

# CS138 Programming Assignment 1: Comm

---

<i>Assignment Out:</i>	Jan. 24, 2008
<i>Helpsession:</i>	TBA (Location TBD, 7pm)
<i>Assignment Due:</i>	Feb. 7, 2006 (11:59 pm)

---

## 1 Introduction

Welcome to CS138!

The first step in the project is the implementation of a basic communications infrastructure. Comm is a simple protocol for message passing that will eventually be used by your database servers to exchange information. It also serves as an introduction to some concepts in Java to which you may not have been previously exposed, including threading, networking, and serialization.

## 2 The Assignment

The assignment is to write a multithreaded, packet-based communications package for use in the rest of the project. It uses a specific packet data structure, and has a specific interface, but the implementation is up to you.

You must extend the abstract class defined in `Comm.java`. We reproduce the method signatures for your convenience and reference:

```
public abstract class Comm {  
  
    public Comm(int port)  
    public abstract void startThreads();  
    public abstract void stopThreads();  
    public abstract CommPacket receive();  
    public abstract boolean send(CommPacket pack, boolean sync);  
    public String getHost();  
    public abstract int getPort();  
  
}
```

The `Comm` constructor accepts one argument – a port to listen on. It should save this value for

future reference.

The `startThreads` method should start the receiving and (async-)sending threads associated with the communications package. (This isn't done in the constructor because some one-shot "clients" may not actually want to receive anything and may just want to do one synchronous send and quit.)

The `stopThreads` method should terminate the receiving and sending threads. This is a bit tricky, but take a look at the `Thread.interrupt()`, `Thread.join()`, and `ServerSocket.close()` methods in the Java documentation. Make sure you properly close any open sockets when shutting down, or you will often find yourself fighting with "address already in use" errors when binding to ports.

The `send` method sends a packet. It has two operating modes – synchronous and asynchronous. Asynchronous mode should simply place the packet in a local queue and it will be sent when the (async-)sending thread gets around to it. Synchronous mode should send the packet and return when the packet has been transmitted. Packets sent synchronously should be sent immediately and not placed on the queue. All the information about destination is encapsulated in the packet itself; the `send` method must examine the packet to determine where to open a connection to. `send` should return false if a failure was detected in the process of a send.

The `receive` method blocks until a new incoming packet is available.

The `getHost` and `getPort` methods return the hostname of the current machine and the port that your receive thread is listening on. These are already implemented.

This brings us to how your Comm package should be used. You should design your processor (in other words, server or client) so that it instantiates the implementation of `Comm` you write. Then, you can simply use the `send` and `receive` methods once you've started the threads to do your network I/O.

The javadocs will be critical for success on this assignment, as well as the entire course - you can find them at <http://java.sun.com/j2se/1.5.0/docs/api/index.html>.

### 3 Packets

You should use packets to communicate between hosts throughout the assignment. The TAs are saving you the trouble of figuring out what your packet datatype should look like, because they want to be able to use their clients with your communications package (for testing purposes), and this is the easiest way to facilitate such interaction. You are provided with a `CommPacket` class which will specify the packets you must use. You can find this class among the TA-provided support code (commented), and the method signatures below:

```
public final class CommPacket implements Serializable
{
```

```

    public CommPacket(int ttl, String mesg, String toServer, int toPort,String
fromServer, int fromPort, Serializable data,int type)

    public void setTTL(int ttl)
    public int getTTL()

    public void setMessage(String mesg)
    public String getMessage()

    public void setToServer(String toServer)
    public String getToServer()

    public void setToPort(int toPort)
    public int getToPort()

    public void setFromServer(String fromServer)
    public String getFromServer()

    public void setFromPort(int fromPort)
    public int getFromPort()

    public void setData(Serializable data)
    public Serializable getData()

    public void setType(int type)
    public int getType()
}

```

The source and destination ports and addresses are self-explanatory. The TTL field is a Time-To-Live field - every time your comm package receives a packet, it should decrement this field by one. If the resulting TTL is positive, then it should enqueue the packet, else it should discard the packet. It implements Serializable, meaning that Java will send it over the network. More on this later.

