JCSP Networking – Some Tutorial Examples

These are some short tutorial examples to help you get to grips with setting up and using JCSP Networking. The examples are very brief, and only show the basic mechanisms.

Setting up a JCSP Networking Node

To run a JCSP networking node that is connected to a CNS and BNS, first run `TCPIPNodeServer` in the `org.jcsp.net2.tcpip` package. The simplest method to set up a JCSP networking Node which is connected to both a Channel Name Server (CNS) and a Barrier Name Server (BNS) is as follows:

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.cns.*;
import org.jcsp.net2.tcpip.*;

public class BasicSetup
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter IP of CNS: ");
        String cnsIP = stdIn.readLine();
        // Connect directly to CNS.  Listening port not known
        Node.getInstance().init(new TCPIPNodeFactory(cnsIP));
        // Must use CNS to create channels
        NetChannelInput in = CNS.net2one("chanIn");
        NetChannelOutput out = CNS.one2net("chanOut");
    }
}
```

This method is left in as legacy to the original JCSP networking approach. `NodeFactory` is now deprecated as it requires the core `jcsp.net` package relying on both the `jcsp.net.cns` and `jcsp.net.bns` sub-packages. This should be avoided whenever possible, but the method remains so existing programs should still work. The new method to set up a JCSP networking Node is as follows:

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.bns.*;
import org.jcsp.net2.cns.*;
import org.jcsp.net2.tcpip.*;

public class BasicSetup2
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
```

```
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Get IP of NodeServer
        System.out.print("IP of NodeServer: ");
        String nsIP = stdIn.readLine();
        // Create address to NodeServer.  Default port is 7890
        TCPIPNodeAddress nsAddr = new TCPIPNodeAddress(nsIP, 7890);
        // Initialise CNS and BNS
        CNS.initialise(nsAddr);
        BNS.initialise(nsAddr);
    }
}
```

This method is more long winded, but does give the JCSP networking user more flexibility for creating and connecting to individual Nodes. The original JCSP networking package allowed specific port listening, but also required the user to initialise the CNS in much the same manner if this was done. However, to connect to specific channels on a Node without using a CNS required channel labels, which had to be incorporated into the channel message. As channel messages are now small and light, this possibility has been removed.

The `TCPIPNodeAddress` can be created using either a port or an IP address and port. The former uses the most globally accessible IP address of the host machine to determine the actual `NodeAddress`, and is only useful for Node initialisation. The latter can be used to create a connection to a specific Node or to force the Node to be associated with that particular IP address during initialisation. There is a third option, which is a no argument `TCPIPNodeAddress`. This is only useful for Node initialisation, and the underlying architecture will select a randomly available port to listen on in this instance.

It is also now possible to modify some of the parameters for JCSP networking to suit individual needs. The original architecture had communication Link processes that were given maximum priority, which is fine in many cases but can lead to device flooding in some others. By default, the new architecture gives Link processes medium priority but this can be changed. The underlying TCP/IP Link processes also create buffered streams to service the underlying network streams. The stream originally had a fixed size, which was 8192 bytes. This can also be set. Finally, original JCSP had the Nagle algorithm switched off. The new version also has this feature, but the user can also set this. The advanced features can be set as follows:

```
import java.io.*;
import org.jcsp.lang.*;
import org.jcsp.net2.*;
import org.jcsp.net2.tcpip.*;

public class BasicSetup3
{
    public static void main(String[] args) throws Exception
    {
        // Set Link properties
        Link.LINK_PRIORITY = ProcessManager.PRIORITY_MAX;
        TCPIPLink.BUFFER_SIZE = 1500;
```

```
        TCPIPLink.NAGLE = true;
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
    }
}
```

Channel Creation

The creation of channels has also been opened up to allow direct connection between processes if required. It is also now possible to define your own encoding and decoding methods for message passing. The default behaviour is object serialization, but a filter that will simply write bytes directly is also provided. Examples of how to create channels are given in the following example:

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.bns.*;
import org.jcsp.net2.cns.*;
import org.jcsp.net2.tcpip.*;

public class ChannelCreation
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Get IP of NodeServer
        System.out.print("IP of NodeServer: ");
        String nsIP = stdIn.readLine();
        // Create address to NodeServer.  Default port is 7890
        TCPIPNodeAddress nsAddr = new TCPIPNodeAddress(nsIP, 7890);
        // Initialise CNS and BNS
        CNS.initialise(nsAddr);
        BNS.initialise(nsAddr);
        // We can create channels using the CNS
        // The other Node creates the converse ends
        NetChannelInput in = CNS.net2one("channelIn");
        NetChannelOutput out = CNS.one2net("channelOut");
        // We can create channels to a specific Node
        // or with a specific VCN
        System.out.print("Enter IP of remote node: ");
        String remoteIP = stdIn.readLine();
        System.out.print("Enter port of remote node: ");
        int remotePort = Integer.parseInt(stdIn.readLine());
```

