset, int length);
    int  read(ByteBuffer[] dsts);
    int  write(ByteBuffer src);
    int  write(ByteBuffer src, long position);
    int  write(ByteBuffer[] srcs, int offset, int length);
    int  write(ByteBuffer[] srcs);
    long position();
    void position(long newPosition);
    long size();
    void truncate(long size);
    void force(boolean flushMetadataToo);
    int  transferTo(long position, int count,
                   WritableByteChannel dst);
    int  transferFrom(ReadableByteChannel src,
                     long position, int count);
    FileLock lock(long position, long size, boolean shared);
    FileLock lock();
    FileLock tryLock(long pos, long size, boolean shared);
    FileLock tryLock();
    static final int MAP_RO, MAP_RW, MAP_COW;
    MappedByteBuffer map(int mode, long position, int size);
}
```

NOTE: ALL methods throw IOException

Selector

```
abstract class Selector {  
    static Selector open() throws IOException;  
    Set  keys();  
    Set  selectedKeys();  
    int  selectNow() throws IOException;  
    int  select(long timeout) throws IOException;  
    int  select() throws IOException;  
    void wakeup();  
    void close() throws IOException;  
}
```

SelectionKey

```
abstract class SelectionKey {
    static final int    OP_READ,      OP_WRITE,
                      OP_CONNECT, OP_ACCEPT;

    SelectableChannel channel();
    Selector           selector();
    boolean            isValid();
    void               cancel();
    int                interestOps();
    void               interestOps(int ops);
    int                readyOps();
    boolean            isReadable();
    boolean            isWritable();
    boolean            isConnectable();
    boolean            isAcceptable();
    Object             attach(Object ob);
    Object             attachment();
}
```