Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

A Critique of JCSP Networking

Kevin CHALMERS, Jon KERRIDGE and Imed ROMDHANI

School of Computing, Napier University, Edinburgh, EH10 5DT
{k.chalmers, j.kerridge, i.romdhani}@napier.ac.uk

Abstract. We present a critical investigation of the current implementation of JCSP Networking, examining in detail the structure and behavior of the current architecture. Information is presented detailing the current architecture and how it operates, and weaknesses in the implementation are raised, particularly when considering resource constrained devices. Experimental work is presented that illustrate memory and computational demand problems and an outline on how to overcome these weaknesses in a new implementation is described. The new implementation is designed to be lighter weight and thus provide a framework more suited for resource constrained devices which are a necessity in the field of ubiquitous computing.

Keywords. JCSP, JCSP.net, distributed systems, ubiquitous computing.

Introduction

JCSP (Communicating Sequential Processes for Java) [1, 2] is a Java implementation of Hoare's CSP [3] model of concurrency. Previous work was presented [4] that brought about the integration of the differing versions of JCSP, as well as augmenting and extending some of the underlying mechanisms inside the core package. In this paper, we present information on the current implementation of the network package and raise some issues with the current approach. This allows us to address these limitations in a new implementation of JCSP networking, while also attempting to bring the networking package closer to the same level of functionality as the core package.

The rest of this paper is broken up as follows. In Section 1, we present some background information on JCSP and network based approaches for CSP. In Section 2, motivation for this work is given. Section 3 provides a description of the current architecture of JCSP Networking and in Section 4 an analysis and criticism of the current implementation is given. Finally, in Section 5, conclusions are drawn and future work considered.

1. Background

In this article we are concerned with the networking capability of JCSP [5] as opposed to the core functionality. The main library for JCSP is aimed at concurrency internal to a single JVM, whereas the network library was designed to permit transparent distributed parallelism using the same basic interfaces as present in core JCSP. Unlike core JCSP, there is no channel object that spans the network. Instead, JCSP networking operates using a channel end concept, with a node (a single JVM within the network) declaring an input end, and other nodes connecting to this input end via an output end. The input end and output end make up a virtual networked channel between two nodes. The input end and output end

have the same interfaces as `ChannelInput` and `ChannelOutput` in the core JCSP package. Beyond this, JCSP networking adds little to the functionality of the main packages, and due to the recent improvements in JCSP [4] can now be considered lacking certain constructs that would make it on par with core JCSP.

Of particular importance in this case are the lack of a basic network synchronisation primitive (the JCSP `Barrier`) and the multi-way synchronisation primitive (JCSP `AltingBarrier`) which leads to a lack of transparency between locally synchronizing processes and distributed synchronizing processes. These constructs and other, Java 1.5 specific, considerations were given as future work in [4]. This paper brings these constructs closer to implementation by illustrating the need to address some of the underlying architectural decisions made for JCSP networking. First we shall look at other implementations of CSP inspired networking architectures.

1.1 Network Based CSP Implementations

There is now a wealth of CSP inspired distributed concurrency libraries available, ranging from **occam-π** and the pony architecture [6] through C++CSP [7] and Python [8]. Most base their architecture on the T9000 model [9] for creating virtual channels across a communication mechanism.

JCSP Networking [5] enables the virtual connections to be created via `NetChannelLocation` structures sent between nodes to allow virtual connections to be created. A `NetChannelLocation` signifies the location of a `NetChannelInput` end which a `NetChannelOutput` end can connect to; the input end acts as a server connection to an output end. The location structures are fairly trivial, being formed by the address of the node on which the channel is declared, and a Virtual Channel Number (VCN) uniquely identifying the channel on said node, although other methods involving channel names may be used. Initial interaction between nodes is usually managed by a Channel Name Server (CNS) which allows registration of channel names by the server end of a connection, and the resolution of these names – thus providing the location – by a client end of a connection. After initial interaction, locations can be exchanged directly using networked channels or all channels may be declared with the CNS. `NetConnections` are also available, and methods for permitting the mobility of channels and processes (via code mobility [10]) are also available [11].

C++CSP networking [7] enhances the original C++CSP library [12] by adding the capability for two C++CSP programs to communicate via TCP/IP based sockets. Unlike JCSP, there is no CNS – channels connect using unique names on the node. VCNs exist in the underlying exchanges between nodes. C++CSP networking is limited by how it sends data, due to differing machine architectures that may be in operation, and a lack of an object serialisation approach in C++ similar to that found in Java. Recently C++CSP was updated to version 2 [13], a move that concentrated more on utilizing multi-core systems than implementation of a networking architecture.

pony [6] is the **occam-π** approach to networking, and shares a number of similarities with JCSP. Instead of a CNS, pony uses an Application Name Server (ANS) and controls the system of nodes via a main node. Channel mobility is also controlled and there is no current implementation of process mobility. The architecture of pony has been inspired by the need to implement transparent concurrency and parallelism in a cluster computing environment, which are more controlled than standard distributed systems architectures, towards which JCSP is more aimed.

