Distributed Systems

José Orlando Pereira

HASLab / Departamento de Informática
Universidade do Minho

2011/2012

Case study

- Simple echo server:
  - Echos all bytes received
  - Handle multiple concurrent clients
- Implementation:
  - Listens on port 12345
  - Read and write back

Sockets in java.net

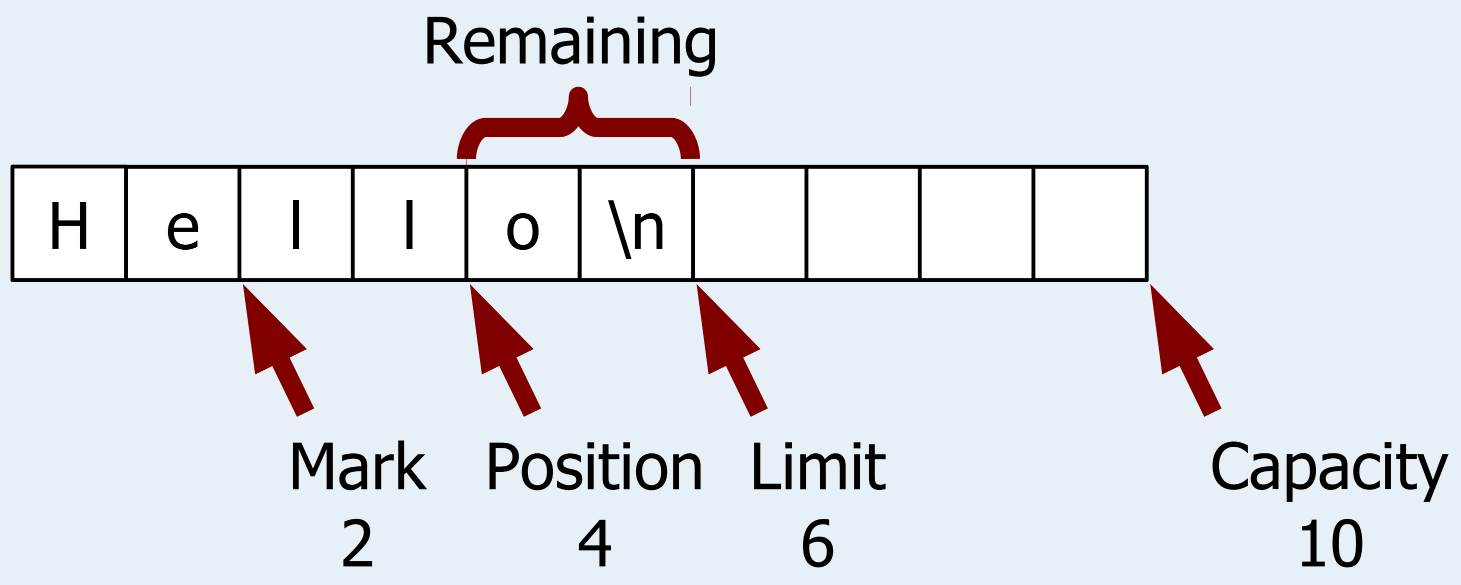
```
ServerSocket ss=new ServerSocket(12345);

while(true) {
        Socket s=ss.accept();

        InputStream is=s.getInputStream();
        OutputStream os=s.getOutputStream();

        // i/o

        s.close();
}
```

Sockets in java.nio

```
ServerSocketChannel ss=SelectorProvider.provider().openServerSocketChannel();
ss.socket().bind(new InetSocketAddress(12345));

while(true) {
	SocketChannel s=ss.accept();

	// i/o

	s.close();
}
```

