CSP Networking for Java (*JCSP.net*)

Peter H. Welch, Jo R. Aldous and Jon Foster

Computing Laboratory, University of Kent at Canterbury, England, CT2 7NF

P.H.Welch@ukc.ac.uk, jra@dial.pipex.com, jon@jon-foster.co.uk

Abstract. JCSP is a library of Java packages providing an extended version of the CSP/occam model for *Communicating Processes*. The current (1.0) release supports concurrency within a single Java Virtual Machine (which may be multi-processor). This paper presents recent work on extended facilities for the dynamic construction of CSP networks across distributed environments (such as the *Internet*). Details of the underlying network fabric and control (such as machine addresses, socket connections, local multiplexing and de-multiplexing of application channels, acknowledgement packets to preserve synchronisation semantics) are hidden from the JCSP programmer, who works entirely at the application level and CSP primitives. A simple brokerage service – based on channel names – is provided to let distributed JCSP components find and connect to each other. Distributed JCSP networks may securely evolve, as components join and leave at run-time with no centralised or pre-planned control. Higher level brokers, providing user-defined matching services, are easy to bootstrap on top of the basic *Channel Name Server* (CNS) – using standard JCSP processes and networking. These may impose user-defined control over the structure of network being built. JCSP network channels may be configured for the automatic loading of class files for received objects whose classes are not available locally (this uses the network channel from which they were received – the sender must have them). Also provided are *connection* channels (for extended bi-directional transactions between clients and servers) and *anonymous* channels (to set up communications without using any central registry – such as the given CNS). The aims of ***JCSP.net*** are to simplify the construction and programming of dynamically distributed and parallel systems. It provides high-level support for CSP architectures, unifying concurrency logic within and between processors. Applications cover all areas of concurrent computing – including e-commerce, *agent* technology, home networks, embedded systems, high-performance clusters and *The Grid*.

1 Introduction

JCSP [1, 2, 3, 4, 5] provides direct expression in Java for concurrent systems based on Hoare's algebra of *Communicating Sequential Processes* (CSP [6, 7, 8]). It follows the model pioneered by the **occam** concurrency language [9] in the mid-1980s, which was the first commercial realisation of the theory that was both efficient and secure.

P.M.A. Sloot et al. (Eds.): ICCS 2002, LNCS 2330, pp. 695–708, 2002.

JCSP extends the **occam** model by way of some of the proposed extras (such as shared channels) for the *never implemented* **occam3** language [10] – and by taking advantage of the dynamic features of Java (such as recursion and runtime object construction). These latter, however, throw concurrency security issues (such as race hazards) back on the system designer (that **occam**, with its compile-time memory allocation, could take care of itself). An important and open research issue is how to extend the security mechanisms of **occam** to cover dynamic concurrent systems – but that is not the subject of this paper.

JCSP views the world as *layered networks of communicating processes*, each layer itself being a process. Processes do not interact directly with other processes – only with CSP *sychronisation objects* (such as communication channels, event barriers, shared-memory CREW locks) to which groups of processes subscribe. In this way, *race hazards* are (largely) designed out and *deadlock/livelock* analysis is simplified and can be formalised in CSP for mechanically assisted verification. The strong de-coupling of the logic of each process from any other process means we only have to consider one thing at a time – whether that thing is a process with a serial implementation or a network layer of concurrent sub-processes. A consequence is that CSP concurrency logic scales well with complexity. None of these properties are held by the standard Java concurrency model based on *monitors* and *wait conditions*.

The channel mechanism of CSP lends itself naturally to distributed memory computing. Indeed, this was one of the most exciting attributes of **occam** systems mapped to *transputer* [11] networks. Currently, JCSP only provides support for shared-memory multiprocessors and, of course, for uniprocessor concurrency. This paper reports recent work on extended facilities within JCSP for the dynamic construction, and de-construction, of CSP networks across distributed environments (such as the *Internet* or tightly-coupled clusters).

The CTJ Library [12, 13], which is an alternative to JCSP for enabling CSP design in Java (with an emphasis on real-time applications), already provides *TCP/IP socket* drivers that can be plugged into its channels to give networked communication. Independent work by Nevison [14] gives similar capabilities to JCSP. The work presented here offers a higher level of abstraction that lifts many burdens of dynamic network algorithms from the JCSP programmer, who works only at the application level and CSP primitives.

