Communication - Introduction and Sockets

Objectives

This lecture introduces the use of sockets, as modeled in Java. We introduce the notion of a network connection, and give an example of a server application capable of handling simultaneous connections from several clients.

Introduction

We have seen in the previous lecture that, sometimes, distributing our systems over several hosts is beneficial. Moreover, there are other scenarios where we need to contact a remote host for some service the remote host provides (e.g., a web site).

In this lecture, we will introduce the programming interface which facilitates communication over the network, namely, sockets.

What is communication

We are going to talk about communication, and how communication actually works. But, what is really communication? In general, communication between two parties involves exchanging information. In the context of computers, network communication is about exchanging bits between two processes, residing either on the same computer or on different computers connected by a network. In the previous lecture, we've seen how to make two objects on different Java RTE's communicate. Here we look on a more general case of enabling two processes to communicate.

Communication is about sending bits between processes. What do these bits represent? And do we really want to work with bits? The first question will be answered in the next section. As for the second, computer communication, as everything else in computers, works using bytes.

Encoding

We established that computer communication is about exchanging bytes. But what do these bytes represent? As everything in a process's memory is represented by bytes, we can potentially send everything between two processes. For example, we might (but should never) do something like this: `send(&object, sizeof(object), destination)`, that is, send `sizeof(object)` bytes from the memory used to store the state of `object` to `destination`. But doing so will result in a non-portable code as different RTE's may use different type representations.

Each application should devise its own way of exchanging information. The simple (and the most widely used) solution is to encode data into textual representation (like strings), and send it over the network. Why textual representation? Because it is debug-able! Programmers can see what they are sending and what was received (please remember and use that in your code).

Now at least two other questions arise:
The first – Are strings represented uniformly on all architectures, by all compilers? The answer, as you have come to expect, is no!
The second – How can we send binary data (for example an executable file) using strings? The answer for that is to use string encoding of binary data (google for Base64 encoding).

Byte order

There are two ways to interpret multi-byte values. For example, take the following binary representation of an unsigned int, `1`: `00 00 00 01`. Each byte is represented by two digits, in hexadecimal. On big endian machines, this int is stored in the following way: `01 00 00 00`, such that the most significant byte is stored at the higher address. On little endian machines, this same value will look like this in memory: `00 00 00 01`, as the most significant byte appears at the lowest address. In order to reduce complexity, it has been decided that all information sent through the network is assumed to be in network byte order (which happens to be big endian, by convention)

For more info, see wikipedia .

UNICODE and UTF-8,16,32

To facilitate the correct exchange of strings across multiple architectures, the UNICODE standard was created. UNICODE presents several encoding schemes for strings, which we can use to interchange information between processes in a portable way (independent of operating systems, RTEs, languages, compilers and GOD). Amongst the most used encoding schemes are UTF-8, UTF-16 and UTF-32. The number after UTF specifies the width (how many bytes) of each character in the encoding scheme. For example, in UTF-32, each character takes exactly 4 bytes (which leads to the possible representation of 2^32 different characters). In contrast, UTF-8 declares that characters may be represented by a single byte. Well, this seems kind of limiting, using only 2^8 different characters. But there is a gotcha: UTF-8 allows several consecutive bytes to represent a single character. That it, a character in UTF-8 may be represented by one, two, three or even four bytes.

Terminology

Before going into finer details regarding communication, we need to define the following:

Server
A server is a process that is accessible over the network. Note the confusing term as in many cases a computer that runs one or more servers is also called a server, especially if it is a big one :)

Client
A client is a process that initiates a connection to a server.

Host
A host is any computer with a network presence – that is, connected to a network and can communicate with other computers connected to the same network.

IP Address
Each host is uniquely identified by an IP Address, which is a 32 bit number, usually written in dot notation. An example of an IP Address is `132.72.50.21`. We will talk more about IP Addresses in the next lectures.