```
        TCPIPNodeAddress remoteAddr =
            new TCPIPNodeAddress(remoteIP, remotePort);
        NetChannelInput in2 = NetChannel.numberedNet2One(45);
        NetChannelOutput out2 = NetChannel.one2net(remoteAddr, 49);
    }
}
```

The approach to creating channels using a `NodeAddress` is a bit slow however, as the Link creation process must be performed each time. This is because Link connections are stored by unique `NodeID` structures. As it is possible for more than one machine to have the same IP:port combination in an application if cross boundary communication is used, this a requirement to ensure that existing Links are used. In fact, the original JCSP networking library would not allow such a connection due to the Link creation process occurring each time a new channel was created from a location, and thus inter-domain networking broke down. A more efficient method for creating channels is to use the `NodeID`, which can be obtained from the Link itself:

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.tcpip.*;

public class ChannelCreation2
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Get address of remote Node
        System.out.print("Enter IP of remote node: ");
        String remoteIP = stdIn.readLine();
        System.out.print("Enter port of remote node: ");
        int remotePort = Integer.parseInt(stdIn.readLine());
        TCPIPNodeAddress remoteAddr =
            new TCPIPNodeAddress(remoteIP, remotePort);
        // Create and get link to remote node
        NodeID remoteID =
            LinkFactory.getLink(remoteAddr).getRemoteNodeID();
        NetChannelInput in = NetChannel.numberedNet2One(45);
        NetChannelOutput out = NetChannel.one2net(remoteID, 49);
    }
}
```

This approach uses the `LinkFactory` to create and start a Link to a specified Node. The `LinkFactory` will connect to the remote Node and perform a handshake operation. If it is found that an existing Link is in operation between the two Nodes, then the original Link is returned instead. If a `NodeID` is given to the `LinkFactory`, then a lookup operation is performed first to check if an existing Link is in operation.

It is also possible to create output channel ends from a `NetChannelLocation`. Each channel has a location associated with it. For an input end, this is the actual location of the channel. For an output end, it is the location of the input end that is returned. An example of this is as follows:

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.tcpip.*;

public class ChannelCreation3
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Get address of remote Node
        System.out.print("Enter IP of remote node: ");
        String remoteIP = stdIn.readLine();
        System.out.print("Enter port of remote node: ");
        int remotePort = Integer.parseInt(stdIn.readLine());
        TCPIPNodeAddress remoteAddr =
            new TCPIPNodeAddress(remoteIP, remotePort);
        // Create and get link to remote node
        NodeID remoteID =
            LinkFactory.getLink(remoteAddr).getRemoteNodeID();
        NetChannelOutput out = NetChannel.one2net(remoteID, 49);
        // Create an input end and send location to remote Node
        NetChannelInput in = NetChannel.net2one();
        out.write(in.getLocation());
        // Read in location and create new output channel
        NetChannelLocation loc = (NetChannelLocation)in.read();
        NetChannelOutput out2 = NetChannel.one2net(loc);
    }
}
```

Channels can also be created with specific encoding and decoding filters instead of the default object serialization method. Two basic filters are provided with JCSP – the object serialization one and a raw byte filter. To use a specific filter, simply pass it in as an argument during channel creation:

```
NetChannelInput in =
    NetChannel.net2one(new RawNetworkMessageFilter.FilterRX());
NetChannelOutput out =
    NetChannel.one2net(loc, new RawNetworkMessageFilter.FilterTX());
```

Creating a Custom Channel Encoder / Decoder

The interface for encoding and decoding filters has been exposed to allow JCSP networking users the ability to create their own encoding / decoding mechanisms as necessary. For example, we can create a filter which takes an Integer object and sends it as 4 bytes instead of the entire serialized Java object:

```
import java.io.IOException;
import org.jcsp.net2.*;

public class IntegerNetworkMessageFilter
{
    public static class FilterTx implements NetworkMessageFilter.FilterTx
    {
        public byte[] filterTX(Object obj) throws IOException
        {
            if (!(obj instanceof Integer))
                throw new IOException("Not an Integer");
            int toSend = ((Integer)obj).intValue();
            return new byte[] {(byte)(toSend >>> 24),
                               (byte)(toSend >>> 16),
                               (byte)(toSend >>> 8),
                               (byte)toSend};
        }
    }

