One of the significant disadvantages of Java RMI is that it relies on the assumption that both client and server run within a JVM.

What if we want to:
- Integrate legacy code written in COBOL
- Use a remote server developed by someone else using Python, C, ...
- Make the server available for others to integrate with

JNI (Java Native Interface) could be a work-around in some cases, but there is a significantly increased software engineering burden.

What solutions?
- Use a protocol that all machines implement — Sockets
- Use a standard that supports many languages — CORBA
- Use a language agnostic infrastructure — MQ
Sockets (aka. “Hard Jexit”)  

- Throw away all the support structures that RMI provides
  - `rmiregistry` lookup
  - Dynamic loading of stub classes
  - Easy serialisation/deserialisation of parameters
  - RMIP for client-server coordination, including handling of remote exceptions
  - Utilisation of Java’s security model
  - Activatable objects for scalability

- Note: blurring the distinction between local and remote is A Bad Thing, so we won’t miss that
  - A local method call should never be thought of in the same terms as a remote call
  - Make sure you document what is remote (in code and other documentation)
  - Make sure you handle issues that may arise due to the network

- All networked computers will implement transport layer protocols
- All languages will provide a Socket API — a programmatic interface between a user process and the transport layer
TCP Sockets

**Client**

- Create a **Socket** to connect to remote host and port
- *Create an InputStreamReader to read from socket*
- *Create an OutputStreamWriter to write to this socket*
- Execute protocol; while (continue)
  - Write to socket
  - Read from socket (wait)
- Close socket

**Server with specific IP address**

- Create a **ServerSocket** on a specific port
- Start **listening** on this port
- Accept connection request and create a new socket
- *Create an InputStreamReader to read from socket*
- *Create an OutputStreamWriter to write to this socket*
- Execute protocol; while (continue)
  - Read from socket (wait)
  - Write to socket
- Close socket
The IRC Chat Example

- Server needs to listen for connection requests using a ServerSocket
- A new thread is required to serve each new connection Socket

```java
package comp2207.irc;

// Imports from java.io and java.net
public class ChatServer {

  void startListening(int port) throws IOException {
    ServerSocket listener = new ServerSocket(port);
    while (true) {
      Socket client = listener.accept();
      new ServerThread(client).start();
    }
  }

  public static void main(String[] args) throws IOException {
    if (args.length != 1) {
      System.out.println("Usage: java ChatServer <port>");
      return;
    }
    new ChatServer().startListening(Integer.parseInt(args[0]));
  }
}
```
In the thread managing the socket, I/O Streams are created. The protocol is initiated (client must expect this message).
IRC Protocol

- **IRC** = Internet Relay Chat
- After connection, server responds with **welcome message**
- Server expects:
  - **JOIN** name — this must be the first line sent by client. Server will check name. Connection closed by server if this is not the structure of the first message, or if the same name has been registered
  - **YELL** message — will be sent to all registered clients
  - **TELL** name message — will be sent to the named person if they are registered
  - **EXIT** — the user is unregistered and the server closes the connection
- The **server** needs to **parse** each line sent by the client
- In this case, the protocol just involves exchange of **byte streams**
public void run() {
    try {
        // Tokeniser class (see code) used to parse client input
        ReqTokenizer reqTokenizer = new ReqTokenizer();
        // First, the client must register.
        Token token = reqTokenizer.getToken(_clientIn.readLine());
        if (!(token instanceof JoinToken)) {
            _clientSocket.close();
            return;
        }
        // Check the client’s registration request
        token = reqTokenizer.getToken(_clientIn.readLine());
        while (!(token instanceof ExitToken)) {
            if (token instanceof YellToken)
                yell(_clientName, ((YellToken)token)._msg);
            // Process other kinds of token
            token = reqTokenizer.getToken(_clientIn.readLine());
        }
        _clientSocket.close();
        unregister(_clientName);
    }
    catch (IOException e) {
        unregister(_clientName);
    }
}
An Experiment

- You can, of course, write a client in Java, C, Python, but we're going to use telnet (a client I didn't write)
- In order to participate you need to be able to run telnet
  - **Mac/Linux** you can use `telnet` from the command line (terminal)
  - **Windoze** you can install PuTTY
    https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
- I will open a port on my laptop
- Then I will let you know what my wireless IP address is
- You then run (or do equivalent on PuTTY interface — NB. not ssh!)