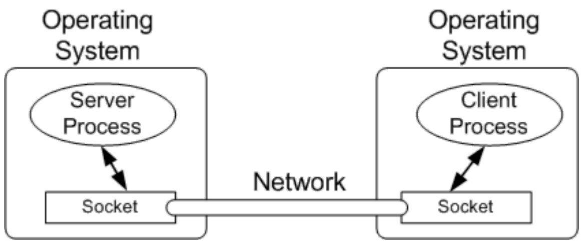
Port
As each host may contain several servers, running side by side, we need some way to distinguish between these servers. For example, consider a single host running both a web server and an ssh server. When a client wants to contact one of these servers, the client will first need the IP of the host running these services. However, the client will also need to indicate the the host's operating system which service is required. To distinguish between servers on the same host, we use ports. A port is just a number, between 0 and 65535. Each server is associated with a port number. When a client sends a message to the host, the client will specify to which server the message is destined, by specifying the relevant port number, and the operating system will transfer the message to the correct server.

Socket
: The programming interface a process uses with the RTE related to communication. You can think of it as a connection's endpoint, which can be used by a process for sending or receiving information. Note - different connection types have different socket types!

Client Server Architecture

Client-Server architecture is the most common way in which two processes communicate over a network. Usually, the clients wish to contact a server, which resides on a different computer, and ask for some service. We think of the server as always-on, meaning the server should be there and listening for requests prior to the time a client initiates the communication. The server will also remain in place after the client is done.

The most notable case today is the world wide web; the client – a web browser – contacts a server – a web server – running on a remote host, asks for a specific web page and presents the web page to the user. The service here is the information contained inside the web page.



Network Communication Models

We can model communications between parties by using the following criteria:

- Is the communication bi-directional? (phone call vs. a T.V. broadcast)
- Is the communication point-to-point? (two parties talking or a radio broadcast)
- Is the communication reliable? (everything sent reaches its destination or not?)
- Is the communication session-oriented? (phone call vs. a snail mail)

We will discuss two communication methods supported by modern RTEs, namely TCP and UDP.

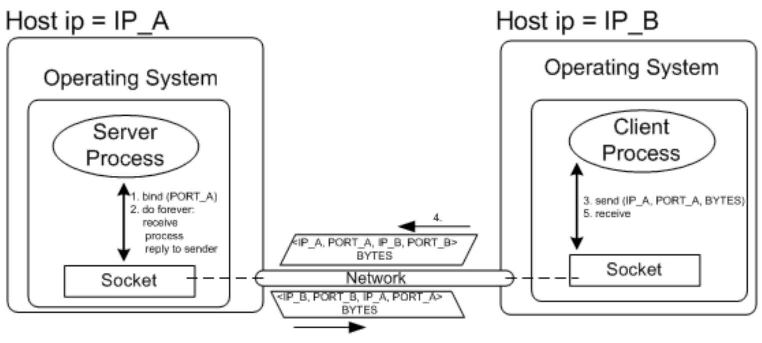
UDP

UDP is an unreliable, one directional, datagram (no session) oriented communication protocol. It can be used in a point to point scenario but not necessarily.

In UDP, two parties (usually a client and a server) can communicate by sending messages to each other. UDP messages are called datagrams, and are independent of each other. Datagrams may be lost during transmission over the network, and the only promise we receive is that if a message arrived at its destination, the message is correct (e.g., was not corrupted along the way).

To send a UDP message from a A (the client) to B (the server), the following must hold:

1. B should ask its RTE to bind a port – that is, associate a number on the host the RTE is running such that messages arriving to this port will be delivered to B.
2. A must know on which host B resides (e.g., the IP).
3. A must know on which port B is listening for incoming messages.
4. A must know what is the format of (or how B will interpret) a message.