CSP.NET [14] is an implementation of CSP for Microsoft's .NET Framework, inspired a great deal by JCSP. Developed in C#, the main advantage of this library over

JCSP is the number of languages in .NET that can now utilize the library. CSP.NET does rely on the remoting capabilities of .NET, and is therefore technology restricted – remoting being the RPC system built explicitly into .NET, requiring .NET to operate. JCSP operates in a manner that is decoupled from the communication mechanism, and is thus independent of it. Initial performance analysis of CSP.NET has shown little difference in performance in comparison to JCSP. The most recent version of CSP.NET is now available commercially (www.axon7.com) and no longer relies on specific features of .NET, but this has yet to be formally reported.

Finally, PyCSP [8] is implemented in the Python programming language. Although at last reporting only having basic networking capabilities, the current approach uses remote objects as opposed to an underlying communication mechanism. The aim of PyCSP appears to be geared towards cluster computing, making a solid networking infrastructure essential in the future.

1.2 Performance Evaluations of Network Based CSP Implementations

Some work towards measuring performance of network enabled CSP implementations has been conducted in previous research. Many of these approaches have focused only briefly on the performance of the communication mechanism, and instead examine the performance of parallelized tasks within the architecture. Brown [7] has examined latency and performance overheads in C++CSP, but no extensive testing of performance of the communication mechanism using different data packet sizes was made. Instead, work was allocated and different packet sizes used to measure performance. Although this can lead to some information about the communication performance, it does not analyze it in great enough detail to find the difference in performance between C++CSP networking and standard communication mechanisms.

Schweigler [6] has done extensive tests examining the CPU overhead and throughput of pony, as well as some comparisons with JCSP and work allocation. Again, little analysis as to the actual costs of sending messages between nodes is given, and most of the conclusions on such overheads are interpreted from the throughput and comparison when parallelizing a task.

Lehmberg's analysis of CSP.NET [14] provides only simple analysis of performance without comparison to other approaches, although a brief comparison to JCSP has been made. The authors note that the tests performed are by no means thorough enough to constitute a benchmark.

For JCSP, little real analysis of the performance of the communication mechanisms has been made. Schaller [15] assessed the different speed ups of Java parallel computing libraries when performing certain tasks across multiple nodes. Vinter [16] has examined similar properties with other packages in Java, performing different tasks but forgoing any analysis of the communication mechanism. Kumar [17] has examined JCSP performance in the context of multiplayer computer games, and although providing some interesting results on the scalability of JCSP, little analysis of the communication was made.

To fully understand how suitable JCSP is in comparison with other approaches to communication, analysis of the current mechanisms has to be made. In Section 4, we provide some of the required parameters. First, a brief description of Java serialisation and object migration is presented to help understand these parameters more fully.

1.3 Serialisation and Object Migration in Java

It is important to understand serialisation in Java as it puts into context some of the performance values we shall be discussing in Section 4. Java serialisation is the process of

converting a Java object into a sequence of bytes for storage or transfer via a communication mechanism. This is usually performed by an `ObjectOutputStream` which acts as a filter on another output stream type to serialize objects. Reflection (the ability to interrogate an object to discover its properties and methods) is used to gather the values within the object and the recreation of the object at the destination from its name (i.e. `java.lang.Integer`).

Control signals are sent with serialized object data to allow recreation of the object at the destination. As a case study, we shall examine the bytes representing an `Integer` object, highlighting some of these control signals as necessary. Further information on serialisation is found in the Java 2 SDK documentation (http://java.sun.com/j2se/1.3/docs/guide/). All values are single bytes unless stated otherwise.

When a new `ObjectOutputStream` is instantiated, a handshake message is sent to allow correct behavior at the destination. Normally four bytes are sent which represent two 16-bit integers: `STREAM_MAGIC` (-21267) and `PROTOCOL_VERSION` (5). These are sent once between an output stream and an input stream. We will not consider the handshake as normal data for this reason.

The next value represents the type of message being sent. For `Integer` this is `TC_OBJECT` (115), which signifies a standard object. Other control signals for base data and arrays also exist. Next is a control signal for the class description, `TC_CLASSDESC` (114), followed by the name of the class as a string (with a 16-bit length header). For `Integer` this is `java.lang.Integer`. A 64-bit integer representing the unique serialisation identifier follows and is used to ensure that the class of the given name is the same at the destination.