JCSP.net provides mechanisms similar to the *Virtual Channel Processor* (VCP) of the T9000 *transputer* [11]. Users just *name* networked channels they wish to use. Details of how the connections are established, the type of network used, machine addresses, port numbers, the routing of application channels (e.g. by multiplexing on to a limited supply of socket connections) and the generation and processing of acknowledgement packets (to preserve synchronisation semantics for the application) are hidden. The *same* concurrency model is now used for networked systems as for internal concurrency. Processes may be designed and implemented without caring whether the sychronisation primitives (channels, barriers etc.) on which they will operate are networked or local. This is as it should be.

This paper assumes familiarity with core JCSP mechanisms – channel interfaces, *any-one* concrete channels, *synchronized* communication, buffered *plugins*, processes, parallel process constructors and *alternation* (i.e. passive waiting for one from a set of *events* to happen).

2 Networked JCSP (*JCSP.net*)

We want to use the same concurrency model regardless of the physical distribution of the processes of the system. Figure 1(a) shows a six process system running on a single processor (**M**). Figure 1(b) shows the *same* system mapped to two machines (**P** and **Q**), with some of its channels stretched between them. The semantics of the two configurations should be the same – only performance characteristics may change.

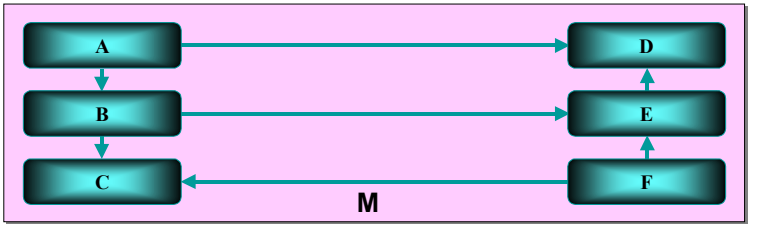

Figure 1(a). Single processor system

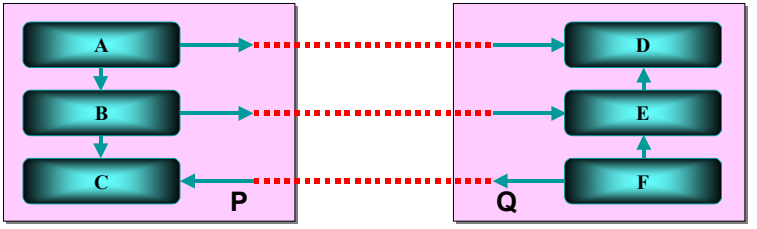

Figure 1(b). Two processor system

2.1 Basic Infrastructure

To implement Figure 1(b), a bi-directional link must be established between the machines and the application channels multiplexed over this. Figure 2 shows broad details of the drivers for such a link. Each application channel is allocated a unique *Virtual Channel Number* (VCN) on each machine using it. For example, the channel connecting processes **A** and **D** has VCN `97` on machine **P** and `42` on machine **Q**.

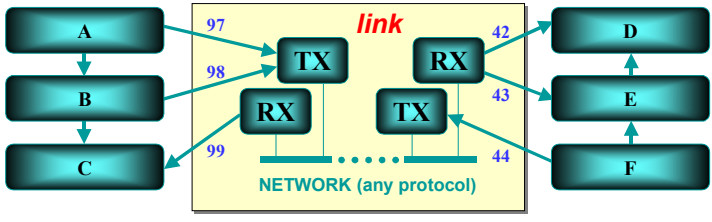

Figure 2. Virtual channel numbers and a link

Application messages sent across the network contain source and target VCNs, as well as the underlying network's requirements (e.g. target network address) and the application data. This information is automatically added and removed as messages fly through the ***JCSP.net*** infrastructure. Applications processes just read and write normal channel interfaces and may remain ignorant that the implementing channels are networked. The source VCN is needed to route acknowledgement packets back to the source infrastructure to preserve application synchronisation semantics (unless the networked application channel is configured with a `jcsp.util` *overwriting buffer*).

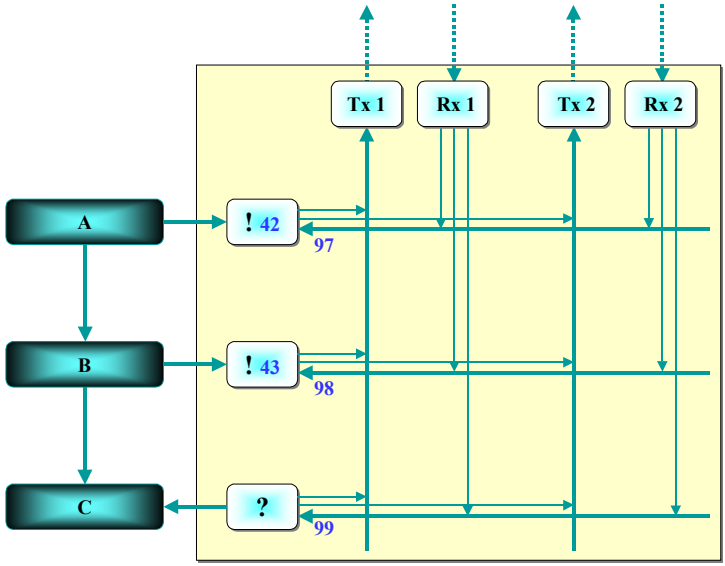

Figure 3. ***JCSP.net*** crossbar (2 links, 2 writers, 1 reader)