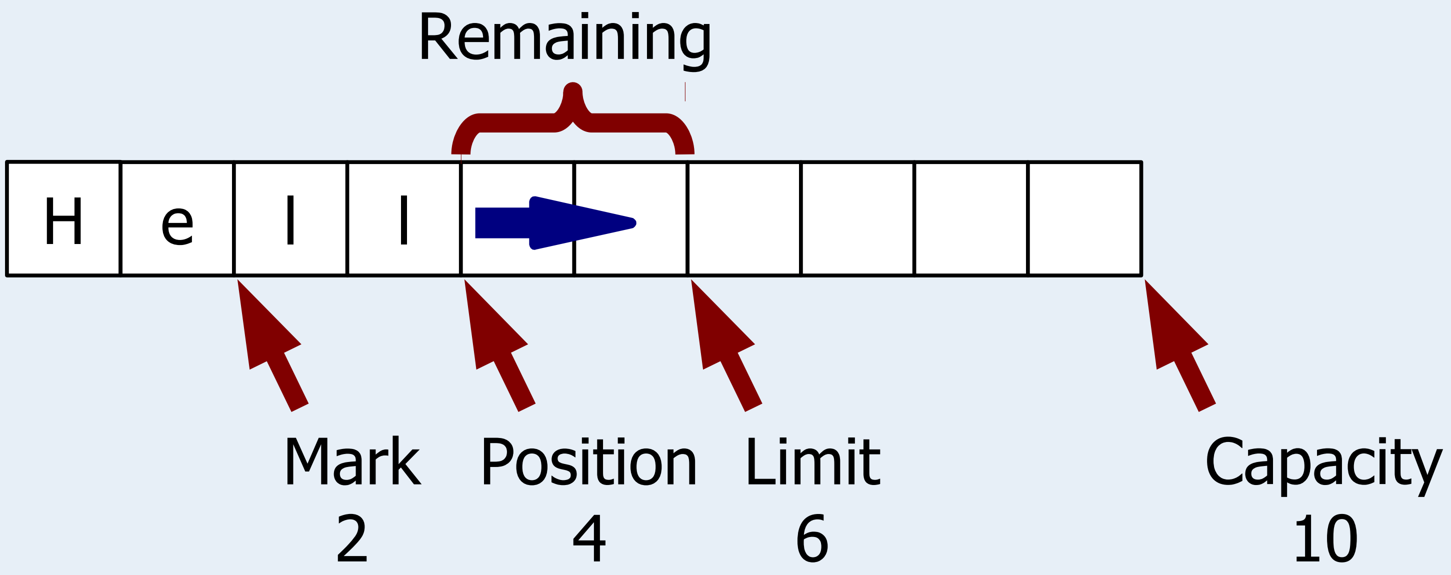
Buffers in java.nio

- Buffer = Array + Indexes:



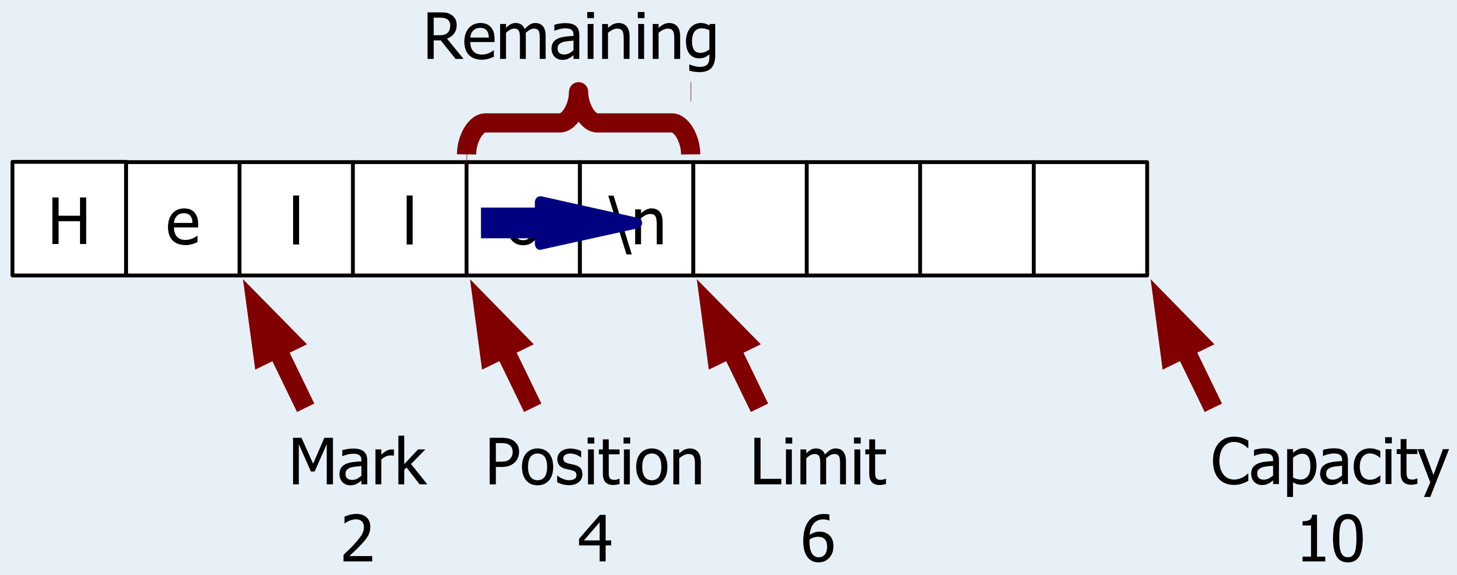
Buffers in java.nio

- Put/read: advances position, sets content

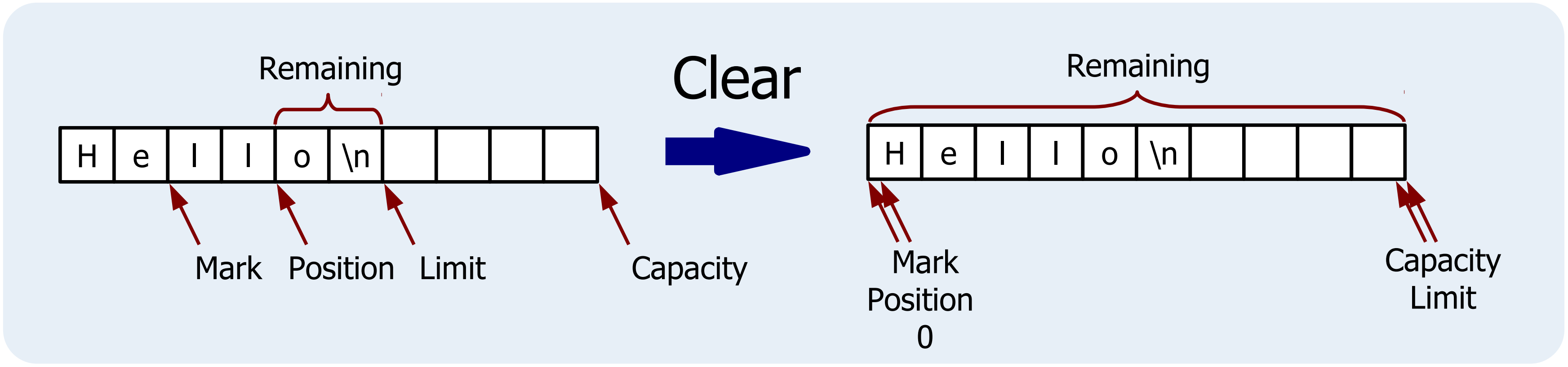