A single byte representing control flags to determine how the object was serialized (for example, custom methods can be used within an object) is next and then a 16-bit value representing the number of object attributes. With `Integer` there is only the wrapped primitive integer. The attribute types are given which may be other objects, thus invoking the previous steps for describing the object. For `Integer` the attribute is a primitive integer represented by 'I' (74). The attribute names are given as strings with length headers – `Integer`'s attribute being `value`. A final control signal for the end of the class description is then written – `TC_ENDBLOCKDATA` (120).

If the class of the sent object is a subclass the description of the parent class must also be sent. The parent class may declare attributes not visible to the subclass, but are used by inherited methods. `Integer` extends `Number` (`java.lang.Number`), which has no internal attributes.

When all classes have been described, a byte to signal the attribute values of the object, `TC_BASE` (112) is written. The values of the attributes are written to the stream in the order they were declared. For `Integer`, the 4 bytes representing the 32-bit integer `value` are written.

Taking into account the control signals and descriptions sent for an `Integer` object, we can calculate a total of 77 bytes sent to represent a 4 byte value. This is a significant overhead although `Integer` is a special case with a direct primitive representation of the sent object. However, it does point to the need to avoid serialisation if possible. We have not discussed the recreation of the object at the destination which involves using reflection on the sent name to get the specific class, creating a new instance of the class, and assigning the values of the properties using reflection. This is a costly process in comparison to sending primitives.

Java serialisation tries to overcome overhead problems by using references to previously sent objects and classes within the object stream. For example, if an object is sent twice over a stream, instead of a new complete message, a `TC_REFERENCE` (113) byte is sent, followed by a 32-bit value representing the place in the stream of the object. Class descriptions may also be referenced if an object of the same type is sent more than once. The former cuts the 77 bytes of `Integer` to 5 bytes and the latter cuts the message to 10 bytes. This requires lookup tables within the stream object to accomplish. Over time, these lookup tables can become large, and may cause a memory exception. To combat this, the object streams can be explicitly reset, which causes the output stream to clear its lookup table and send a signal to the input stream to do likewise. This does mean that complete descriptions of classes need to be sent again. Also, the serialiser has no method to determine if an object has been modified between sends, therefore any modification will not be seen at the destination if the object stream is not reset. The serialiser has no method to distinguish between mutable and immutable objects.

The reason to examine serialisation is that JCSP networking relies heavily upon it. To avoid referencing problems, the underlying streams are reset after each message, thus every object is sent as a new object. The first instinct for doing this is the possibility that a user may be optimizing their own application to avoid garbage collection, thus using a pool of messages. JCSP uses this mechanism internally within the networked channels. Each output channel is allocated two message packets that are swapped between sending. If the underlying object streams were not reset after every communication the channel would appear to only send two objects, and then only ever send those two objects. Unfortunately, resetting the streams in this manner leads to other problems, which we shall discuss in Section 4. It should be possible to replace the serialiser with a more efficient implementation, but this is currently left for future work.

2. Motivation

We are examining JCSP in the context of ubiquitous computing (UC) [18], which is the idea of having an environment populated by numerous computationally able elements, that interact together to provide new and unique services. To accomplish UC, dynamic architectures and movement is envisioned; for example to enable agents to move between elements to accomplish goals. An architecture the size and complexity envisioned by UC requires abstractions that enable simpler design and reasoning. The π-Calculus [19] and similar mobile abstractions have been put forward as a possible model for this understanding [20], and we are interested in the practical examination of such abstractions, using JCSP as a case study. The scalability of mobile channels and processes has been shown by the work of Ritson [21], with an architecture involving millions of interacting mobile processes operating simply with no great problems for design or analysis.

The reason for examining JCSP is that it is more mature than similar frameworks when considering distributed mobility [11]. Java is also a more commonly available framework, particularly for mobile devices which enable a close approximation of the capabilities of the elements available in a UC environment. Work on the Transterpreter [22] may lead to **occam-π** being available on more devices, but there is currently no network capability.

We are also hoping to develop a universal mechanism to allow the abstractions that we require within multiple frameworks, a discussion of which is given in Section 5. Although we have taken JCSP as a case study, it cannot be considered that Java will be available on all the elements in a UC environment. As an outcome of our work we also address some of the future work given in [4].

3. Current JCSP Networking

Figure 1 illustrates the current architecture of JCSP networking. Within the diagram, ovals represent processes and rectangles objects. Channels are represented by arrows, and dashed lines between components represent object based connections (uses, contains). Channels that have potential infinite buffering are indicated with an infinity symbol. This diagram appears different from those previously presented for JCSP networking as there is no `NetChannelOutputProcess`. To reduce the number of Java threads supporting the JCSP infrastructure, this has been integrated into `NetChannelOutput` object.