The supporting ***JCSP.net*** infrastructure on each machine is itself a simple JCSP crossbar, dynamically set up as `jcsp.net` channels are created. Figure 3 shows this for two *external network links* (to two different machines) and the three *networked application channels* to/from processes **A**, **B** and **C** on machine **P** – figure 1(b). The current state of this crossbar is implemented by just five JCSP *any-one* channels.

Each network link is driven by a pair or processes, **`Tx`**/**Rx**, responsible for low-level transmission/reception (respectively) over the network. Each application network input channel is managed by an Input Control Process (ICP), labelled "**`?`**". Each application network output channel is managed by an Output Control Process (OCP), labelled "**`!`**". Each control and network transmission process is the server end of a JCSP `Any2OneChannel`. Each control process can write to any of the **`Tx`** channels. Each **Rx** process can write to any of the control process channels.

The channels connecting application output processes to their OCPs are instances of JCSP *Extended Rendezvous* (ER) channels. These are similar to normal CSP channels (i.e. unbuffered and fully synchronising). Readers block until a writer writes. Writers block until a reader reads *and* (here is the difference) *subsequently releases it*

explicitly. The period between a reader reading and its explicit release of the channel (and blocked writer) is the extended rendezvous. In this period, the reader may do anything except read from that channel again.

As indicated in Figure 3, each OCP has burnt into it the target VCN of the network channel it is supporting. It also locates and holds on to the local `Tx` channel it needs to use – this will not change. Its actions follow a simple cycle. It waits for and reads an object (reference) from an application process, wraps this is a (pooled and recycled) message wrapper containing source and target VCNs, forwards that to the appropriate `Tx` process and waits for an acknowledgement on its server channel. Assuming all is well with that acknowledgment, it releases the application writing process. If there is an error reported by the acknowledgment packet (e.g. network or remote node failure), it releases the writer but causes the writer to throw a suitable exception.

An ICP listens (at the start of its cycle) on its server channel. An arriving packet contains an incoming application object, source VCN and `Tx` link reference (for the link on which the network packet arrived). It outputs the object down the channel to the application reader and, when that has been taken, sends an acknowledgment packet back to the `Tx` link (tagged with the source VCN). The channel between an ICP and its reader is not an ER channel and may be buffered (in accordance to the needs of the application).

Each `Tx` process is set up connected to the correct remote machine. All data from application channels on this machine to the remote one (and acknowledgements of data in the other direction) go via this process. Its behaviour is trivial: wait for packets arriving on its server channel and output them to the network. It abstracts details of the network from the rest of the infrastructure – making *different* types of network easy to manage.

Each `Rx` process cycles by waiting for a network packet to arrive. This will contain a target VCN, which is used to find the relevant ICP server channel and to which the packet is forwarded – end of cycle. Again, details of the type of network is abstracted within this process.

It is crucial to avoid network packets unread by an application process blocking other traffic. Application network channels must not get in each others' way within the infrastructure. A `Tx` server channel needs no buffering for this – we may assume the `Rx` process at the other end is never blocked (see later in this paragraph). An OCP server channel needs no buffering – the OCP will be waiting for acknowledgement packets routed to it. An ICP server channel needs buffering – but this will be limited to the number of network links currently open (and, hence, `Rx` processes that write to it will never block). That limit requires that the same JCSP networked channel cannot be bound to more than one application output channel per machine – which is, in fact, no limitation since any number of application processes on a machine may securely share that channel. This is ensured by the ***JCSP.net*** channel constructors.

Of course, such reasoning should be formalised and verified to build (even) greater confidence. ***JCSP.net*** infrastructure, however, is directly modelled by a CSP network so this should not be intractable.

Finally, we re-emphasise that the ***JCSP.net*** *programmer* needs no knowledge of the details in this section. JCSP *processes* only see channel *interfaces* and do not care whether the *actual* channels plugged into them are local or networked.