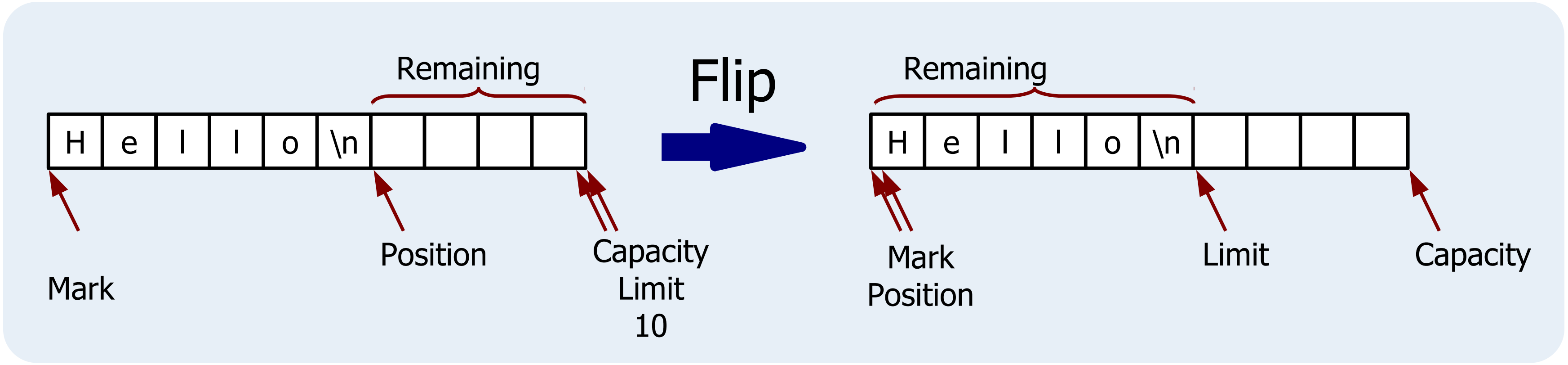
Buffers in java.nio

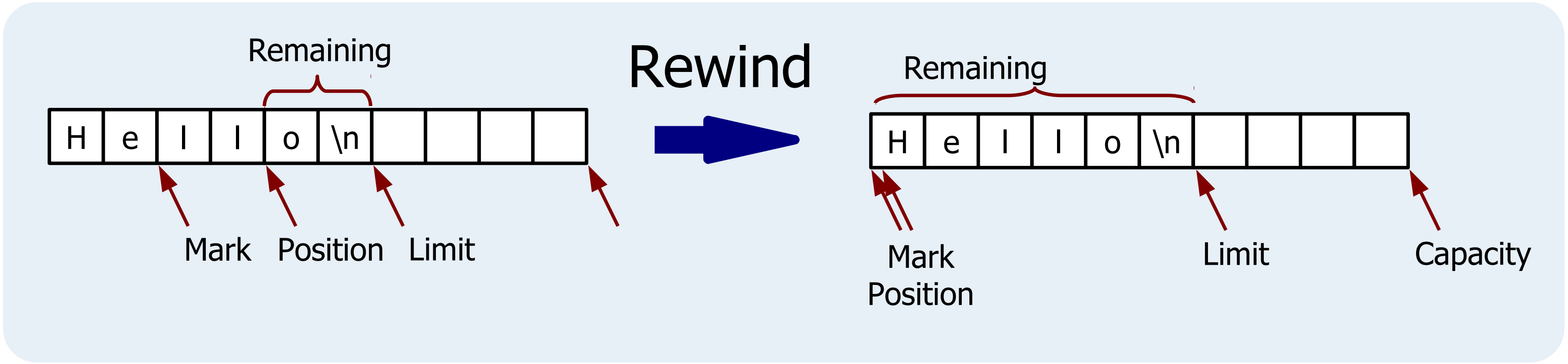
- Get/write: advances position, gets content