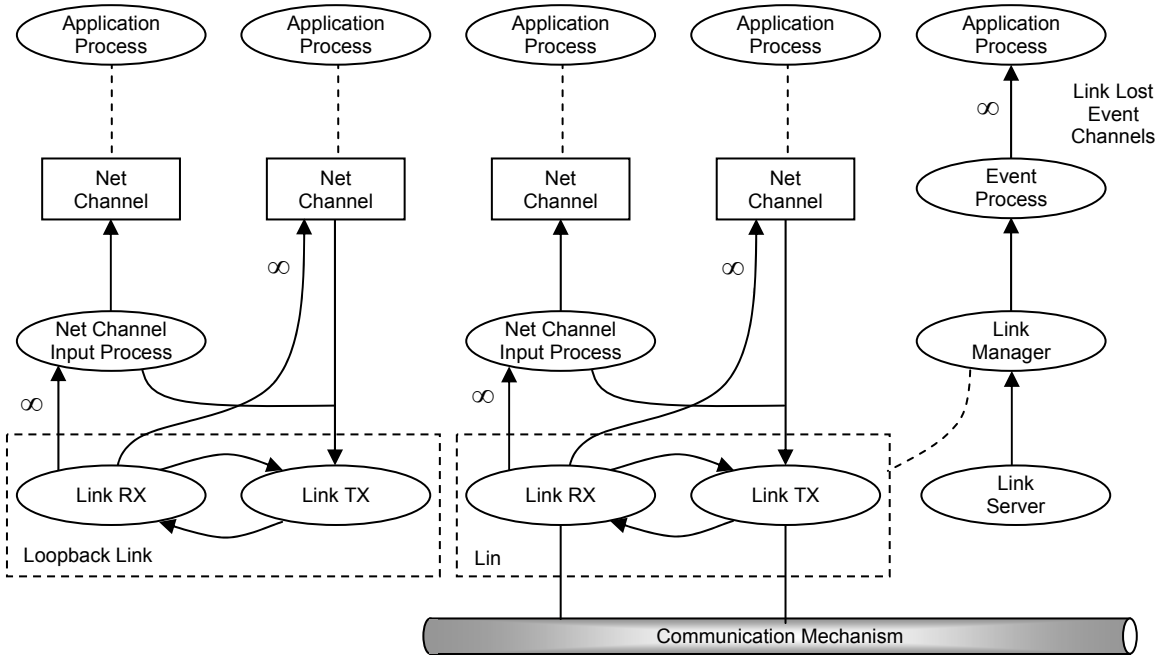

Figure 1: current JCSP network architecture.

The `Link` encapsulates the connection to another node within the system by sending and receiving messages via the underlying stream mechanism, which is dependent on the inter-node communication fabric. The `Link` process has two sub-processes: `LinkTX` which is responsible for sending messages, and `LinkRX` which is responsible for receiving messages. As a node may be connected to multiple remote nodes, there may be multiple pairs of these processes.

The other form of connection is the `LoopbackLink` which simulates a local connection. The `LoopbackLink` allows a channel output to connect to a channel input on the same node. Because JCSP allows for this eventuality, the `LoopbackLink` must always be in operation. There is only ever one `LoopbackLink` and corresponding TX and RX processes active on a node.

The `LinkServer` processes listen for incoming connections from other nodes, and start up a new `Link` process accordingly. With TCP/IP the `LinkServer` process encapsulates a normal server socket. In most cases, only one `LinkServer` is created; but if different communication mechanisms are in use or the node must listen on multiple interfaces, then multiple `LinkServer` processes may be active.

The `Link` processes are managed by a `LinkManager` process. `LinkServer` processes connect to the `LinkManager` process to communicate new connections. `Link` processes are also connected to the `LinkManager` to allow notification of a connection failure. An `EventProcess` is spawned by the `LinkManager`, which is used to

communicate link failures to the application level. An application process must create a new `LinkLostEventChannel` to allow this message to be received. The `EventProcess` is a sequential delta outputting upon the `LinkLostEventChannels` any `LinkLost` message it receives.

Channels similarly have a manger called `IndexManager`. This is not shown in Figure 1, but it contains connections to all networked channel constructs. Unlike `LinkManager`, this is a shared data object controlled via Java synchronized methods. Whenever a new channel is created, it is registered with the `IndexManager`. The `Link` processes use the `IndexManager` to access the channel ends during operation.