A UDP Line Printer Server

Following is an example of a UDP line printer, which accepts UTF-8 encoded string from other hosts, and prints them to its standard output.

```
50 lines ...
import java.io.IOException;
import java.net.DatagramPacket;
```

```java
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.nio.charset.StandardCharsets;

public class UdpServer implements Runnable {

    private DatagramSocket sock;
    private int port;

    private UdpServer(int port)  {
        this.port = port;
    }

    @Override
    public void run() {
        try {
            sock = new DatagramSocket(port);

            byte[] buf = new byte[(1 << 16)];
            byte[] ansBuf = "done".getBytes(StandardCharsets.UTF_8);

            while (true) {
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                sock.receive(packet);

                // print the line
                String data = new String(
                        packet.getData(),           //the bytes
                        0,                          //offset
                        packet.getLength(),         //length
                        StandardCharsets.UTF_8);    //charset
                System.out.println("len: " + packet.getLength() + ":" + data);

                // send an answer: get the address of the sender from the received packet
                InetAddress address = packet.getAddress();
                int clientPort = packet.getPort();
                packet = new DatagramPacket(ansBuf, ansBuf.length,
                        address, clientPort);
                sock.send(packet);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new UdpServer(7777).run();
    }
}
```

Please note the following:

- The connection is UDP so we use a UDP socket named `DatagramSocket`
- Binding – The server first creates a `DatagramSocket`, and binds this socket to the requested port. This server can now receive incoming messages destined for this port. Practically, what makes a process a server (and not a client) is exactly this action of binding the socket.
- Do forever semantic - The server follows this loop:
    - Receive a message – this is done in (`_socket.receive(packet)`) which also saves the message into a `packet`. Note: the call to `_socket.receive(packet)` will **block** until an entire packet has been received!
    - Decode the message – Convert the message to a string `encoder.fromBytes(packet.getData(), packet.getLength())`, assuming the bytes received in the message were a UTF-8 encoded string.
    - Give a service according to the message – Here the service is simply printing the string to `System.out`. In general, this is where the actual code of the server is invoked (to do what is special about this server).
    - Send a reply – build a new packet containing an encoding of the string "done", and send the encoded string using the `_socket` to the client who requested the service. Note: the call to `_socket.send(packet)` will **block** until the entire packet has been sent.

Also note the following technical details:

- `buf` – is just a byte buffer of size 1<<16 (1 << 16 is 2^16 which equals 64KB).
- `Packet` – is just a container for a collection of bytes + their size + an address.
- `InetAddress` – is just a container for an IP address.

A UDP Line Printer Client

As we discussed, the essence of using communication is to be able to send messages between different RTE's, possibly different types of RTE's. We shall see two implementations of the line printer client, the first is in Java and the second is in C++, using Boost.

Line Printer Client in Java

The code of the UDP client in Java is quite similar to that of the UDP server in Java. We use the same classes of `DatagramSocket` and `DatagramPacket`. Note that the order of operations is inverse in the client: we first call send (the client takes the initiative), then we call receive (to wait for an answer).

37 lines ...

```java
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.nio.charset.StandardCharsets;

public class UdpClient {

    public static void send(String host, int port, String msg) {
        byte[] buf = msg.getBytes(StandardCharsets.UTF_8);
        byte[] ansBuf = new byte[(1 << 16)];
        try {
            InetAddress address = InetAddress.getByName(host);
            DatagramSocket _socket = new DatagramSocket();
            DatagramPacket packet = new DatagramPacket(buf,
                    buf.length, address, port);

            //send the message:
            _socket.send(packet);

            //get the reply:
            packet = new DatagramPacket(ansBuf, ansBuf.length);
            _socket.receive(packet);
            System.out.println(new String(packet.getData(), 0, packet.getLength(), StandardCharsets.UTF_8));

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        if (args.length != 3) {
            System.out.println("Usage: java UdpClient host port message");
            return;
        }
        send(args[0], Integer.parseInt(args[1]), args[2]);
    }
}
```

Note the following:

- The client must know the address and port of the server in advance (here we get this information via the command line arguments).
- The client initializes a `Datagram` socket, but does not bind this socket, which means that an arbitrary port number will be assigned to this socket by the operating system when the socket is used to send a packet.
- The client builds a new packet with the line to send, and fills it with the UTF-8 encoded message.
- After sending the datagram, the client waits for an answer (`socket.receive()`). Note that `receive` will return any incoming packet. If you wish to listen for packets from a specific host, you should use `connect()` beforehand.