2.2 Establishing Network Channels (the *Channel Name Server*)

JCSP.net does not require networked applications to be pre-planned and mapped on to a pre-arranged configuration of processors. The model is very dynamic – processes find each other simply by knowing (or finding out) the names of the network channels they need to use.

The mechanism is brokered by a *Channel Name Server* (CNS), which maintains a table of network addresses (e.g. IP-address/port-number), channel type (e.g. *streamed*, *overwriting*) against user-defined channel names (which can have a rich structure akin to URLs). The CNS provides a minimum functionality to make dynamic connections. More sophisticated brokers may be built as user applications, bootstrapped on top of the CNS (see Section 2.5). Once processes know where each other are, network connections can be built without registering at a CNS or higher-level broker (see Section 2.4). If different network types are required, each needs its own CNS.

A CNS must be started first and each application on each machine must be able to find it. By default, this is obtained from an XML file the first time a network channel is created by an application on each machine. Alternatively, there are various ways of setting it directly.

Within a processor, networked application channels are either *to-the-net* or *from-the-net*. They may have *one* or *any* number of application processes attached – i.e. `One2NetChannel`, `Any2NetChannel`, `Net2OneChannel` and `Net2AnyChannel`. The '`2Net`' channels implement only `write` methods – the '`Net2`' channels only `read`s. As usual, only '`2One`' channels may be used as *guards* in an `Alternative`.

To construct networked channels using the CNS, we only need to know their names. For example, at the receiving end:

```
Net2OneChannel in = new Net2OneChannel ("ukc.foo");
```

allocates an ICP and target VCN for this channel and registers the name `"ukc.foo"` with the CNS, along with that VCN and the network address for a *link-listener* process on this machine (which it constructs and starts if this is the first time). If the CNS cannot be found (or the name is already registered), an exception is thrown.

```
One2NetChannel out = new One2NetChannel ("ukc.foo");
```

The above allocates an OCP and source VCN for this channel and asks the CNS for registration details on `"ukc.foo"`, blocking until that registration has happened (time-outs may be set that throw exceptions). The target VCN from those details is given to the OCP. If a network connection is not already open to the target machine, a connection is opened to the target's *link-listener* process and local **`Tx`**/**`Rx`** processes created. Either way, the **`Tx`** server channel connecting to the target machine is given to the OCP. On the receiving machine, the *link-listener* process creates its local **`Tx`**/**`Rx`** processes and passes it the connection information.

Unlucky timing in the creation of opposing direction network channels may result in an attempt to create two links between the same pair of machines. However, this is resolved by the machine with the *'lower'* network address abandoning its attempt.

If network channels are being constructed sequentially by an application, all input channels should be set up first – otherwise, there will be deadlock!

2.3 Networked Connections (Client-Server)

JCSP.net *channels* are uni-directional and support *any-one* communication across the network. This means that any number of remote machines may open a named *output* network channel and use it safely – their messages being interleaved at the *input* end. Establishing *two-way* communication, especially for a prolonged conversation, can be done with *channels*, but is not easy or elegant.

So, ***JCSP.net*** provides *connections*, which provide logical pairs of channels for use in networked *client-server* applications. For example, at the *server* end:

```
Net2OneConnection in = new Net2OneConnection ("ukc.bar");
```

sets up infrastructure similar to that shown in Figure 3, except that the *Server Control Process* (SCP, instead of ICP) provides *two-way* communication to its attached application process. Similarly:

```
One2NetConnection out = new One2NetConnection ("ukc.bar");
```

sets up a *Client Control Process* (CCP, instead of OCP) for *two-way* communication to its attached application process.

Application processes only see *client* and *server* interfaces (rather than *writer* and *reader* ones) to these connections. Each provides *two-way* communications:

```
interface ConnectionClient {
  public void request (Object o);     // write
  public Object reply ();             // read
  public boolean stillOpen ();        // check (if unsure)
}

interface ConnectionServer {
  public Object request ();           // read
  public void reply (Object o);       // write & close
  public void reply (                 // write & maybe close
    Object o, boolean keepOpen
  );
}
```

JCSP.net *connections* are bi-directional and support *any-one* client-server transactions across the network. This means that any number of remote machines may open a named *client* network connection and use it safely – transactions being dealt with atomically at the *server* end (see Figure 5 in Section 2.4 below).

A transaction consists of a sequence of `request/reply` pairs, ending with the server making a *write-and-close* `reply`. If necessary, the client can find out when the transaction is finished – by invoking `stillOpen` on its end of the connection. The server infrastructure locks the server application process on to the remote client for the duration of the transaction. Other clients have to wait. Any application attempt to violate the `request/reply` sequence will raise an exception.