As mentioned, networked channels come in two forms: `NetChannelInput` and `NetChannelOutput`. An output end is connected directly to its corresponding `LinkTX` process. As network channels are Any2One, an input end may receive messages from any `LinkRX` process. The messages are not sent directly to the channel end, but are sent to a `NetChannelInputProcess` which then forwards the message onto the channel end. The channels from the `LinkRX` to the channel end / process are buffered with an infinite buffer, meaning that there is no risk of deadlock on the `LinkRX` process when it sends the message to a channel. The amount of buffering needed is bounded by the number of external processes trying to communicate to that particular network channel. It holds received, but not yet accepted, messages. That number cannot be pre-calculated, since it may, of course, change during run-time. However, it will always be finite!

3.1 Current Functionality

The basic operation during a send / receive operation occurs as follows:

1. An application calls `write` on the `NetChannelOutput`.
2. The `NetChannelOutput` wraps the sent object inside a message object, and writes this to the `LinkTX` process. The message object contains details on destination and source to allow delivery and subsequent acknowledgement. The `NetChannelOutput` then waits for an acknowledgement message from the `LinkRX`, blocking the writer (so as to maintain synchronisation semantics of CSP channels).
3. The `LinkTX` serializes the message onto the connection stream to the remote node.
4. The `LinkRX` on the remote node deserializes the message from the stream, retrieves the destination index, and requests the channel from the `IndexManager`.
5. To allow quick acknowledgement, the `LinkRX` attaches the channel to its partner `LinkTX` to the message. The message is then written to the `NetChannelInputProcess`.
6. The `NetChannelInputProcess` reads the message and writes the data part to the channel end. This is a blocking write, so until the receiving process is ready to read the message the `NetChannelInputProcess` waits. Once the write has completed, an acknowledgement message is written to the channel attached to the message during the previous step.
7. The `LinkTX` serializes the message onto the connection stream to the remote node.
8. The `LinkRX` on the remote node deserializes the message from the stream, retrieves the destination index, and requests the corresponding channel from the `IndexManager`.

9. As the message is an acknowledgement, the message is written directly to the channel end.
10. On receiving the acknowledgment message, the `NetChannelOutput` message can complete the write operation and release the writer.

These ten steps capture most of the functionality underlying the network architecture. Other conditions, such as link failure, message rejection, etc, are not covered here. With these steps in mind, we can move forward and critique the current implementation.

4. Critique of JCSP Networking

Our test bed consists of a PC communicating to a PDA device via a wireless network. The PC is a Pentium IV 3 GHz machine with 512 MB of RAM running Ubuntu Linux. The PDA is an HP iPaq 2210 with a 400 MHz processor and 64 MB of memory, shared between storage and applications. The operating system on the device is Windows Mobile 4.2, and thus provides a similar API to a standard Windows based desktop. The wireless network is an 802.11b network running at 11 Mbps. The PC is connected via a standard Ethernet interface to the router, and the PDA is connected via a wireless interface. Considering how small and resource restrictive components in ubiquitous computing may be, this test bed is fairly powerful. However, this setup allows us to discover limitations. Of particular note is the JVM running on the PDA. This is an IBM J9 JVM, and due to resource limitations can only create just under 400 simple threads with little internal stack. As every process in JCSP is handled by a thread, this allows us to examine JCSP networking in a very limited environment, not envisioned during original development.

We are interested in analyzing the resource usage and general performance of JCSP, and have therefore sent objects of various sizes and complexities via normal networked streams, buffered network streams, normal JCSP network channels and unacknowledged JCSP network channels. The buffered streams are required as JCSP buffers its own streams when used within a TCP/IP environment. The unacknowledged channels are a feature of JCSP networking and it was hoped that examination of these would permit understanding of the overheads of message sending. As we shall see, it has helped us discover another problem instead. As JCSP sets Nagle[1] 'off' for its TCP/IP connections, all the results presented also have Nagle deactivated.

As mentioned, different complexities and sizes of objects have been examined. By complexity, we refer to the number of aliased objects that exist within the sent object itself. Here we will be presenting `TestObject4` to demonstrate properties. `TestObject4` is the largest object we have used, in byte size, and is complex. It inherits from `TestObject`. The class definitions are:

```
public class TestObject implements Serializable {
   protected Integer[] ints;
   protected Double[] dbls;
   ...}

public class TestObject4 extends TestObject {
   private TestObject testObject;
   private Integer[] localInts;
   private Double[] localDbls;
```

[1] Nagle increases performance for small packet sizes by condensing numerous small messages into single packets.

```
  public TestObject4(int size) {
        ints = new Integer[size];
        dbls = new Double[size];
        localInts = new Integer[size];
        localDbls = new Double[size];
        for (int i = 0; i < size; i++) {
              ints[i] = localInts[i] = new Integer(i);
              dbls[i] = localDbls[i] = new Double(i);
        }
  }

  public static TestObject create(int size) {
        TestObject4 tObj1 = new TestObject4(size);
        TestObject4 tObj2 = new TestObject4(size);
        tObj1.setTest(tObj2);
        tObj2.setTest(tObj1);
        return tObj1;
  }
}
```

To create an instance of the object, `create (int size)` is used. A single `TestObject4` has four internal arrays (two for `Integer` and two for `Double`), with the internal objects within these arrays being aliased. `TestObject4` has a reference to another `TestObject4`, which in turn references the original object. Therefore, there are numerous aliases within the objects being sent. The tests use internal array sizes from 0 to 100.