Line Printer Client in C++

Following is an implementation of a line printer client in C++. Recall using communication means asking services from the RTE. Also recall that the OS RTE API is not object oriented but functional. It is also designed to give its caller full access to its functionality which means in turn a lot of technical code.

To avoid this overhead we will use a package written on top of the OS API. It is named: Poco. Another package we will use is boost. This time it is to avoid the need to work directly with C++ arrays. The boost::scoped_array class is a wrapper around a C++ array allocated on the heap (obtained through new). It ensures that the array will eventually be freed when the scoped_array variable leaves its scope.

36 lines ...

```cpp
#include <iostream>
```

```
#include <Poco/Net/SocketAddress.h>
#include <Poco/Net/DatagramSocket.h>
#include <boost/scoped_array.hpp>

int main(int argc, char **argv)
{
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] <<
            "server port" << std::endl;
        return 1;
    }

    Poco::Net::DatagramSocket sock;
    Poco::Net::SocketAddress server(argv[1], argv[2]);

    boost::scoped_array<char> buf(new char[256]);
    std::string line;
    while (std::cin >> line) {
        // send line to server.
        int len = line.length();
        if (sock.sendTo(line.c_str(), len, server) <= 0) {
            std::cerr << "cannot send line. sorry" << std::endl;
            continue;
        }
        // receive answer from server:
        if ((len = sock.receiveFrom(buf.get(), 256, server)) <= 0) {
            std::cerr << "problem with receive. sorry" << std::endl;
            continue;
        }

        std::string ans(buf.get(), len);
        std::cout << "got " << len << " bytes" << std::endl;
        std::cout << ans << std::endl;
    }
    sock.close();
}
```

TCP

TCP is a reliable, session oriented, bi-directional communication protocol. TCP supports only point to point communication. TCP communication is defined by a **connection** between a client and a server.

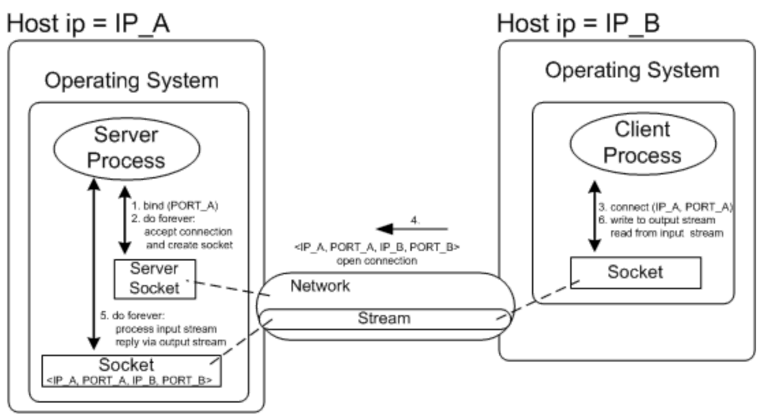
In TCP, two parties (a client and a server) can communicate by using a bi-directional data stream between them. TCP ensures reliable and correct transmission of data across the network. Nothing gets lost. Note we say two parties communicate using a data stream rather than using messages. You should keep that point in mind.