No network acknowledgements are generated for any *connection* messages. Clients must commit to read `reply` messages from the server and this is sufficient to keep them synchronised. Note that *connections* may not be buffered.

Note, also, that a connection is not *open* until the first `reply` has been received. Clients may open several servers at a time – but only if all clients opening intersecting sets of servers open them in an agreed sequence. Otherwise, the classic deadlock of *partially acquired resources* will strike.

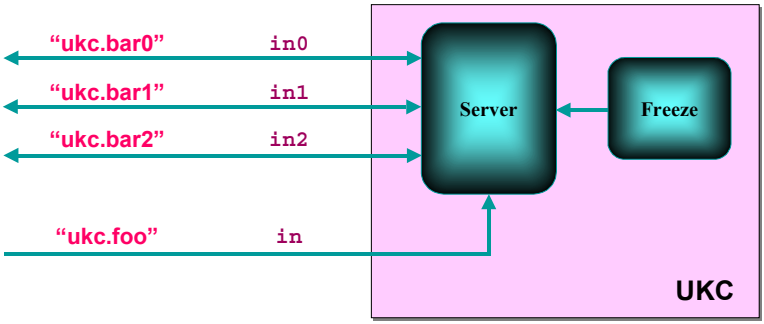

Figure 4. Servicing many events (network connections, network and local channels)

Network connections are *ALTable* at their server ends – i.e. '`2One`' connections may be used as *guards* in an `Alternative`. For example, `Server` in Figure 4 may wait on events from any of its three networked server *connections* (indicated by the double-arrowed lines), its networked input *channel* and its local input channel.

Finally, we note that JCSP is extended to support *connections* within an application node (i.e. they do not have to be networked). As for channels, processes only see the *interfaces* to their connections and do not need to know whether they are networked.

2.4 Anonymous Channels/Connections

Networked channels and connections do not have to be registered with the CNS. Instead, *input* ends of channels (or *server* sides of connections) may be constructed anonymously. For example:

```
Net2OneChannel in = new Net2OneChannel ();
Net2OneConnection in2 = new Net2OneConnection ();
```

Remote processes cannot find them *by name* using the CNS. But they can be told where they are by communication over channels/connections previously set up with the process(es) that created them. '`Net2`' channels and connections contain location information (*network address* and *VCN*). This can be extracted and distributed:

```
NetChannelLocation inLocation = in.getLocation ();
NetConnectionLocation in2Location = in2.getLocation ();

toMyFriend.write (inLocation);
toMyOtherFriend.write (in2Location);
```

Remember that your friends may distribute this further!

A process receiving this location information can construct the *output*/*client* ends of the networked *channels*/*connections*. For example:

```
NetChannelLocation outLocation =
  (NetChannelLocation) fromMyFriend.read ();

One2NetChannel out =
  new One2NetChannel (outLocation);
```

and on, perhaps, another machine:

```
NetConnectionLocation out2Location =
  (NetConnectionLocation) fromMyOtherFriend.read ();

One2NetConnection out2 =
  new One2NetConnection (out2Location);
```

Figure 5 shows an example use of anonymous connections. The **UKC** machine holds the *server* end of a CNS-registered connection `"ukc.bar"` and provides a number of (identical) `Server` processes for it. This connection has been picked up by applications on machines **P** and **Q**, which have set up *client* ends for `"ukc.bar"`. Now, only one client and one server can be doing business at a time over the shared connection. If that business is a lengthy `request`/`reply` sequence, this will not make good use of the parallel resources available.

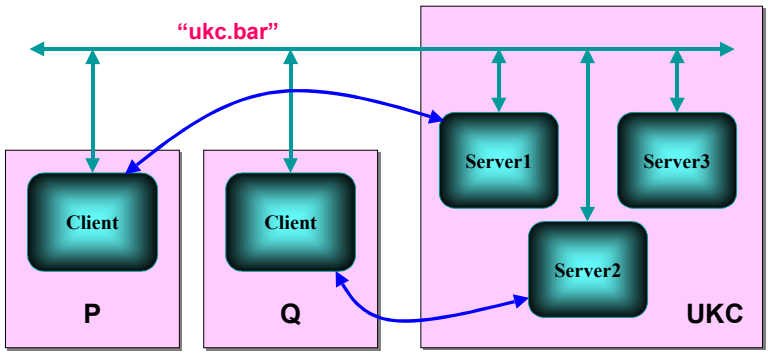

Figure 5. Registered and anonymous connections used in a server farm

Transactions over a *shared resource* should be as brief as possible – in this case: (client) *'gimme a connection'* and (server) *'OK, here's an unregistered one I just made'*. The publicly registered connection is now only used to let a remote client find one of the servers in the **UKC** *farm*. Lengthy *client-server* transactions are conducted over dedicated, and unregistered, connections (shown as curved lines in Figure 5). All such transactions may continue *in parallel*. At the end of a transaction, the server disconnects and discards the unregistered connection – any attempted reuse by the client will fail (and throw exceptions). Note that the unregistered connections will be multiplexed over the link fabric (e.g. *TCP/IP sockets*) that must already be present.