  `telnet <IP address> <port>`

- You should see the welcome message from my server
- JOIN <name>
- Etc.
Following the Protocol

- Telnet works because it supports bidirectional interactive text-oriented communication at various levels of sophistication (via negotiation).
- That is, in effect, what the IRC server is doing, but it does send TELLs and YELLs out of sync with the user typing.
- As long as the client and server comply with a common application layer protocol, they can talk.
- For this reason, protocol standards are developed.
- Using the Socket API in our favourite language, we can write a client or a server that operates as that party in a standard protocol.
A Simple Python HTTP Client

```python
#!/usr/bin/python
import sys, socket
try:
    from urllib.parse import urlparse
except ImportError:
    from urlparse import urlparse
if len(sys.argv) != 2:
    print "usage:pcurl.py<url>"
sys.exit()
url = urlparse(sys.argv[1])
if url.scheme != "http":
    print "URL must refer to an HTTP server"
sys.exit()
# Set up a TCP/IP socket
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((url.hostname,80))
s.send("GET" + url.path + "HTTP/1.0\n\n")
while True:
    resp = s.recv(1024)
    if resp == "": break
    print resp,
s.close()
```
A Simple Python HTTP Client (that works with modern servers)

```
#!/usr/bin/python
import sys, httplib
try:
    from urllib.parse import urlparse
except ImportError:
    from urlparse import urlparse

# Get URL from the commandline arguments
if len(sys.argv) != 2:
    print "usage: pcurl2.py <url>"
    sys.exit()
url = urlparse(sys.argv[1])
if url.scheme != "http":
    print "URL must refer to an HTTP server"
    sys.exit()
c = httplib.HTTPConnection(url.hostname)
# Protocol
c.request("GET", url.path)
resp = c.getresponse()
print resp.status
print resp.reason
print resp.read()
```
UDP Sockets

**Client**

- Create a **Datagram socket** to connect to remote host and port
- *Create an empty datagram packet to store outgoing message*
- Write to socket
- Blocking read on socket — wait for packet to arrive

**Server**

- Create a **Datagram Socket** on a specific **port**
- *Create an empty datagram packet to store incoming message*
- Blocking read on socket — wait for packet to arrive
- *Reuse datagram packet, filling it with outgoing message*
- Write to socket
UDP Sockets in Java: Client

```java
// Import appropriate things
public class ShoutClientUDP {
    public static void main(String[] args) {
        // Check args
        try {
            BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
            String msg = stdin.readLine();
            byte[] data = msg.getBytes();
            InetAddress ad = InetAddress.getByName(args[0]);
            int prt = Integer.parseInt(args[1]);
            DatagramPacket p = new DatagramPacket(data, data.length, ad, prt);
            DatagramSocket soc = new DatagramSocket();
            soc.send(p);
            soc.receive(p);
            System.out.println(new String(p.getData()));
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```
// Import appropriate things
public class ShoutServerUDP {
    public static void main( String[] args ) {
        // Check args
        try {
            DatagramSocket soc = new DatagramSocket( Integer.parseInt( args[0] ) );
            while (true) {
                DatagramPacket p = new DatagramPacket( new byte[1024], 1024 );
                soc.receive( p );
                String msg = new String ( p.getData() );
                byte[] data = msg.toUpperCase().getBytes();
                p.setData( data );
                p.setLength( data.length );
                soc.send( p );
            }
        } catch( IOException e ) {
            System.err.println( e.getMessage() );
        }
    }
}
Comments on Shout Server UDP

- A new thread is not created for each client
- No need to handle multiple connections
- Could have a single thread just processing datagrams
- Messages back are also just datagrams
- Given that UDP is connectionless, we have less of a coupling between client and server
- Of course, we’re limited by datagram size
- Can we get the benefits of loose coupling between client and server, but without UDP constraints?
Summary

- Going down to the **socket** level (i.e. the **transport layer**) is one means to achieve **language independence**
- It means that we can write **clients** to interact with **servers** supporting known protocols
- We may employ **simple clients** to interact with **servers we provide**
- We do **loose** a lot of the (mostly) very useful **abstractions**
- We don’t miss blurring the boundary between **local** and **remote**
- **Sockets** is **essential** for many tasks, and **beneficial** for many
- Can we **recover** some of the useful **abstractions**, but retain **language independence**?
- Can we also have **loose coupling**?

Tim Norman  (ECS)
Summary

- Going down to the socket level (i.e. the transport layer) is one means to achieve language independence.
- It means that we can write clients to interact with servers supporting known protocols.
- We may employ simple clients to interact with servers we provide.
- We do loose a lot of the (mostly) very useful abstractions.
- We don’t miss blurring the boundary between local and remote.
- Sockets is essential for many tasks, and beneficial for many.
- Can we recover some of the useful abstractions, but retain language independence?
- Can we also have loose coupling?
- Of course these are rhetorical questions!
- We explore Message Queue models next.