To understand the complexity and size of `TestObject4`, we can use the following formulae. For the number of unique objects sent, relative to *n* (the size of the internal arrays) we have:

$$2 \cdot (\text{TestObject4 (1)} + \text{Inherited Array Objects (2)} + \text{Own Array Objects (2)} + 2 \cdot n)$$

The number of object references sent is greater than this value as the objects in the arrays declared in `TestObject` are sent as reference data. The total number of object references can be gained by multiplying *n* by 2 again:

$$2 \cdot (\text{TestObject4 (1)} + \text{Inherited Array Objects (2)} + \text{Own Array Objects (2)} + 2 \cdot 2 \cdot n)$$

Calculating the amount of data in bytes is more difficult, due to the message headers as described in Section 1.3. The simplest method is:

$(n = 0) \rightarrow 326 \text{ bytes}$
$(n = 1) \rightarrow 500 \text{ bytes}$
$(n > 1) \rightarrow 500 + ((n - 1) * 68) \text{ bytes}$

The increase of 68 bytes per increment in size of the message is due to the size of the object being sent. An `Integer` wraps 4 bytes, and `Double` 8 bytes. Two of each object type is created in total – one for each `TestObject4` – for a total of 32 bytes. Each object also takes up 4 bytes of reference information, and eight object references are created in total – four for the new objects and each new object is aliased once. This requires 36 bytes, which gives us 68 bytes in total.

The reason to use complex objects that are however small in size is threefold. Firstly, the platform used is restricted in performance, and thus small message sizes are the most

likely to be sent. As we are concerned with UC, it is the abstraction we are more concerned with, and it may be that numerous communications are occurring between these devices. This is unlike parallel computing approaches, where large blocks of data are processed to try and increase performance by having a processor spend most of its time processing as opposed to communicating. Tests have also been conducted using large byte arrays of data, but this does not allow capturing of the serialisation and message overhead we present here.

Secondly, we are trying to discover the cost of sending messages via JCSP, taking into account serialisation. Sending large objects is not the norm within Java if we consider other remote communication mechanisms such as Java Remote Method Invocation (RMI). Therefore, we hope to analyze the overheads of sending messages via JCSP in comparison to the underlying stream mechanism. A comparison with RMI is left for future work as RMI is not a standard feature on mobile devices.

Thirdly, the maximum size of the object we present is smaller than the buffer underlying the network streams within JCSP. With these experiments, we are hoping to avoid the operation of the buffer being automatically flushed due to filling. Any overhead associated with this operation can be captured during large block sending.

4.1 Resource Usage

Our first criticism of JCSP networking is the required resources to start a networked node. If we examine Figure 1, we can see that each connection to another node requires two processes; each input channel requires a process; and a `LinkServer`, a `LinkManager`, an `EventProcess`, and two processes for loopback are created at startup. For an initial unconnected node, with no input channels declared we have 6 threads created (including the main thread). Of these processes, the `LinkServer` is required to accept incoming connections.

When the node connects to a remote Channel Name Server (CNS) during initialisation, the number of threads required is 11 (including main). The connection to the CNS involves the two `Link` processes, a service process for connection to the CNS, a `NetChannelInputProcess` for the connection from the CNS to the service process, and when the first `NetChannelOutput` is created, a small process to handle link failures is spawned also. Only the two `Link` processes are required.

During a connection operation, five processes are created and subsequently destroyed to handle handshaking between the two nodes. The `Parallel` in standard JCSP is robust, and will try its best to manage all the used process threads in the system. However, many of these processes are spawned using the `ProcessManager` object; therefore the threads are not taken from the standard pool but are recreated every time, although it would be possible to modify `ProcessManager` to utilize the `Parallel` pool. The starting and subsequent destruction of threads may not be considered a serious strain on the system per se, but it may increase the active thread count beyond the system limit. `Links` may also be created to connect to a remote node already connected to, and the handshake process takes place to determine if the new `Link` should be kept. This requires the creation and subsequent destruction of temporary threads, which again may cause the thread count to increase beyond system capabilities.

The number of processes created as operations continue is also substantial. Each connection to a new node requires two further processes, and each new `NetChannelInput` requires a further process. It can be seen that it is not hard to reach the 400 thread limit within the PDA without inclusion of the application processes. As an example, being connected to five nodes, with ten input channels (two for each node) and the initial CNS connection will require a total of 31 processes.