2.5 User Defined Brokers

If you want a matching service more sophisticated than the *simple naming* provided by the CNS, build your own broker as a server for a CNS-registered connection. Anyone knowing (or told) the name of that connection can use your new broker. Applications may then register service details with it, along with the location of their networked server connection (or input channel) supporting it. The broker can be asked for details of registered services or to search for a service matching user requirements. The broker returns the locations of matched services. The networked connections or channels supporting these services – as far as the CNS is concerned – are *anonymous*.

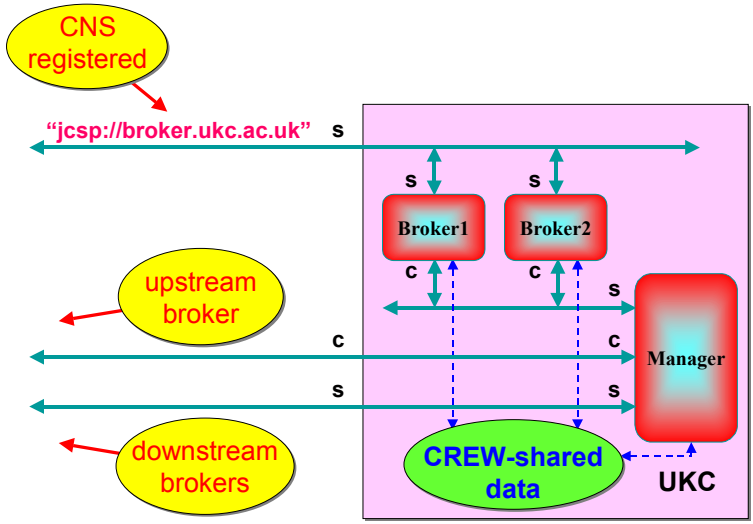

Figure 6. Component of a dynamically scalable server farm

This broker can be structured and operated like the server farm in Figure 5. However, Figure 6 extends this to show the design of a component of a dynamically scalable and distributed service (which may be a model for any server – not just the broker discussed here). The temporary connections established between the brokers and their clients (see Figure 5) are not shown here.

The broker processes are clients on an (internal) connection to their manager. The manager provides service to its local brokers and to the managers of external brokers *downstream*. It also holds a client connection to an external *upstream* manager. The global network of these components forms a *tree topology* of client-servers, which is guaranteed to be *deadlock-free* [15].

Broker processes report their use to their local manager. If the node is in danger of running out, extra ones can be created dynamically by the manager. The manager and its local brokers share a database protected by a *Concurrent-Read-Exclusive-Write* (CREW) lock, which is one of the core capabilities provided by JCSP. The managers in each node regularly keep in touch with each other about the nature of their holdings and can inform their brokers to re-direct clients as necessary.

2.6 Networked Class Loading (Remote Process Launching and Agents)

Objects sent across ***JCSP.net*** channels or connections currently use Java *serialization*. This means that the receiving JVM has to be able to load – or already have loaded – the classes needed for its received objects. Default ***JCSP.net*** communications deliver *serialized* data. If an object of an unknown class arrives, a *class-not-found* exception is thrown as *de-serialization* is attempted.

To overcome this, ***JCSP.net*** channels and connections can be set to download those class files automatically (if the receiving `ClassLoader` cannot find them locally) – the sending process, after all, must have those files! ***JCSP.net*** machine nodes cache these class files locally in case they themselves need to forward them.

Consider the simple *server farm* shown in Figure 5. Let the (unregistered) connections dynamically set up between a *client* and *server* process be *class-loading*. Suppose the client has some work it needs to farm out and that that work is in the form of a ***JCSP*** process. Let `out` be its networked connection to the remote server:

```
CSProcess work = new MyProcess (...);    // To be done ...
out.request (work);                      // Send to worker.
work = (MyProcess) out.reply ();         // Get results.
```

Results may be safely extracted from the returned `work` – a non-running process.

