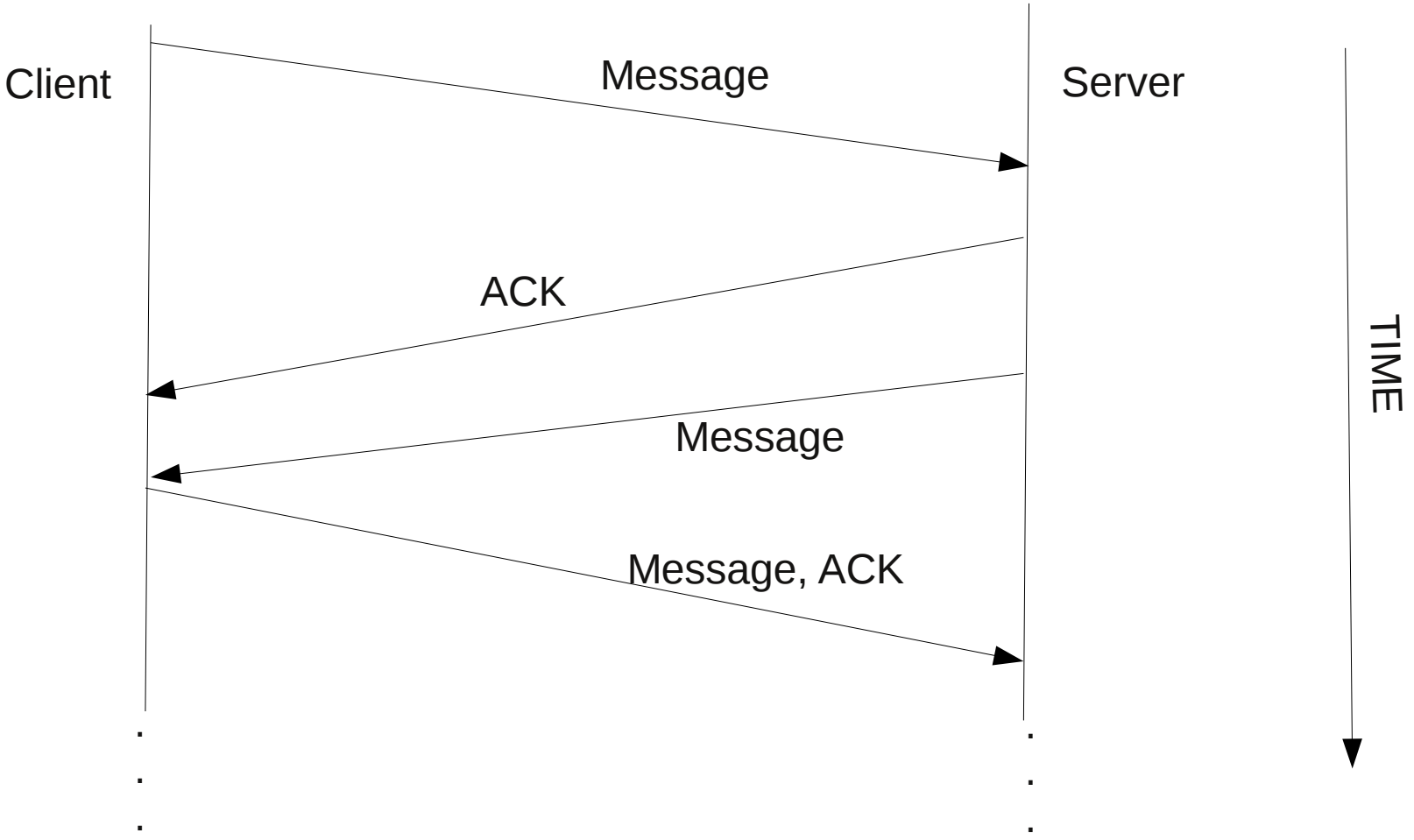
CS 360

TCP, Streams, Threads, States

TCP

- A reliable, connection based protocol



States

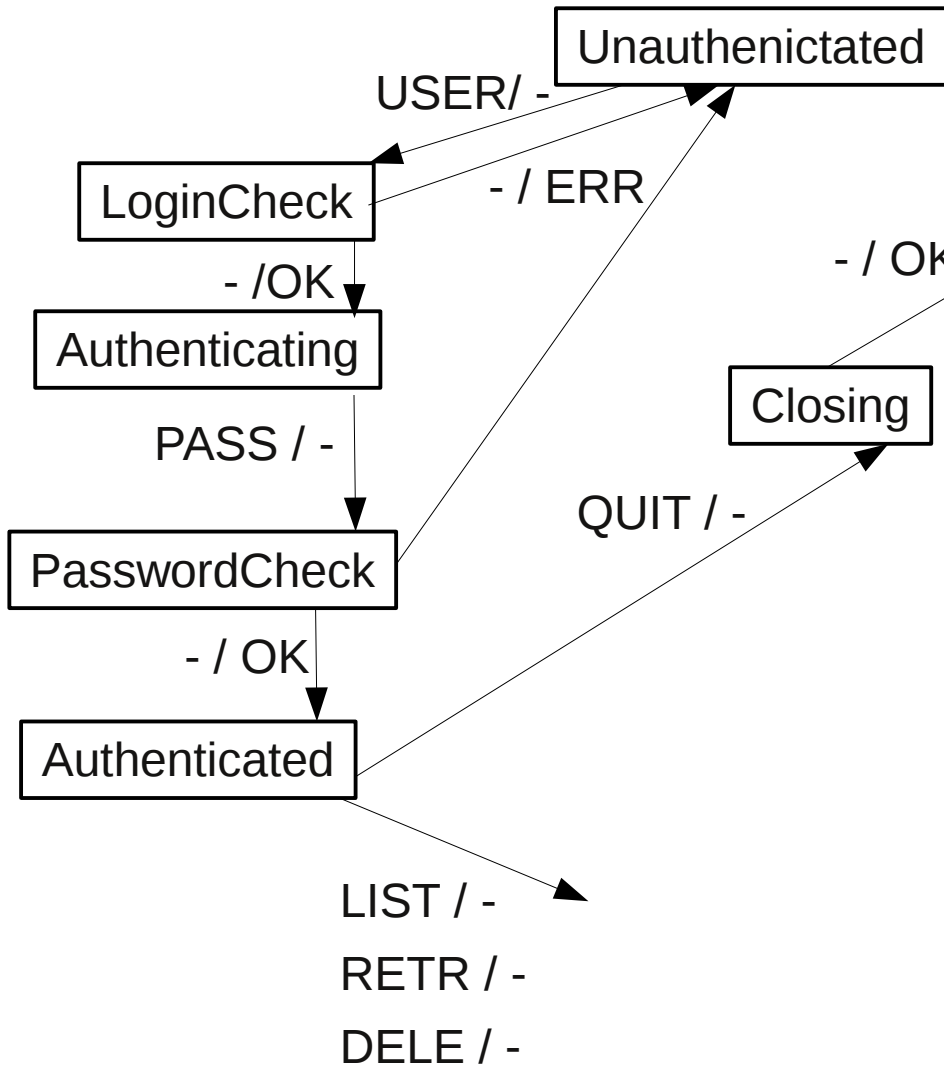
- What states can the protocol be in?
  - connecting, connected, ..., disconnecting, disconnected....
  - how can we model this?

```
public enum ProtocolState // looks like a class!
{