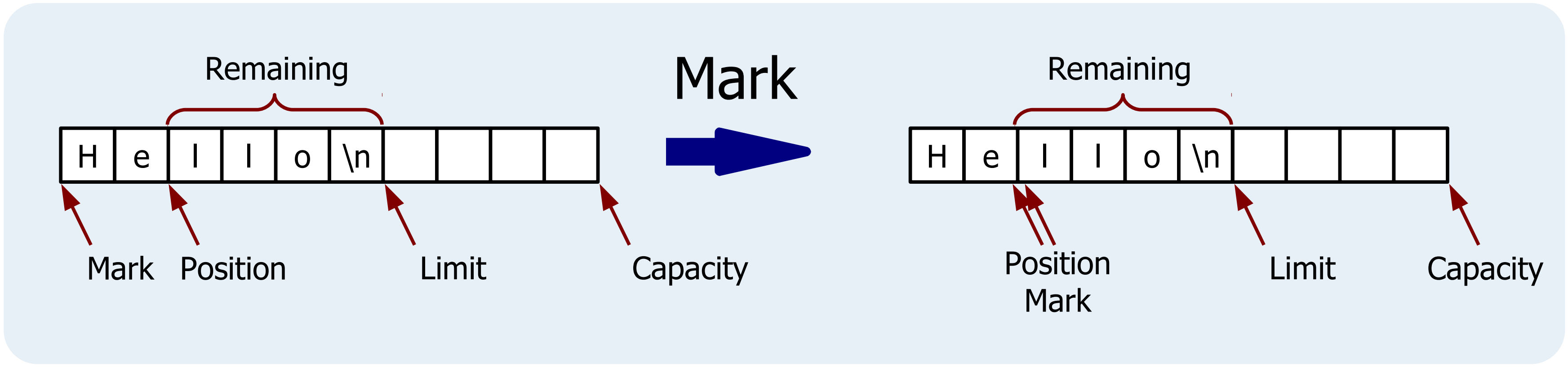
Buffers in java.nio

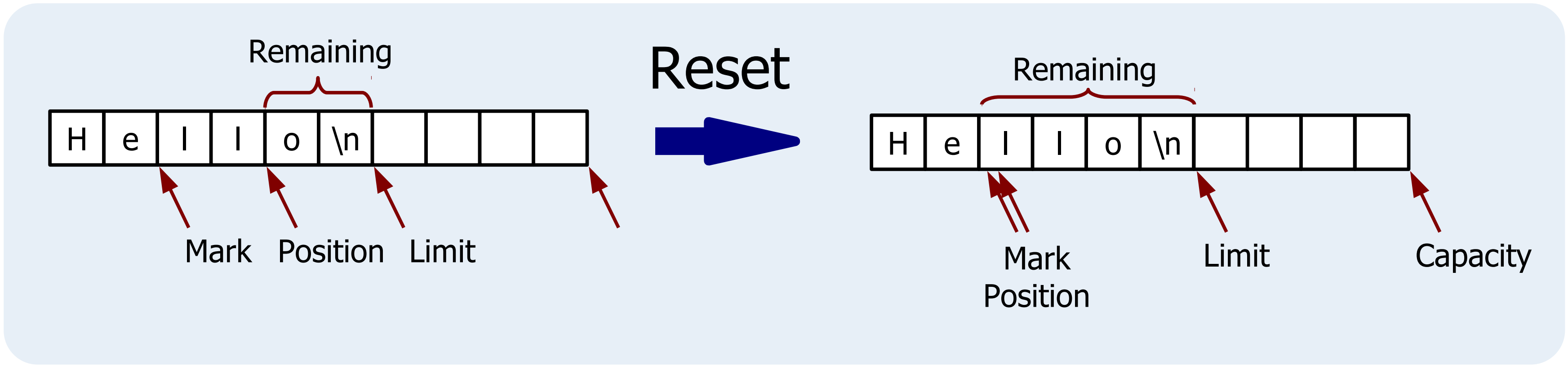






Buffers in java.nio





Sockets in java.nio

```
ServerSocketChannel ss=SelectorProvider.provider().openServerSocketChannel();
ss.socket().bind(new InetSocketAddress(12345));

while(true) {
    SocketChannel s=ss.accept();

    ByteBuffer buf=ByteBuffer.allocate(100);

    while(s.read(buf)>0) {
        buf.flip();
        s.write(buf);
        buf.clear();
    }

    s.close();
}
```

Non-blocking I/O

- How to service multiple sockets?
  - Multiple threads
  - Polling with a single thread
- Efficient polling:
  - Use select() to wait for I/O
  - Execute I/O operation without blocking

Non-blocking I/O

```
ServerSocketChannel ss=SelectorProvider.provider().openServerSocketChannel();
ss.socket().bind(new InetSocketAddress(12345));

ss.configureBlocking(false);

Selector sel=SelectorProvider.provider().openSelector();
ss.register(sel, SelectionKey.OP_ACCEPT);

while(true) {
    sel.select();

    for(Iterator<SelectionKey> i=sel.selectedKeys().iterator(); i.hasNext(); ) {
        SelectionKey key = i.next();

        // i/o

        i.remove();
    }
}
```

Non-blocking I/O

```
if (key.isAcceptable()) {
        SocketChannel s=ss.accept();

        s.configureBlocking(false);
        s.register(sel, SelectionKey.OP_READ);
}
```

Non-blocking I/O

```
if (key.isReadable()) {
    ByteBuffer buf=ByteBuffer.allocate(100);
    SocketChannel s=(SocketChannel)key.channel();

    int r=s.read(buf);
    if (r<0) {
        key.cancel();
        s.close();
    } else {
        buf.flip();
        s.write(buf);
    }
}
```

What if write would block?

Non-blocking I/O

- Need to poll before writing
- Bytes read must be saved until writing is possible
- Signal interest on writing

```
if (key.isReadable()) {

    ...

    } else {
        buf.flip();
        key.attach(buf);
        key.interestOps(SelectionKey.OP_WRITE);
    }
}
```

Context

Non-blocking I/O

- Get bytes attached to key
- Reset interest to reading

```
if (key.isWritable()) {
        SocketChannel s=(SocketChannel)key.channel();
        ByteBuffer buf=(ByteBuffer)key.attachment();

        s.write(buf);
        key.interestOps(SelectionKey.OP_READ);
}
```