At the server end, let `in` be its networked connection to the remote client:

```
CSProcess work =                         // Class file may
 (CSProcess) in.request ();              // be downloaded.
work.run ();                             // Do the work.
in.reply (work);                         // Send results.
```

If this is the first time the server farm has been given an instance of `MyProcess` to run, its class will be automatically downloaded from the `in` connection (and cached).

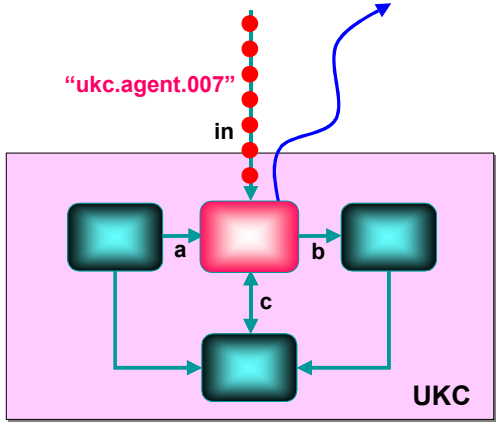

Figure 7. Mobile processes (agents)

The `MyProcess` process launched above was self-contained. It is just as easy to download and launch processes that plug into and engage with local resources – which brings us into *agent* technologies. Figure 7 shows an agent server environment on the **UKC** machine. Agents arrive on the CNS-registered (`"ukc.agent.007"`) input channel, plug into the local network, run and then move on. Code for the actual agent receiving process could take the simple form:

```
while (running) {
  Bond james = (Bond)in.read ();          // Download maybe?
  james.plugin (a, b, c);                 // Connect locally.
  james.run ();                           // Do the business.
  NetChannelLocation escapeRoute =
    james.getNextLocation ();             // Where next?
  One2NetChannel escape =
    One2NetChannel (escapeRoute);         // Set it up.
  running = james.getBomb ();             // Leave politely?
  escape.write (james);                   // Goodbye.
  escape.disconnect ();                   // Lose channel.
}
```

In the above loop, `Bond` is just an interface extending `jcsp.lang.CSProcess` with the `plugin`, `getNextLocation` and `getBomb` methods needed later. If the actual process read in is of a class not seen before, its code is downloaded. The agent is then plugged into the local channels and run. As it runs, its behaviour depends on its code, any data it brought with it and what it finds in the local environment. Before it finishes, it must have decided upon where it wants to go next (and whether it wants to *bomb* the loop of this agent handler). After it finishes, the handler gets the next location data, sets up the escape channel, sends the agent down the channel, deletes the escape channel local infrastructure and (maybe) loops around to wait for the next `Bond`.

3 Summary and Discussion

This paper presents some of the key facilities of ***JCSP.net*** and the JCSP mechanisms used to implement them. Application concepts include *any-one* networked channels and *connections* (that enable extended *client-server* conversations), a *Channel Name Server* (CNS) for the dynamic construction of application networks, and *anonymous* channels and connections (that evade the normal CNS registration). Applications outlined include a scalable parallel and multiprocessing broker for user-defined matching services (itself CNS-registered so that applications can find it), remote process launching on worker farms and mobile agents.

Lack of space prevents the description of several other capabilities: e.g. networked multiple barrier synchronisation (including two-phase locks), networked buckets (a non-deterministic version of barriers), networked *CREW* locks and networked *CALL* channels (which provide a method call interface to channel semantics). The semantics and APIs of their *non-networked* cousins are in the standard JCSP documentation [1]. ***JCSP.net*** also provides for the migration of *server connection ends* (respectively

input channel ends) between networked machines in a way that loses no data and is transparent to any of the processes at the *client* (respectively *output*) *ends* – i.e. they are unaware of the migration.

Also only outlined in this paper is the implementing JCSP infrastrucure. The basic software crossbar of concurrent processes supporting standard network channels is explained (Sections 2.1 and 2.2), but not the additional structures needed to deal with *streaming* (buffered) channels, *class-downloading* channels, and *connections* (that, for example, need two *Virtual Channel Numbers* per *Server Control Process*).

Such extra details will be presented in later papers, along with performance results from simple benchmarks and a range of classical parallel and distributed computing demonstrators.

The aims of ***JCSP.net*** are to provide a *simple* way to build complex, scalable, distributed and dynamically evolving systems. Its primitives are based upon well-established CSP operations, so that its applications may be directly and formally modelled (and *vice-versa*). It allows the expression of concurrency at *all* levels of system – both *between* parallel running devices (whether home, cluster or *Internet* distributed) and *within* a single machine (which may be multi-processor). The same concepts and theory are used regardless of the physical realisation of the concurrency – shared memory and distributed memory systems are programmed in the same way.