  CONNECTING,
  CONNECTED,
  ...
  DISCONNECTED;
}
```

State Diagram

POP3 Client Authentication

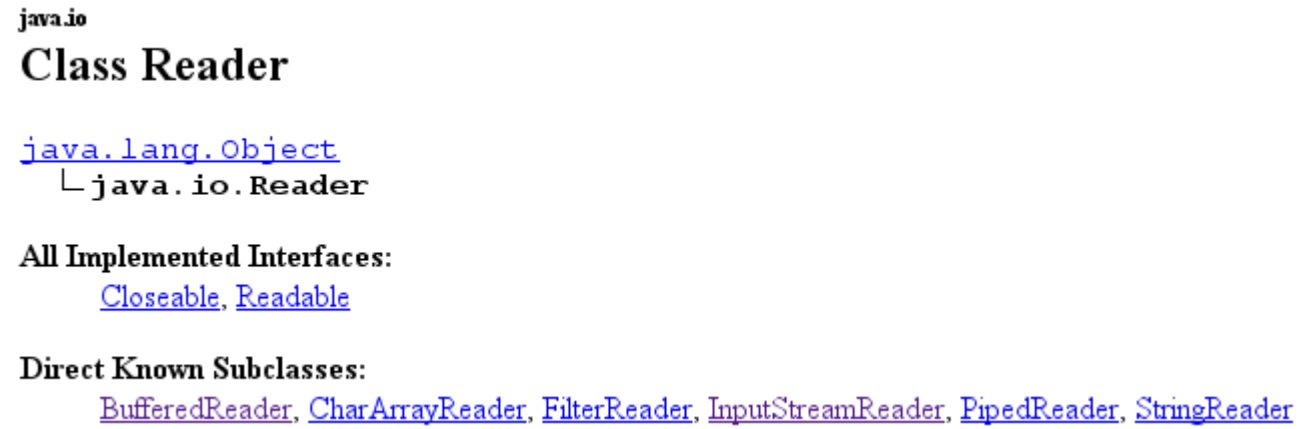
Client / Server




```
zeus$ telnet pop3.mailexample.net 110
+OK POP3 server ready
USER chadd
+OK
PASS iwantmail
+OK login successful
LIST
1 2505
2 14302
.
RETR 1
(send message 1)
DELE 1
QUIT
+OK POP3 server disconnecting
```