Object oriented + Event driven

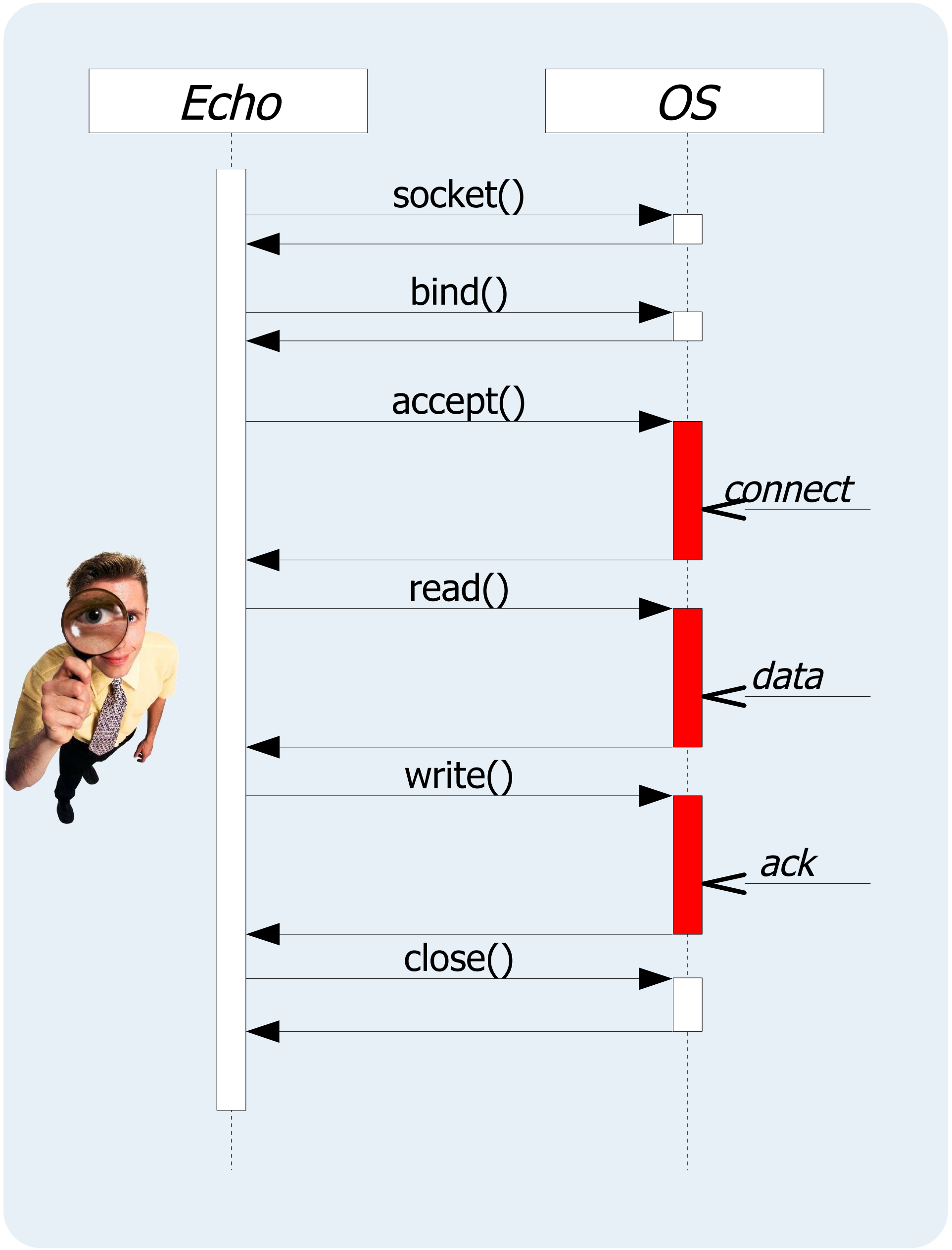
- Encapsulate context data + event-handling code

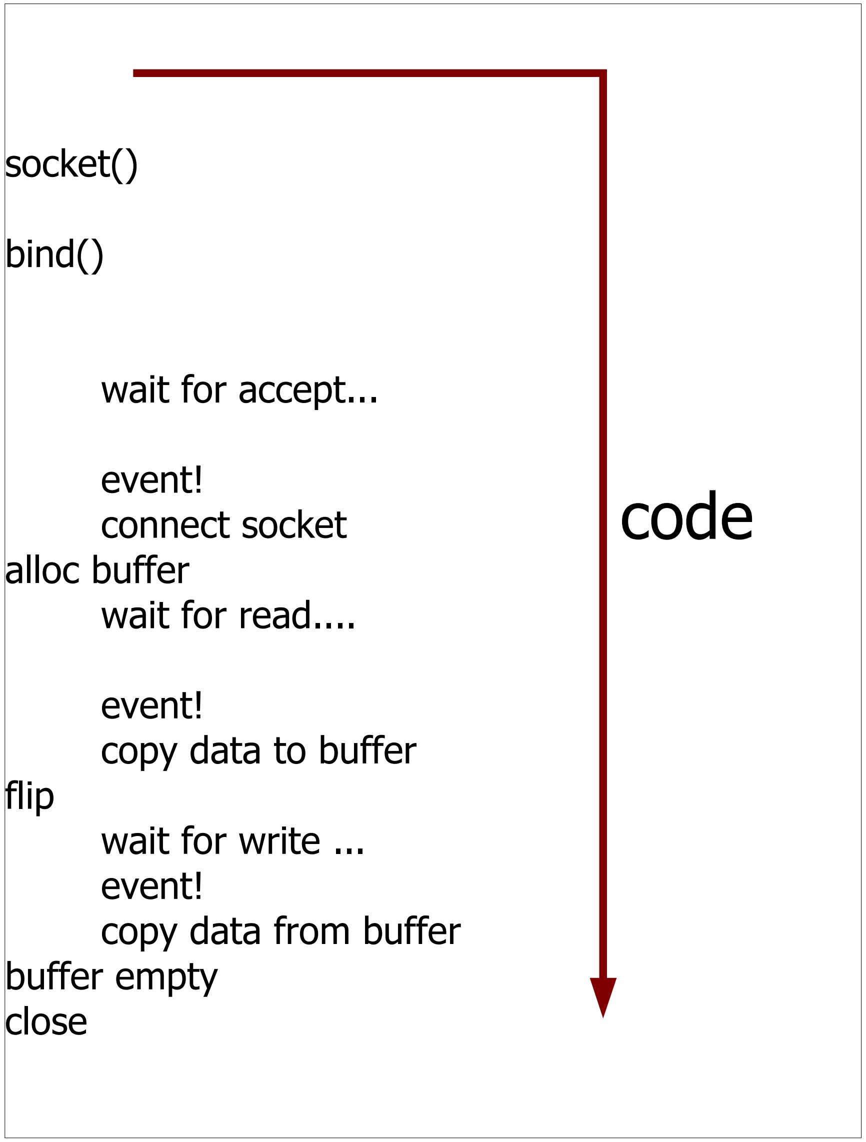
```
public class Echo {
	private ByteBuffer buf;
	// ...

	public Echo(SelectionKey key) {
		// initialization
	}

	public void handleRead() throws IOException {
		// input
	}

	public void handleWrite() throws IOException {
		// output
	}
}
```