This contrasts with the models provided by PVM, MPI and BSP, none of which address concurrency *within* a processing node. We believe that concurrent *processes* are too powerful an idea to leave out of our armoury of system design, analysis, implementation and verification tools. Again, this conflicts with received wisdom that says that concurrent algorithms are *harder* than equivalent serial ones. There is no need to stir up an argument – time and experience will tell. Current experiences with *large* serial applications are not promising. We cite the design illustrated in Figure 3 for an experience of the simplicity wrought, and complex functionality quickly generated, by (CSP based) concurrency.

The status of ***JCSP.net*** is that it is being alpha-tested within our research group [16] at UKC. Special thanks are accorded to Brian Vinter (of the Southern University of Denmark, currently visiting us on sabatical) for considerable assistance. Details of beta-testing will be posted on the ***JCSP*** website [1]. We are negotiating to see if commercial support can be provided.

JCSP.net is part of a larger project on language design, tools and infrastructure for scalable, secure and simple concurrency. In particular, we are developing the multi-processing **occam** language with various kinds of dynamic capability (see the ***KroC*** website [17]) that match the flexibility available to Java systems, but which retain strong semantic checks against major concurrency errors (such as *race hazards*) and ultra-low overheads for concurrency management (some two to three orders of magnitude lighter than those accessible to Java). The Java and **occam** concurrency work feed off each other in many interesting ways – for example, there will be a ***KroC.net***.

References

1. Welch, P.H., Austin, P.D.: JCSP Home Page, http://www.cs.ukc.ac.uk/projects/ofa/jcsp/ (2002)
2. Welch, P.H.: Process Oriented Design for Java – Concurrency for All. In: PDPTA 2000 volume 1. CSREA Press (June 2000) 51-57
3. Lea, D.: Concurrent Programming in Java (Second Edition): Design Principles and Patterns. The Java Series, Addison-Wesley (1999) section 4.5
4. Welch, P.H.:, Java Threads in the Light of occam/CSP. In: Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21. IOS Press (Amsterdam), ISBN 90 5199 391 9 (April 1998) 259-284
5. Welch, P.H.:, Parallel and Distributed Computing in Education. In: J.Palma et al. (eds): *VECPAR'98*, Lecture Notes in Computer Science, vol. 1573. Springer-Verlag (June 1998)
6. Hoare, C.A.R.:, Communicating Sequential Processes, CACM, 21-8, (1978) 666-677
7. C.A.R.Hoare. Communicating Sequential Processes. Prentice Hall ISBN 0 13 153289 8 (1985)
8. A.W.Roscoe. The Theory and Practice of Concurrency. Prentice Hall, ISBN 0 13 674409 5 (1997)
9. Inmos Limited. occam2.1 Reference Manual, Technical Report. http://wotug.ukc.ac.uk/parallel/occam/documentation/ (1989)
10. Inmos Limited. occam3 Reference Manual, Technical Report. http://wotug.ukc.ac.uk/parallel/occam/documentation/. (1992)
11. May, M.D., Thompson, P.W., Welch, P.H.: Networks, Routers and Transputers. IOS Press, ISBN 90 5199 129 0 (1993)
12. Hilderink, G.H.: CTJ Home Page. http://www.rt.el.utwente.nl/javapp/ (2002)
13. Hilderink, G.H., Broenink, J., Vervoort, W., Bakkers, A.W.P: Communicating Java Threads, In: Bakkers, A.W.P et al. (eds): Parallel Programming in Java, Proceedings of WoTUG 20, Concurrent Systems Engineering Series, vol. 50. IOS Press (Amsterdam), ISBN 90 5199 336 6 (1997) 48-76
14. Nevison, C.: Teaching Distributed and Parallel Computing with Java and CSP. In: Proceedings of the 1st ACM/IEEE International Symposium on Cluster Computing and the Grid, Brisbane, Australia (May 2001)
15. Martin, J.M.R., Welch, P.H.: A Design Strategy for Deadlock-Free Concurrent Systems. Transputer Communications 3(4). John Wiley & Sons, ISSN 1070 454 X (October 1996) 215-232
16. Welch, P.H.: Concurrency Research Group Home Page, Computing Laboratory, University of Kent at Canterbury http://www.cs.ukc.ac.uk/research/groups/crg/ (2002)
17. Welch, P.H., Moores J., Barnes, F.R.M., Wood, D.C.: KRoC Home Page. http://www.cs.ac.ukc.ac/projects/ofa/kroc/ (2002)