Initiating a TCP connection requires the following (programming-wise):

1. The server opens a new server socket, binds the server socket to a port and waits for incoming connections.
2. The client opens a new socket and and connects this new socket to the server. As with UDP the client must first know the server's address and the port on which the server is bound. The client either chooses its own port or lets the operating system assign a port number for him.
3. The client's operating system will send a request for the server's operating system to initiate a new TCP connection.
4. The new TCP connection (if successfully initiated) is uniquely identified by the following 4-tuple:
   `<server address, server port, client address, client port>.`
5. The server gets a new, regular socket. To this end the client and the server hold each a regular TCP socket. Each socket contains an input and output stream through which the server and client may send/receive data to/from each other.

API-wise, when a TCP server `accept()`s a new connection to its server socket, a new, regular, TCP socket is created. The server can then use this socket to communicate with the client. That is, a server socket is basically a socket factory, producing regular TCP sockets.

Note: in contrast to popular belief, the new socket generated by the server socket does not use a new port number. It is bound to the same port as the server socket. The operating system can distinguish between different TCP streams by checking the 4-tuple we discussed above.



A TCP Line Printer

We will implement a very crude line printer, where a client repeatedly sends lines to the server, which prints these lines. The client does not try to listen for replies from the server.

Note that the server can service several clients concurrently, but in a very inefficient manner.

A TCP Line Printer client in Java

35 lines ...

```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.Socket;

public class TcpClient {

    private static void run(String serverName, int port) {
        //try with resources: automatically close the defined resources when complete or on failure
        try(Socket socket = new Socket(serverName, port);
                BufferedReader userIn = new BufferedReader(new InputStreamReader(System.in));
                BufferedWriter out    = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
                // the next line is not used since we do not listen to the server's replies.
                BufferedReader in     = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

            String line;
            while ((line = userIn.readLine()) != null) {
                out.write(line);
                out.newLine(); // make sure to add the end of line as br.readLine strips it
                out.flush();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("Usage: java TcpClient host port");
            return;
        }
        run(args[0], Integer.parseInt(args[1]));
    }
}
```

Note the following:

- The TCP client differs from the UDP client mainly in the way data is communicated. Where in the UDP client we used a single message at a time (datagram, `packet`s), in the TCP case we use streams. Each socket has two streams associated with it: an input stream, for incoming data, and an output stream, for outgoing data.

- We wrap the socket output stream using a `OutputStreamWriter`, which is set to use UTF-8 encoding. This tells the `OutputStreamWriter` to first encode every string written to the `OutputStreamWriter` using UTF-8, and send the resulting byte array to the `OutputStream` given in the constructor (in our case, `socket.getOutputStream()`). This is why we do not see explicit calls to encode and decode from/to byte arrays to strings. This is handled by the stream.
- As TCP is connection oriented, we use the same stream throughout our communication with the server. Everything sent through the stream will arrive correctly at the server side, in the correct order.
- We use `out.flush()` after feeding each line to the `OutputStreamWriter` to force sending the string to the server. Otherwise, `OutputStreamWriter` is allowed to buffer data, and send it in bigger chunks for efficiency reasons.

Similarly to the UDP server, whenever we `send` or `receive` from a stream, the call blocks until the desired data has been read or received.

A TCP Line Printer Server in Java

The following code shows a Java version of a TCP server. The server runs a next thread each time a client connects to it. In the thread, the server reads incoming data from the client using an `InputStreamReader`.

57 lines ...

```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

public class TcpServer implements Runnable {

    private int port;

    public TcpServer(int port) {
        this.port = port;
    }

    public void run() {
        try (ServerSocket socket = new ServerSocket(port)) {
            while (true) {
                // accept() blocks until a client connects to us
                // It returns a socket connected to the client.
                final Socket client = socket.accept();
                new Thread(() -> {
                    try (BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
                            BufferedWriter out = new BufferedWriter(new OutputStreamWriter(client.getOutputStream()))) {

                        String line;
                        while ((line = in.readLine()) != null) {
                            System.out.println(line);
                            // Our client is not listening to a response so there is no point in sending one.
                            // If you do need to send a response, this is how you do it:
                            // out.write(line);
                            // out.newLine();
                            // out.flush();
                        }

                    } catch (IOException ex) {
                        ex.printStackTrace();
                    } finally {
                        try {
                            client.close();
                        } catch (IOException ex) {
                            ex.printStackTrace();
                        }
                    }
                }).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new TcpServer(7777).run();
    }
}
```