The main reason for the heavy resource usage would appear to stem from the common CSP / **occam** philosophy of when in doubt, use a process. In Java this is sometimes an expensive approach to take, particularly when considering resource constrained devices. One of the main problems is the use of extra processes as managers, and processes for controlling the `NetChannelInput`. Channels should be lightweight, but a thread in Java is not lightweight so using a thread for a channel is wasteful. This approach is also dangerous in a dynamic architecture when, for example, the application process forgets to destroy the channel when it is finished with it, resulting in the process being lost and the resources not reclaimed. The garbage collector will not recover these as there is still an active thread holding the resources. As one of the goals of JCSP is to transparently allow channels to be either local or distributed, we cannot rely on a process actively destroying an unused channel if it has no knowledge of whether it is networked or not. Modifying the existing input channel so that it uses fewer resources is therefore a necessary goal.

4.2 Complexity

The next criticism we level at JCSP networking may be considered subjective. It concerns the internal complexity of the implementation. The basic premise of JCSP networking is trivial; there are two arrays of Any2One channels creating a crossbar between the channel ends and `Link` processes. One of the supposed properties of JCSP networking is the fact that the architecture is removed from the underlying communication mechanism, meaning that JCSP can be implemented over any guaranteed delivery mechanism. The argument is that if the correct addressing mechanism is used, then JCSP can operate around it. Although this statement is true, it is difficult to achieve, requiring understanding JCSP networking internals. Without the source code, it would be incredibly difficult for a custom communication mechanism to be used. If JCSP truly sat above the communication medium, then all that should be required are the necessary input and output streams, and an addressing mechanism.

As an example, the TCP/IP version of the `Link` process must implement numerous abstract methods from the `Link` class, including writing and reading of test objects, handshaking, waiting for `LinkLost` replies, and reading and writing of `Link` decisions (whether this `Link` should be used). There are many methods that need implementation for addressing, protocols, and a `NodeFactory` (used to initialize a JCSP `Node` during startup). Some of these require knowledge of objects such as `Profile` and `Specification` which are undocumented. There is also a reliance on the `org.jcsp.net.security` and `org.jcsp.net.settings` sub-packages. This is to name a few of the hurdles that must be overcome to allow JCSP to operate upon a new communication mechanism.

4.3 Message Cost

Object messages are an expensive method of transferring data. Any sent object must be wrapped within the object message before being sent to the other node, and acknowledgement messages are themselves objects. If we consider the amount of extra data sent using serialisation and as reflection is used during recreation, this leads to an overhead. The message types are defined within an object hierarchy, with specialized messages extending simpler ones. As inheritance information is also sent within a serialized object, this adds a further overhead.

For instance, a send message to another node requires a source and a destination value, which are two 64-bit values, but the size of the message object is 249 bytes without any

data inside it. An acknowledgment message is 205 bytes. This is a significant overhead considering that the information required is only 16 bytes (the source and destination).

There is also extra information sent within the message, such as a channel name for resolution by the receiving node if the destination index is not known, and a flag indicating if the message should be acknowledged. Name lookup puts an extra strain on the node as it must find the name in a table prior to message delivery. For named channels the CNS should really be used.

Having the messages wrapped up in objects also restricts JCSP interacting with other process based frameworks. Under the current implementation it would be impossible for JCSP to send a message to pony for example. This reflects badly on the use of JCSP in a ubiquitous computing environment, as we cannot expect all devices to be able to use a JVM. The nature of distributed systems also requires a great deal more platform heterogeneity, and currently JCSP does not offer this.

4.4 Objects Only

Following from the previous point is the inability of JCSP to send anything but objects between nodes. In principle this is not a problem if we consider JCSP in only the Java context, but it does again make it difficult to communicate with other platforms. It would also be useful to send raw data between nodes as required, which can be done in principle as a byte array in Java is considered an object, but there is again an overhead involved due to serialisation.

This limitation also means that primitive data must be wrapped in an object prior to sending. This brings a further overhead. The core JCSP packages implement a primitive integer channel to allow a slight increase in performance, and it should be reasonable that JCSP networking do this as well. To achieve this in the current implementation would require the channel to wrap the primitive in an object for sending, as the message object only uses an object as data to transfer.

4.5 Performance

We now present experimental data regarding the general performance of JCSP networking when sending messages. The results presented are the mean of a roundtrip of 60 objects of a given size, taken from the PDA. The results from the point of view of the PC are the same. These values are gathered by sending and receiving back an object of the given size, acquiring the time for performing this action 10 times, and this in turn is performed 10 times. Thus in total 100 objects are sent and received, but this is split into batches of 10 to allow a finer grained analysis. Finally, of these 10 batches, the largest and smallest two results are removed to smooth the data. Initially, the average of 100 send-receive operations was taken, but due to unexpected peaks and valleys in the results, a closer examination was taken. Although it has shown the peaks and valleys are not the result of individual runs, the real reason has yet to be determined.