    public static class FilterRx implements NetworkMessageFilter.FilterRx
    {
        public Object filterRX(byte[] bytes) throws IOException
        {
            if (bytes.length != 4)
                throw new IOException("Not 4 bytes");
            return new Integer((bytes[0] << 24)
                             + ((bytes[1] & 0xFF) << 16)
                             + ((bytes[2] & 0xFF) << 8)
                             + (bytes[3] & 0xFF));
        }
    }
}
```

An outgoing (encoding) filter implements `NetworkMessageFilter.FilterTx` which defines a method that takes an object and returns a byte array. The underlying Link layer only understands byte array messages. If there is a problem during encoding, the filter should throw an `IOException`. Internally, the write operation of the channel catches this and converts it into a more suitable exception (`JCSPNetworkException`). The decoding filter performs the reverse operation, and implements `NetworkMessageFilter.FilterRx`.

A Quick Note on Performance

As the new implementation serializes Java objects at the channel level (with the `ObjectNetworkMessageFilter`), each output channel is given a buffer to serialize objects into. By default this buffer is 8192 bytes in size, but will grow for larger objects, with the size being reset to the original size afterwards. This growth comes at a cost to performance

however. For larger objects (> 8192 bytes serialized) there will be drops in performance. The `BUFFER_SIZE` attribute has been publically exposed to allow the JCSP user to configure the system to the required buffer size. If large chunks of data are to be sent, it is best to use the `RawNetworkMessageFilter` instead, or to create a custom encoder / decoder. In future releases, this problem may be resolved by exposing the `LinkTx` stream for more efficient serialization purposes. This would also remove the need for a `LinkTx` process in general.

The problem is not evident in the deserialization process as the incoming bytes are used as the reading buffer. In a future Java 1.4+ release the usage of the `java.nio` and `ByteBuffer` objects may reduce some of the memory problems.

Other Channel Operations

Poison

Network channels are now poisonable in much the same manner as standard channels. Unlike standard channels, each end of the channel is given an immunity level. Technically, it is only the input end that requires poisoning, as if the input end were to become poisoned, all output ends will likewise become poisoned no matter the immunity level. However, as networked channels are Any-2-One in nature, it becomes feasible that output ends may have poison called upon them but do not spread the poison to the input end. Effectively the output ends act as a barrier for poison spread if required. The following example illustrates how to create a poisoned channel:

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.tcpip.*;

public class Poison
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Get address of remote Node
        System.out.print("Enter IP of remote node: ");
        String remoteIP = stdIn.readLine();
        System.out.print("Enter port of remote node: ");
        int remotePort = Integer.parseInt(stdIn.readLine());
        TCPIPNodeAddress remoteAddr =
            new TCPIPNodeAddress(remoteIP, remotePort);
        // Create and get link to remote node
        NodeID remoteID =
            LinkFactory.getLink(remoteAddr).getRemoteNodeID();
        // Create an input channel with immunity level
```

```
        NetChannelInput in = NetChannel.net2one(500);
        // Create a numbered (50) input channel with immunity level
        NetChannelInput in2 = NetChannel.numberedNet2Any(50, 500);
        // Create an output channel with immunity level
        NetChannelOutput out = NetChannel.one2net(remoteID, 50, 500);
        // Poison channels as standard
        in.poison(501);
        out.poison(501);
    }
}
```

To guard against poison use try-catch:

```
try
{
    Object obj = in.read();
}
catch (NetworkPoisonException pe)
{
    // Spread poison
}
```

`NetworkPoisonException` extends `PoisonException` and is unchecked. The reason to create a separate exception follows from the original ideas for JCSP poison [1], and allows the two types of poison to be treated separately if needs be.

Extended Rendezvous

The extended read operations have also been incorporated into the new API. A simple example is:

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.tcpip.*;

public class ExtendedRendezvous
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Create input end and start a read operation
        NetChannelInput in = NetChannel.net2one();
        Object read = in.startRead();
        // Perform extended operations
        // Finish read operation
        in.endRead();
    }
}
```

Generally extended rendezvous is used to stretch a channel, and maybe the further reading end will have been poisoned or destroyed. In such an occurrence, the channel input in an extended read can be poisoned and `endRead()` does not need to be called. The poison will be sent back instead of an acknowledgement to the writer.

Channel Destruction

Unlike standard JCSP channels, network channels can be destroyed. For the original networking implementation this was a requirement as input ends were serviced by a process, which meant a thread. The requirement for channel destruction is less so now, but if a channel is not destroyed it does remain accessible to the underlying network architecture. Each channel has an infinitely buffered channel to allow messages to always be sent to the channel end object without blocking the Link process, and if not destroyed, the input end may continue to receive messages and consume memory. In the future, the garbage collector may be able to do this for us, but no experiments have been done to confirm this.

To destroy a channel, simply call the destroy method upon it:

```
in.destroy();
```

This will remove the channel from the `ChannelManager` and set the channel state to `DESTROYED`. If an input end is destroyed, then any pending `SEND` messages are acknowledged with a rejection signal, causing the corresponding output end to throw a `JCSPNetworkException`.

Asynchronous Channel Writing - **DANGEROUS**

The original JCSP implementation provided a `UnacknowledgedChannel` to simulate asynchronous operations. This was used extensively in server type scenarios in the underlying architecture, such as the original Channel Name Server and the `NetConnectionServer`. This was required to avoid a server deadlocking if the connection to the client end failed. This has now been resolved, but currently the CNS still uses this technique for simplicity, although asynchronous connections will be added soon to get round this problem. As the underlying channel object utilises an infinitely buffered standard channel, this operation is inherently dangerous, and will in all likelihood be removed (unless someone can argue for keeping it). To perform an asynchronous write, call the `asyncWrite()` method:

```
out.asyncWrite(obj);
```

Channel Error Handling

Both the input and output channel ends may throw a `JCSPNetworkException` during operations, either from encoding / decoding problems or in the case of output ends when the connection to the input end is lost. At no point should an operation cause deadlock

outside of the application level. This is unlike the original JCSP networking implementation, which could leave a write operation hanging if the Link to the input end went down.

The `JCSPNetworkException` is unchecked, so it does not require explicit try-catch code. This is so the net channel interfaces are exactly as the standard channel interfaces. An example error handling program is given below:

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.bns.*;
import org.jcsp.net2.cns.*;
import org.jcsp.net2.tcpip.*;

public class ChannelErrorHandling
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Get IP of NodeServer
        System.out.print("IP of NodeServer: ");
        String nsIP = stdIn.readLine();
        // Create address to NodeServer.  Default port is 7890
        TCPIPNodeAddress nsAddr = new TCPIPNodeAddress(nsIP, 7890);
        // Initialise CNS and BNS
        CNS.initialise(nsAddr);
        BNS.initialise(nsAddr);
        // Resolve a channel name with the CNS
        NetChannelOutput out = CNS.one2net("channel");
        // Loop
        while (true)
        {
            // Try and write to the channel
            try
            {
                out.write("Hello, world!");
            }
            catch (JCSPNetworkException jne)
            {
                // Presume connection failure.  Try and resolve again
                out.destroy();
                out = CNS.one2net("channel");
            }
        }
    }
}
```

Buffered Channels

Unlike the original JCSP networking architecture, channel buffering is not currently implemented for networked channels. The hope is to add this functionality soon. The

original implementation achieved buffering by utilising the extra process between the Link process and the channel end (the `NetChannelInputProcess`). As this process would block until the input channel end object was ready to read, making the channel between this process and the channel end buffered implemented the channel buffering easily. The same effect can be achieved in the new architecture in the same manner.

```
import java.io.*;
import org.jcsp.lang.*;
import org.jcsp.util.*;
import org.jcsp.net2.*;
import org.jcsp.net2.tcpip.*;

public class BufferedChannel implements CSProcess
{
    private final ChannelInput input;
    private final ChannelOutput output;

    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Create a net channel
        NetChannelInput in = NetChannel.net2one();
        // Create a normal channel with a buffer
        One2OneChannel chan = Channel.one2one(new Buffer(10));
        // Create and start buffered channel
        new ProcessManager(new BufferedChannel(in, chan.out()));
        // Read from normal input end
        chan.in().read();
    }

    private BufferedChannel(ChannelInput input, ChannelOutput output)
    {
        this.input = input;
        this.output = output;
    }

    public void run()
    {
        while (true)
            output.write(input.read());
    }
}
```

Networked Barriers

Network barriers are a new addition to JCSP networking, and operate in much the same manner as a standard JCSP `Barrier`; `NetBarrier` extends `Barrier`. The key difference lies in how the `NetBarrier` is constructed. A two-tier approach has been taken, with a `NetBarrier` having a number of locally enrolled processes, and there being a number of `NetBarrier` ends synchronising together. To enable this, one `NetBarrier` end must declare the `NetBarrier` and act as a server for it. This `NetBarrier` end then has a number of client ends enrolled on it. A client end only synchronises with the server end when all its locally enrolled processes have synchronised with it. Once all enrolled client ends and locally enrolled processes have synchronised with the server end, the `NetBarrier` server signals all client ends to be released, completing the synchronisation operation.

Creating a `NetBarrier` is much the same as creating a networked channel. The following program gives an example of the two methods.

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.bns.*;
import org.jcsp.net2.cns.*;
import org.jcsp.net2.tcpip.*;

public class BarrierCreation
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Get IP of NodeServer
        System.out.print("IP of NodeServer: ");
        String nsIP = stdIn.readLine();
        // Create address to NodeServer.  Default port is 7890
        TCPIPNodeAddress nsAddr = new TCPIPNodeAddress(nsIP, 7890);
        // Initialise CNS and BNS
        CNS.initialise(nsAddr);
        BNS.initialise(nsAddr);
        // Declare a NetBarrier server end with the BNS
        NetBarrier bar1 = BNS.netBarrier("bar1", 5, 10);
        // Resolve a NetBarrier from the BNS and create a client end
        NetBarrier bar2 = BNS.netBarrier("bar2", 10);
        // Create a NetBarrier without the BNS
        NetBarrier bar3 = NetBarrierEnd.netBarrier(5, 10);
        // Create a numbered NetBarrier
        NetBarrier bar4 = NetBarrierEnd.numberedNetBarrier(49, 5, 10);
        // Connect to a NetBarrier server end
        TCPIPNodeAddress remoteAddr =
            new TCPIPNodeAddress("192.168.1.100", 4000);
        NetBarrier bar5 = NetBarrierEnd.netBarrier(remoteAddr, 49, 10);
```