TCP Streams

- socket.getInputStream()
- socet.getOutputStream()
- Writer/Reader



java.io

Class Reader

java.lang.Object
  └java.io.Reader

All Implemented Interfaces:
Closeable, Readable

Direct Known Subclasses:
BufferedReader, CharArrayReader, FilterReader, InputStreamReader, PipedReader, StringReader

```
// client
Socket socket = new Socket();
socket.connect(sAddr); // generate the sAddr

bWrite = new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream()));

bRead = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
```

BufferedReader - readLine

Read a line of text until the newline character. The newline character is not returned.

readLine

```
public String readLine()
                    throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:

IOException - If an I/O error occurs

BufferedReader - mark/reset

- mark() - mark a byte in the stream.
- reset() - move back to the previous mark on the next read

- Useful to recover from an error situation

- ready() - will the next read block?

BufferedWriter

- Buffered: wait until the buffer fills to write data to the underlying structure
- write(String) (inherited from java.io.Writer)
- flush()

Design

- We must be able to build a command line and Android client
- We must share code with the Server
- We must be able to handle protocol changes
- Lots of a SYNCHRONICITY

Possible Client Design

PUIM CommandLine|Android Client

contains PUIM Client

read/write to UI

// interface
// callbacks
displayTextMessage()
displayErrorMessage()

PUIM Client

handle connection

manage state

read/write network

// API
connect()
registerUI()
sendMessage()
disconnect()

This might be a Thread.

PUIMProtocol

States
Messages

Callbacks / Event Listener

```
public class Beeper ... implements ActionListener {
    ...
    //where initialization occurs:
        Button button = new Button();
        button.addActionListener(this);

    public void actionPerformed(ActionEvent e) {
      // button was pressed!
    }
}


public class Button {
    ActionListener theListener;
    public void addActionListener(ActionListener listener)
    {
      theListener = listener;
    }
    public void actionPerformed(ActionEvent e)
    {
      theListener.actionPerformed(e)
    }
}
```


http://docs.oracle.com/javase/tutorial/uiswing/events/intro.html

Callbacks

```
public class Client ... implements PUIMClientHandler {

    PUIMReadThread readThread = new PUIMReadThread();
    readThread.addClient(this);

    public void displayTextMessage(String from, String msg) {
      // display the message
    }
}
```

```
public class PUIMReadThread {
    PUIMClientHandler theClient;
    public void addClient(PUIMClientHandler client)
    {
      theClient = client;
    }
    public void messageReceived(String from, String msg)
    {
      theClient.displayTextMessage(from, msg);
    }
}
```

Threads

```
public class MyThread implements Runnable
{

   @Override
   public void run()
   {
   }

}

MyThread myThread = new MyThread();
Thread listenThread = new Thread(myThread);

listenThread.start(); // call run
```

Synchronization

- Some data structures are thread safe:
  - ConcurrentHashMap<K, V>
  - ConcurrentLinkedQueue<E>

- Often threads will pass messages to each other through Queues

```
ConcurrentLinkedQueue<String> messageQueue =
   new ConcurrentLinkedQueue<String>();
```

Sychronization

- synchronized method

```
public synchronized void once()
{
    // only one thread can be executing this
    // method, per object!
}
```

- synchronized block

```
synchronized(this) // every object contains a lock
{
    this.x++;
}
```

Producer/Consumer

- Design Pattern
- One thread produces data, the other consumes data.

```
public void consume()
{
    synchronized (queue)
    {
        while(queue.isEmpty())
        {
            queue.wait();
            // wait releases the lock until
            // notify is used
        }
        msg = queue.remove();
    }
    // do work with msg
}
```

```
public void produce(msg)
{
    synchronized (queue)
    {
        queue.add(msg);
        queue.notify()
    }
}
```