## 4 Testing

There is substantial support code to test your Comm implementation. The ring server, `CommTest`, will take an incoming packet, extract the message (which should be the data field of the packet), and append the following information: local server hostname, port, and a timestamp. It then prints out the **new** fulltext of the message to the console, and forwards the message to the next

machine in the ring (provided on the command line). The TTL field should be managed by your Comm package, which will cause the packet to die after a number of hops.

As for the actual ring of servers, there is a script provided named `startup.sh` that you should have copied over, along with a sample list of servers in `list`. You can start multiple servers on the same machine with different ports, which is easier to test, and advisable because you aren't using other machines' resources.

`startup.sh` requires that you give it a list of servers. We have a default list that starts three servers on the local machine, but you can change this if you want. It needs to be run from your project root (the directory above 'comm').

Example, assuming your project root is `/u/yourusername/course/cs138`:

```
cd /u/yourusername/course/cs138
comm/startup.sh comm/list
```

`cleanup.sh` simply looks at your serverlist and kills all java processes on that machine that you own. `list` is a sample serverlist – modify this as you like it. The format is simply a single entry per line, where an entry is `machinename:port`.

## 4.1 Test Client

There is also a test client, `CommClient`, that can be used to inject a packet into the ring. Its invocation is simple – from another shell in the project directory, run:

```
java comm.CommClient portIn serverOut portOut
```

Use for `portIn` any open port on the local machine, and `serverOut` and `portOut` are a server/port pair from your list of servers.

## 5 Error Handling

A large part of this assignment is going to be in error handling - what if the local port is already in use, or the destination is illegal, or a packet received is illegal, etc? You should make use of Java's built-in capabilities for exceptions - your program should not crash for any reason. Think about the errors you have to handle - there are many. Printing out an error message to the console is an acceptable way of dealing with not sending a packet, but you should exit if the error cannot be recovered from. You should be able to handle someone telnetting to the port on which you are running a server and typing in garbage **without** having to quit to recover.

For a few exceptional conditions you should catch and specifically handle, consider the following:

- Invalid destination hostnames/ports
- Invalid data being read or written from the network streams
- Local port in use

## 6 Threading

The comm package creates two separate threads for dedicated I/O (one for receiving, one for sending). The applications that use your comm package (in this case, the ring servers and client) will have at least one other thread, the “main” thread. This (and any other threads your application has) will handle sending and receiving messages to/from your comm package.

Java’s built-in mechanisms for threading allow us to do this easily. Inheriting from the `java.lang.Thread` class allows one to write a thread - one fills in the `run()` method with the code for the thread, and then calls the `start()` of the thread to begin execution. The javadocs are the best reference for this one, but for additional information, see <http://java.sun.com/docs/books/tutorial/essential/threads>.

It might be worthwhile to make use of the classes in the `java.util.concurrent` package. It provides simple abstractions over multiple threads, especially in terms of managing tasks. Our advice is to take the time to study it.

A key thing to remember when you are implementing your package is that you will have multiple threads working with the queues you use for your packets. Regular `java.util.Queue`s won’t do it for you. Fortunately, there is a package in the standard library called `java.util.concurrent` which contains several useful utilities – specifically `java.util.concurrent.BlockingQueue`.