```
    }
}
```

Notice that a `NetBarrier` server end requires two number values passed as creation parameters. The first parameter is the number of locally enrolled processes, and the second is the number of remote enrolled ends expected. This second parameter allows the JCSP user the ability to specify how many synchronising Nodes should be in operation, and when `sync()` is called on the server end, it will still block until the expected number of client ends have enrolled AND synced.

NetBarrier Operation

As mentioned, the `NetBarrier` extends `Barrier`, and can thus be used as a `Barrier` in existing applications with no modification. The basic `Barrier` operations are:

```
bar.sync();
bar.enroll();
bar.resign();
bar.reset(numToEnroll);
```

The first method (`sync()`) causes the calling process to synchronise with the barrier and wait until all locally enrolled processes, and all remote ends, have done likewise. The second method (`enroll()`) increments the count of locally enrolled processes by one, and the third method (`resign()`) decrements the number by one. The `reset(numToEnroll)` method changes the number of enrolled processes to the passed in value.

There are some differences in the underlying architecture during the enrolment and resignation methods. If a client `NetBarrier` end has all locally enrolled processes resigned, then the client end will likewise resign from the `NetBarrier` server end. If a process subsequently enrols on the resigned client end, then the `NetBarrier` client end will re-enrol itself automatically with the server end. The server end can never have less than one process enrolled with it in the current implementation, as this process is used to service the server end. If the enrolled count on a server end does reach zero a `JCSPNetworkException` is raised. This decision is due to one of the aims of the implementation being to use as few processes as possible. If the JCSP networking user wants to create a `NetBarrier` server end which may have the possibility of having all locally enrolled processes removed, then a service process should also be created and enrolled with the `NetBarrier`. This is fairly trivial:

```
while (true)
{
    bar.sync();
}
```

The `NetBarrier` does add two new operations not seen in the `Barrier`:

```
bar.destroy();
NetBarrierLocation loc = (NetBarrierLocation)bar.getLocation();
```

These are inherited from the `Networked` interface which `NetBarrier`, `NetChannelInput` and `NetChannelOutput` all implement (as will `NetConnection`). The former is required for the same reason as channels require destruction. The latter method allows barriers to be created from locations; with server ends returning their own location and client ends returning the location of the server end they are connected to.

Barrier Error Handling

During synchronisation operations, a `NetBarrier` may throw a `JCSPNetworkException`. Generally, this will occur due to connection failure between two Nodes. For a client end of a `NetBarrier` this is considered fatal. For a server end, the JCSP networking user has a choice of killing the application by destroying the barrier or continuing. The underlying architecture will decrement the count of remotely enrolled processes on the server end so that the user does not have to do anything. The following example illustrates the latter possibility:

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.tcpip.*;

public class BarrierErrorHandling
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Create a NetBarrier server end
        NetBarrier bar = NetBarrierEnd.numberedNetBarrier(49, 1, 10);
        // Flag to indicate whether sync was successful
        boolean synced = false;
        // Loop
        while (true)
        {
            // Set synced to false
            synced = false;
            // Loop until synced
            while (!synced)
            {
                try
                {
                    // Try and sync on the barrier
                    bar.sync();
                    // Sync successful.  Set flag to true
                    synced = true;
                }
                catch (JCSPNetworkException jne)
                {
                    // Sync failed.  Print message
```

```
                    System.err.println("Lost connection to client end");
                }
            }
            // Do some work
        }
    }
}
```

Mobility

The mobility models previously described in the JCSP mobile paper [2] have been provided in the `org.jcsp.net2.mobile` package with some refinements. The code loading capability is likely close to final, and it is unlikely that the interface will change. The channel mobility model is not so robust, and is likely to change in the future. The hope is to have all networked channels as mobile, as well as having mobility built directly into the protocol allowing cross platform mobility. The problem is that a suitable channel mobility model has not been chosen yet.