Note the following:

- The server uses a new type of socket called a `ServerSocket`. This is only used for TCP servers.
- The server binds the socket with a port. This makes the server process visible to the outside world.
- The `accept` method of the `ServerSocket` is a blocking call. It returns each time a new connection is established.
- The `accept` method returns a new, regular, TCP socket (as regular as the client's socket).
- The server then starts a new Thread for each new connection, which will take care of this connection. The server then continues to wait for new connections.
- Each thread handling a connection is simply reading form the input stream until the connection is closed.
- The thread first decode the bytes arriving into Java character assuming UTF-8. It associates a buffered reader with the character to yield a string that ends every time a line ends.

Blocking Sockets
Note that reads and writes to a socket, using regular input and output streams, is **blocking**. That is, each time we (the buffered reader) call `read` on the input stream (in the server) or we call `write` on the output stream (in the client), the call is suspended until the required set of character has been sent or received. As we shall see in the next lecture, this is one of the main obstacles we need to overcome in order to create scalable servers, which can handle thousands of clients simultaneously.

A TCP Line Printer client in C++

We now see an example of a TCP client written in C++. This client uses the POCO library abstraction over the OS sockets. POCO deliberately attempted to make its C++ objects similar to those of Java. Still, because this is C++, there are many differences.

58 lines ...

```cpp
#include <iostream>
#include <ostream>
#include <istream>
#include <sstream>
#include <string>
#include "Poco/Net/SocketAddress.h"
#include "Poco/Net/StreamSocket.h"
#include "Poco/Net/SocketStream.h"

// This is a useful template function to convert a string into any
// type T for which a stream reader operation is defined.
// This uses the TSL string streams object (istringstream).
// For numbers, the base of the encoding is passed as an optional argument.
// std::dec is the default (decimal encoding). You may use std::hex for hexadecimal.
// The function returns true if the conversion is successful.
template <typename T>
bool from_string(T& t,
        const std::string& s,
        std::ios_base& (*f)(std::ios_base&) = std::dec)
{
    std::istringstream iss(s);
    return !(iss >> f >> t).fail();
}

void usage(char **argv)
{
    std::cerr << "Usage: " << argv[0] << " host port" << std::endl;
}

int main(int argc, char **argv)
{
    if (argc != 3){
        usage(argv);
        return 1;
    }

    std::string host(argv[1]);
     unsigned short port;
    if (! from_string(port, argv[2])){
        usage(argv);
        return 1;
    }

    // SocketAddress represents the address+port of the line server
    Poco::Net::SocketAddress sa(host, port);
    // Create a TCP socket, and connect it to the server.
    Poco::Net::StreamSocket sock(sa);
    // Create a stream, to connects to/from the server.
    // This stream is bi-directional
    Poco::Net::SocketStream sstream(sock);

    std::string line;
    while (std::cin >> line) {
        // note the use of std::flush, which forces the stream to send
        // the bytes to the other side immediately.
        sstream << line << std::endl << std::flush;
    }
    sock.shutdown();
}
```

TCP Server Efficiency

The TCP server we presented above is very VERY inefficient. More specifically, it is not scalable. In other words, as the number of clients rises, the complexity of the server arises in a linear fashion; as each client requires a special, dedicated, thread to handle communication with the client, the server will not be able to handle more than a few hundred concurrent clients (the CPU will be hogged down by several hundred threads, competing for CPU time).

In a next lecture we will see how to overcome this limitation by following the Reactor design pattern. Informally, we will use just one thread to handle all of the connections by waiting for input from all of them concurrently.

Scalability

If you are interested in reading some more about scalability problems and solutions, we recommend the following link: the c10k problem .