## 7 Networking

Java also provides very simple networking facilities. The two classes of interest are `java.net.Socket` and `java.net.ServerSocket`. `Socket` allows you to make outgoing connections, and `ServerSocket` allows you to listen for connections on a port and turn them into `Sockets`. Two examples follow:

```
// Specify the local port and the backlog for connections
ServerSocket ssock = new ServerSocket(5222, 5);

// wait for someone to try to connect, then return the Socket associated
// with the connection - this will block until someone tries to connect
Socket incoming = ssock.accept();
```

```
// you can now use the methods of Socket to read and write, or wrap
// the underlying stream in another stream (see the section on Serialization)
```

Sending is even simpler:

```
// the parameters are the destination address and port
Socket outgoing = new Socket("localhost", 5222);

// you can now use the methods of Socket to read and write, or wrap
// the underlying stream in another stream (see the section on Serialization)
```

The receive and send threads should perform these tasks, respectively, using a queue for messages that are received and a queue for messages that are waiting to be sent asynchronously. Also, you should write your server in such a way that you can implement `send()` synchronously as the `CommInterface` API requires. Data sent in this way should not be put into any sort of queue, but rather sent directly and `send()` should not return until the actual network send returns.

For more information, see <http://java.sun.com/docs/books/tutorial/networking/sockets>.

## 8 Serialization

Java provides a mechanism for nicely marshaling and unmarshaling most arbitrary data between machines, applications or packages. (For more information on this idea, see CS167 or CS168.) This technique is called serialization - Java packages an object into serialized form which can be input into a stream and then deserialized as output from a stream. This saves you time in writing networking code, because you don't have to worry about treating a packet simply as an array of bytes and then manually reconstructing the object. The javadocs for `ObjectInputStream` and `ObjectOutputStream` should prove very helpful, as the following code snippet demonstrates:

```
Socket sock = new Socket("localhost", 5222);
ObjectOutputStream ostream = new ObjectOutputStream(sock.getOutputStream());
ostream.writeObject(new SerializableObject());
```

This will create a network socket, open an object stream, and then write a serializable object (any object that implements `Serializable`) to the stream. You can do this similarly for receiving an object:

```
// from above
ServerSocket ssock = new ServerSocket(5222, 5);
```

```
Socket incoming = ssock.accept();

ObjectInputStream istream = new ObjectInputStream(incoming.getInputStream());
SerializableObject o = (SerializableObject)istream.readObject();
```

Note that you have to check for exceptions as well - this code fragment is not meant to give away everything. Any classes that simply contain base types (such as `CommPackets`) are easily made `Serializable`; classes that involve locks and other machine-specific or process-specific constructs are not so easily packaged and thus are not.

## 9 Code Exchange and Building

To get started, copy the `/course/cs138/asgn/comm` directory to your course directory.

### 9.1 Eclipse

You should use Eclipse. It makes your life much easier. If you don't know how to use it, ask the TAs.

## 10 Handing in

You need to write a simple README, documenting any bugs you have in your code, any extra features you added, and anything else you think the TAs should know about your Comm project.

You should hand in your comm by running `rm -f *.class;`  
`/course/cs138/bin/cs138_handin comm` from the directory containing your source code.

## 11 Extra credit

Here are some suggestions for extra credit.

- smarter threading model - Right now, you just have three threads - one for receiving, one for processing, and one for dealing with asynchronous sends. If the application itself is multithreaded (multiple “processing” threads), spawning a new thread for each packet to be sent might be more efficient. Thread pools (see the CS167 lecture slides) would be an even smarter idea. In this basic application, it won't matter so much, but a multi-threaded model will become much more efficient for your later projects.

- more efficient socket connections - Currently every time a packet is sent by your comm package it opens a new socket and TCP connection, which from a networking perspective places a large overhead on communication. Change your comm so that it can keep N outgoing and M incoming connections open. So when a packet is sent to a destination which already has an open socket simply use the existing connection. When you need to open a new connection, shut down an existing one (preferably the least recently used). Note that a TCP connection must be closed from both sides, so you'll have to inform the other comm package to terminate as well. We believe that this is a worthwhile addition, and is very instructive. **Note:** It might be easier to do some of this using the `java.nio` package - look at the JavaDocs for more information.

Feel free to do something else, but run your idea past a TA beforehand if you want to be sure to get credit for your additional work.