Network process mobility is still limited to stopped processes, but the code loading mechanism is now trivial to use. The following example illustrates:

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.tcpip.*;
import org.jcsp.net2.mobile.*;

public class CodeMobility
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Create address to remote Node
        TCPIPNodeAddress remoteAddr =
            new TCPIPNodeAddress("192.168.1.100", 4000);
        // Create a code loading output channel
        NetChannelOutput out =
            NetChannel.one2net(remoteAddr, 49,
                               new CodeLoadingChannelFilter.FilterTX());
        // Create a code loading input channel
        NetChannelInput in =
            NetChannel.net2one(new CodeLoadingChannelFilter.FilterRX());
        // Send message, required code requested if required
        out.write("Hello, world");
        // Read message, required code requested if required
        Object obj = in.read();
    }
}
```

Channel mobility still requires extra processes acting as message boxes, and currently guarded input is not supported. An example of mobile channel usage is as follows:

```
import java.io.*;
import org.jcsp.lang.*;
import org.jcsp.net2.*;
import org.jcsp.net2.tcpip.*;
import org.jcsp.net2.mobile.*;

public class MobileChannels
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Create a mobile channel input
        MobileChannelInput mobileIn = new MobileChannelInput();
        // Connect an output end to the input end
        MobileChannelOutput mobileOut = new MobileChannelOutput
            ((NetChannelLocation)mobileIn.getLocation());
        // Connect to another Node and send it both ends
        TCPIPNodeAddress remoteAddr =
            new TCPIPNodeAddress("192.168.1.100", 4000);
        NetChannelOutput out = NetChannel.one2net(remoteAddr, 55);
        out.write(mobileIn);
        out.write(mobileOut);
        // We cannot end the process as this would destroy the mobile end.
        // Wait indefinitely
        Channel.one2one().in().read();
    }
}
```

Logging

Those familiar with the original JCSP networking implementation will know that by default logging is switched on, displaying messages on the screen. In the new implementation, logging is switched off. Activating logging is trivial however, as the following example illustrates:

```
import java.io.*;
import org.jcsp.net2.*;
import org.jcsp.net2.bns.*;
import org.jcsp.net2.cns.*;
import org.jcsp.net2.tcpip.*;

public class Logging
{
    public static void main(String[] args) throws Exception
    {
        // First activate logging
        // Normal messages sent to a file, and error messages to the screen
```

```
        Node.getInstance().setLog(new FileOutputStream("logfile.txt"));
        Node.getInstance().setErr(System.err);

        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter port to listen on: ");
        int port = Integer.parseInt(stdIn.readLine());
        // Create a node address to listen on.
        // Use default IP, which is the most accessible
        TCPIPNodeAddress addr = new TCPIPNodeAddress(port);
        // Initialise Node
        Node.getInstance().init(addr);
        // Get IP of NodeServer
        System.out.print("IP of NodeServer: ");
        String nsIP = stdIn.readLine();
        // Create address to NodeServer.  Default port is 7890
        TCPIPNodeAddress nsAddr = new TCPIPNodeAddress(nsIP, 7890);
        // Initialise CNS and BNS
        CNS.initialise(nsAddr);
        BNS.initialise(nsAddr);
    }
}
```

Creating a Custom Link

The majority of the JCSP networking functionality has been implemented in a reusable manner, allowing extensions via simple interfaces. To illustrate, we will create a serial comms Link for JCSP networking. The following has not been tested, but the principles should remain the same.

To create a custom Link and protocol implementation, there are a number of interfaces / abstract classes that require implementation. These are:

- *ProtocolID* – uniquely identifies the protocol
- *NodeAddress* – the addressing mechanism to use
- *Link* – the Link process mechanism between two Nodes
- *LinkServer* – the process used to listen for incoming connections

Serial comms are point-to-point communication, and therefore the `LinkServer` will only ever listen for one incoming connection. This does mean that for such a system, the `LinkManager` is not really required, but this does not matter.

First we will create a `SerialProtocolID` class. All this class is responsible for doing is taking a string representation of a `SerialNodeAddress` and parsing it to recreate the `SerialNodeAddress`. The only value that uniquely defines a com port is the number. We can therefore define a `SerialNodeAddress` string as taking the form *serial**4*. The class definition is:

```
import org.jcsp.net2.*;

public final class SerialProtocolID extends ProtocolID
{
    private static SerialProtocolID instance = new SerialProtocolID();

    public static SerialProtocolID getInstance()
    {
        return instance;
    }

    private SerialProtocolID() {}

    protected NodeAddress parse(String addressString)
    throws IllegalArgumentException
    {
        // Split address
        int index = addressString.indexOf("\\\\");
        String temp = addressString.substring(index + 2);
        int port = Integer.parseInt(temp);
        return new SerialNodeAddress(port);
    }
}
```

The usage of a singleton here is merely a matter of convenience. There is no need to have more than one protocol object within the system. JCSP uses the parse method to parse an address, and a container of protocol types is stored against the protocol name to allow the correct parsing method to be used.

The `SerialNodeAddress` must set two inherited properties (protocol and address), and must provide methods to create a `Link` and `LinkServer` from an address. It also must provide a get method that returns the relevant `ProtocolID`. The following code illustrates:

```
import org.jcsp.net2.*;

public final class SerialNodeAddress extends NodeAddress
{
    private int port;

    public SerialNodeAddress(int portNumber)
    {
        this.port = portNumber;
        this.protocol = "serial";
        this.address = "" + port;
    }

    public final int getPort()
    {
        return this.port;
    }

    void setPort(int portNumber)
    {
        this.port = portNumber;
    }
```

```
    protected Link createLink() throws JCSPNetworkException
    {
        return new SerialLink(this);
    }

    protected LinkServer createLinkServer() throws JCSPNetworkException
    {
        return new SerialLinkServer(this);
    }

    protected ProtocolID getProtocolID()
    {
        return SerialProtocolID.getInstance();
    }
}
```

The `SerialLink` is where the majority of the functionality is required. Link requires a child class to implement three methods: `connect()`, `createResources()`, and `destroyResources()`. The first method is used to perform a handshake operation between two `SerialLink` processes, the second creates any extra resources required for the Link, and the third method destroys these and other resources. To allow serial comms, we will use the `javax.comm` package available from Sun for Linux / Solaris. Windows and Apple users will have to hunt a little harder.

The following code presents the `SerialLink` process:

```
import java.io.*;
import javax.comm.*;
import org.jcsp.net2.*;

public class SerialLink extends Link
{
    public static int BUFFER_SIZE = 8192;
    private SerialPort serialPort;
    private SerialNodeAddress remoteAddress;

    public SerialLink(SerialNodeAddress address)
    throws JCSPNetworkException
    {
        try
        {
            CommPortIdentifier portID = CommPortIdentifier
                .getPortIdentifier("COM" + address.getPort());
            if (portID.isCurrentlyOwned())
                throw new JCSPNetworkException("Comm port already owned");
            this.serialPort =
                (SerialPort)portID.open("JCSPSerialComms", 2000);
            this.serialPort.setOutputBufferSize(4);
            this.serialPort.setSerialPortParams
                (9600, SerialPort.DATABITS_8,
                 SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
           this.serialPort.setFlowControlMode
                (SerialPort.FLOWCONTROL_RTSCTS_IN
                 | SerialPort.FLOWCONTROL_RTSCTS_OUT);
            this.rxStream = new DataInputStream
                (new BufferedInputStream
                    (serialPort.getInputStream(), BUFFER_SIZE));
```

```
            this.txStream = new DataOutputStream
                (new BufferedOutputStream
                    (serialPort.getOutputStream(), BUFFER_SIZE));
            this.remoteAddress = address;
            this.connected = false;
        }
        catch (IOException ioe)
        {
            throw new JCSPNetworkException("Failed to open comm port");
        }
        catch (NoSuchPortException nsp)
        {
            throw new JCSPNetworkException("Failed to create port");
        }
        catch (PortInUseException pue)
        {
            throw new JCSPNetworkException("Comm port already in use");
        }
        catch (UnsupportedCommOperationException uce)
        {
            throw new JCSPNetworkException("Failed to set properties");
        }
    }

    SerialLink(SerialPort serialPort, NodeID nodeID)
    throws JCSPNetworkException
    {
        try
        {
            this.serialPort = serialPort;
            this.rxStream = new DataInputStream
                (new BufferedInputStream
                    (serialPort.getInputStream(), BUFFER_SIZE));
            this.txStream = new DataOutputStream
                (new BufferedOutputStream
                    (serialPort.getOutputStream(), BUFFER_SIZE));
            this.remoteID = nodeID;
            this.remoteAddress =
                (SerialNodeAddress)nodeID.getNodeAddress();
            this.connected = true;
        }
        catch (IOException ioe)
        {
            throw new JCSPNetworkException("Failed to create Link");
        }
    }

    public boolean connect() throws JCSPNetworkException
    {
        if (this.connected)
            return true;
        // The return value
        boolean toReturn = false;
        try
        {
            // Send out NodeID
            this.txStream.writeUTF
                (Node.getInstance().getNodeID().toString());
            this.txStream.flush();
            // Other end responds OK if this connection is new
            String response = this.rxStream.readUTF();
```

```
            if (response.equals("OK"))
                toReturn = true;
            // Read in the other ends NodeID string and parse
            String nodeIDString = this.rxStream.readUTF();
            NodeID otherID = NodeID.parse(nodeIDString);
            // Ensure we have a SerialNodeAddress
            if (otherID.getNodeAddress() instanceof SerialNodeAddress)
            {
                // Set properties
                this.remoteAddress =
                    (SerialNodeAddress)otherID.getNodeAddress();
                this.remoteID = otherID;
                this.connected = true;
                return toReturn;
            }
            throw new JCSPNetworkException("Failed to create Link");
        }
        catch (IOException ioe)
        {
            throw new JCSPNetworkException("Failed to create Link");
        }
    }

    protected boolean createResources() throws JCSPNetworkException
    {
        // Return true
        return true;
    }

    protected void destroyResources()
    {
        try
        {
            synchronized (this)
            {
                if (this.serialPort != null)
                {
                    // Close everything
                    this.txStream.close();
                    this.rxStream.close();
                    this.serialPort.close();
                    this.serialPort = null;
                    // Fire link lost
                    this.lostLink();
                }
            }
        }
        catch (Exception e)
        {
            // Ensure link lost is fired
            this.lostLink();
        }
    }