Object oriented + Event driven

```
if (key.isAcceptable()) {
    SocketChannel s=ss.accept();

    if (s!=null) {
        s.configureBlocking(false);
        SelectionKey nkey=s.register(sel, SelectionKey.OP_READ);
        nkey.attach(new Echo(nkey));
    }
} else if (key.isReadable()) {
    Echo echo=(Echo)key.attachment();
    echo.handleRead();
} else if (key.isWritable()) {
    Echo echo=(Echo)key.attachment();
    echo.handleWrite();
}
```

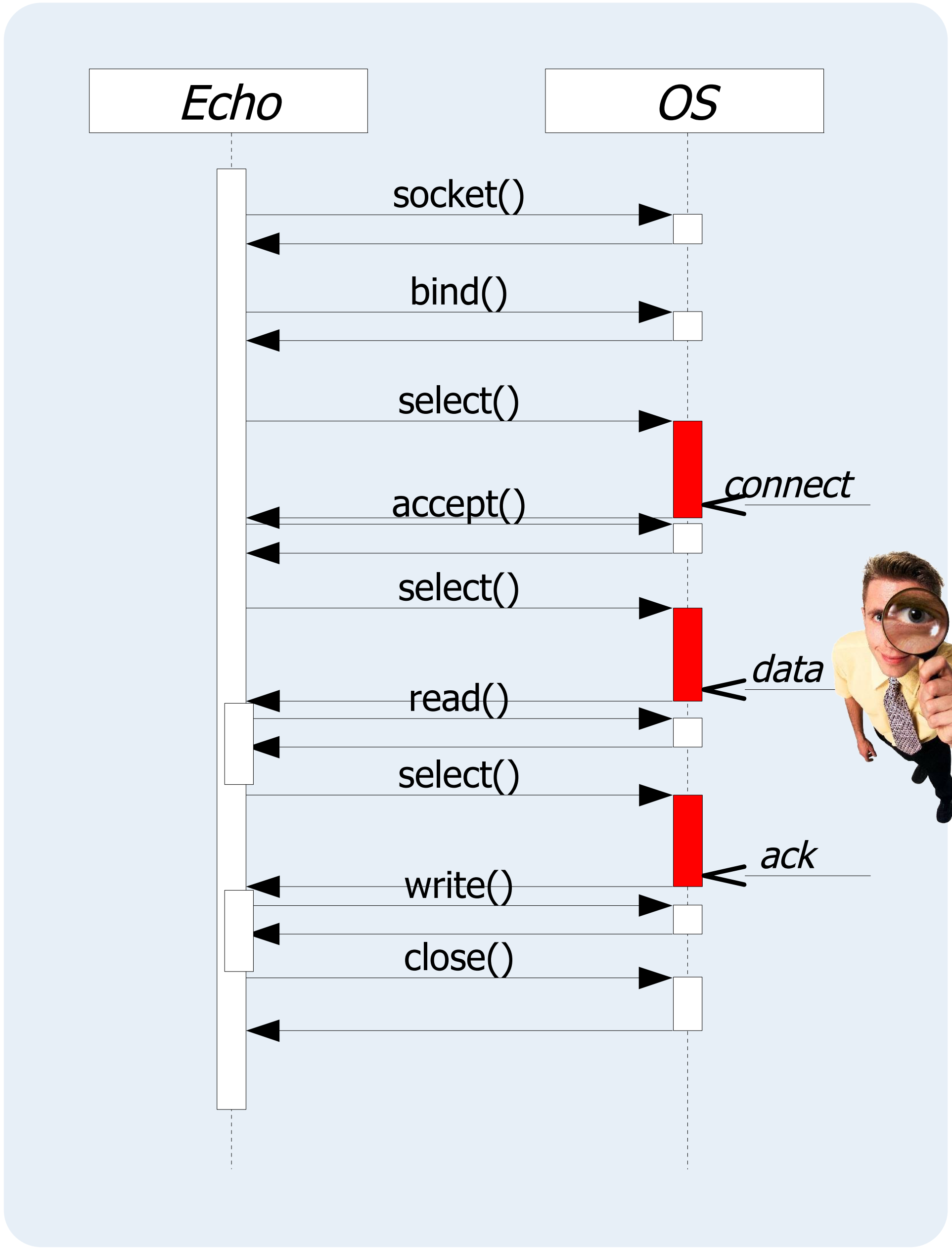
Threaded version

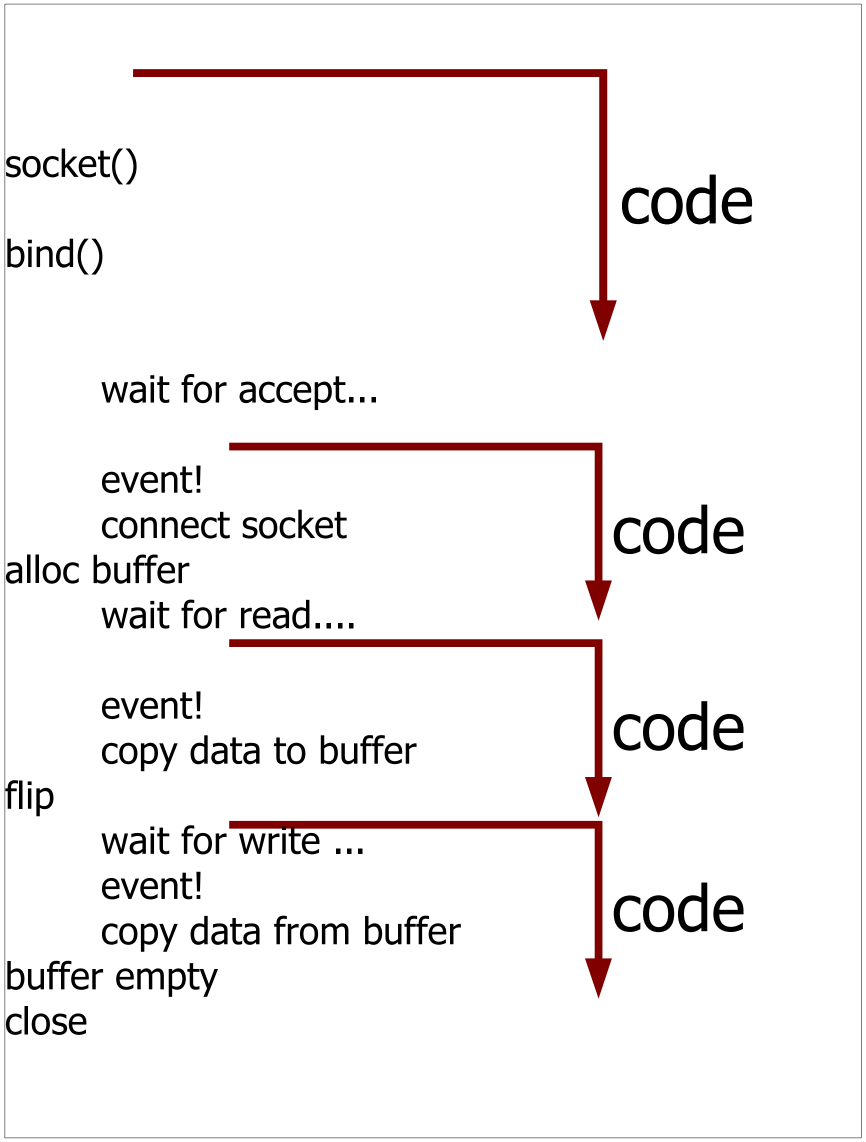




```
socket()

bind()


        wait for accept...

        event!
        connect socket
alloc buffer
        wait for read....

        event!
        copy data to buffer
flip
        wait for write ...
        event!
        copy data from buffer
buffer empty
close
```

Event-driven version





Conclusions

- The program can be regarded as either:
  - Being suspended waiting for something to happen
  - Executing some code deterministically without external intervention

Conclusions

- Both versions of the program:
  - Wait for the same conditions to start executing
  - Execute the same code as a consequence
- If we label such conditions and sections of code equally and log their execution:
  - We cannot distinguish their log files
- If we log the values of key variables in both programs when suspended:
  - We cannot distinguish their log files