}
```

The final class we need to create is the `SerialLinkServer`. Unlike standard network comms, serial comms are point to point, so we do not require the `SerialLinkServer` to be constantly listening for an incoming serial connection. In fact, one application must open

the connection while the other accepts the incoming connection and performs the handshake. This can be set by a flag in the `SerialLinkServer`. The class definition is:

```
import java.io.*;
import javax.comm.*;
import org.jcsp.net2.*;

public class SerialLinkServer extends LinkServer
{
    public static boolean IS_SERVER = true;
    private SerialPort serialPort;
    private SerialNodeAddress listeningAddr;

    public SerialLinkServer(SerialNodeAddress address)
    {
        try
        {
            this.listeningAddr = address;
            CommPortIdentifier portID = CommPortIdentifier
                .getPortIdentifier("COM" + address.getPort());
            if (portID.isCurrentlyOwned())
                throw new JCSPNetworkException("Comm port already owned");
            this.serialPort =
                (SerialPort)portID.open("JCSPSerialComms", 2000);
            this.serialPort.setOutputBufferSize(4);
            this.serialPort.setSerialPortParams
                (9600, SerialPort.DATABITS_8,
                 SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
            this.serialPort.setFlowControlMode
                (SerialPort.FLOWCONTROL_RTSCTS_IN
                | SerialPort.FLOWCONTROL_RTSCTS_OUT);
        }
        catch (NoSuchPortException nsp)
        {
            throw new JCSPNetworkException("Failed to create port");
        }
        catch (PortInUseException pue)
        {
            throw new JCSPNetworkException("Comm port already in use");
        }
        catch (UnsupportedCommOperationException uce)
        {
            throw new JCSPNetworkException("Failed to set properties");
        }
    }

    public void run()
    {
        if (!SerialLinkServer.IS_SERVER)
            return;
        try
        {
            DataInputStream inStream =
                new DataInputStream(serialPort.getInputStream());
            // Receive remote NodeID and parse
            String otherID = inStream.readUTF();
            NodeID remoteID = NodeID.parse(otherID);
            // Ensure we have a SerialNodeAddress
            if (remoteID.getNodeAddress() instanceof SerialNodeAddress)
            {
```

```
                DataOutputStream outStream =
                    new DataOutputStream(serialPort.getOutputStream());
                // Check if it is a new link
                if (requestLink(remoteID) == null)
                {
                    // Send OK message
                    outStream.writeUTF("OK");
                    outStream.flush();
                    // Send our NodeID
                    outStream.writeUTF
                        (Node.getInstance().getNodeID().toString());
                    outStream.flush();
                    // Create, register and run Link
                    SerialLink link = new SerialLink(serialPort, remoteID);
                    registerLink(link);
                    link.run();
                }
                else
                {
                    // Connection already exists?!?
                    // Write EXISTS to remote node
                    outStream.writeUTF("EXISTS");
                    outStream.flush();
                    // Send out NodeID.
                    outStream.writeUTF
                        (Node.getInstance().getNodeID().toString());
                    outStream.flush();
                    throw new JCSPNetworkException
                        ("Failed to open comm port");
                }
            }
            else
                throw new JCSPNetworkException("Failed to open comm port");
        }
        catch (IOException ioe)
        {
            throw new JCSPNetworkException("Failed to open comm port");
        }
    }
}
```

All the `SerialLinkServer` requires is a constructor taking a `SerialNodeAddress`, and a run method, which performs the initial handshaking. This implementation is not really reusable, and a better approach would be to create the `SerialLink` processes individually. However, this approach shows the standard method for creating a new communication Link protocol.

Future Additions

There are a number of features that will hopefully be added soon. These include:

- `NetConnections` with asynchronous methods
- Better channel mobility model
- Mobile barriers, although using locations can achieve this
- Buffered networked channels (although these might require an extra process)
- Networked `AltingBarrier`, although this is tricky

- More efficient I/O using the `java.nio` package
- Primitive type network channels
- Network channel arrays
- Automatic garbage collection of net channels and barriers

References

[1] B. H. C. Sputh and A. R. Allen, "JCSP-Poison: Safe Termination of CSP Process Networks," in J. F. Broenink, H. Roebbers, J. Sunter, P. H. Welch, and D. Wood (Eds.), *Communicating Process Architectures 2005*, pp. 71-107, IOS Press, Amsterdam, 2005.

[2] K. Chalmers, J. Kerridge, and I. Romdhani, "Mobility in JCSP: New Mobile Channel and Mobile Process Models," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), *Communicating Process Architectures 2007*, pp. 163-182, IOS Press, Amsterdam, 2007.