Our first set of data highlights the efficiency gained by placing an 8 KB buffer on the stream when compared to an unbuffered stream. These are Java `BufferedInputStream` and `BufferedOutputStream` objects surrounding the streams provided by Java `Socket` objects. Java serialisation causes individual bytes to be written directly to the stream, which results in numerous small packets being sent in a TCP/IP environment. As JCSP networking places these 8 KB buffers on its streams, this allows us to see why initially JCSP performs better than standard networking on the mobile devices we are investigating. Some preliminary experiments using standard desktop machines appears to show that this advantage is lost as normal buffering and processor speed on desktop architectures

compensates for the lack of buffering. It can be deduced that the PDAs have little or no network buffering to increase performance, leading to the assumption that JCSP is more efficient than it actually is. The comparison of buffered and unbuffered streams is presented in Figure 2.

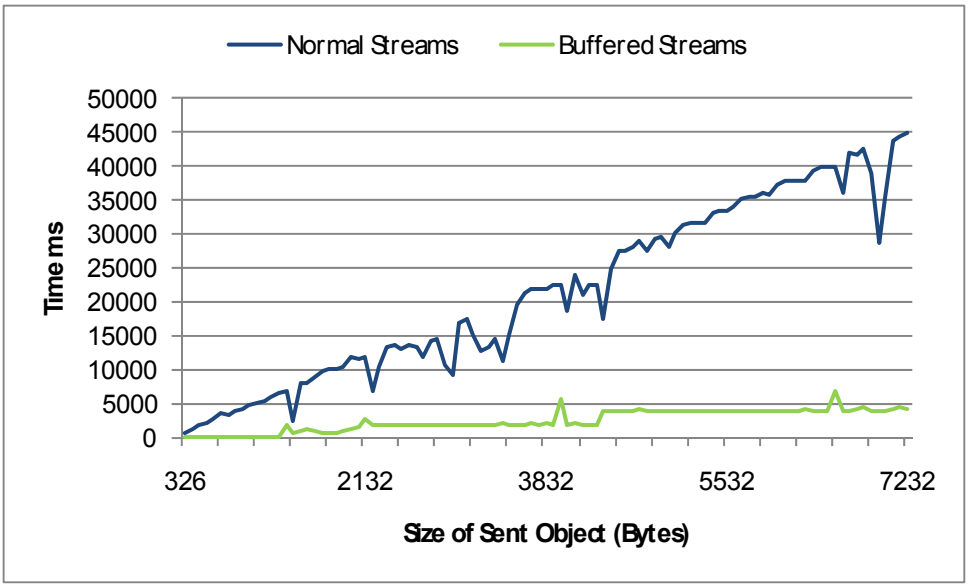

Figure 2: normal streams versus buffered btreams.

There are some interesting points to consider. As previously mentioned, there are valleys and peaks evident which are currently under investigation, initial signs pointing to the internal workings of the JVM or the PDA itself when dealing with certain sizes of data being the cause. The other interesting phenomenon is the steps in the buffered stream results. These steps reflect the extra packets of data sent as the Maximum Transmission Unit (MTU) of the network is reached. The MTU is the maximum single packet size that can be sent in one operation. The MTU for the wirelesos network is 2272 bytes and the MTU for the Ethernet connection is 1492 bytes as set by Linux. The buffer is greater than this value and therefore the data is split into separate packets during transmission.

Our next set of results illustrates the difference between networked channel operations and buffered stream operations. These are presented in Figure 3. The same step increase is apparent, although the networked channels are distinctly pushed left. This is due to the extra overhead of sending objects via JCSP as described in Section 4.3. It also appears that the first step in the results occurs at the MTU size for the Ethernet (1492) and the second occurs around twice the MTU for wireless packets (2272). The reason for this has not yet been ascertained. A packet is fragmented by the router when being transmitted from the PDA to the PC to allow the larger wireless packet to be sent as smaller packets on the Ethernet. There is no such constraint from the PC to the PDA as the wireless network can manage packets from the Ethernet. The PC should be capable of handling the reconstruction of fragmented packets without significant delays. Therefore, the steps should occur at twice the MTU of the Ethernet, especially as the PDA is capable of sending data far quicker than it can receive. This is shown in Figure 5.

We can also see the same peaks appearing in the JCSP networked channel results as the buffered stream results, strengthening our belief that it is not the implementation causing a problem.

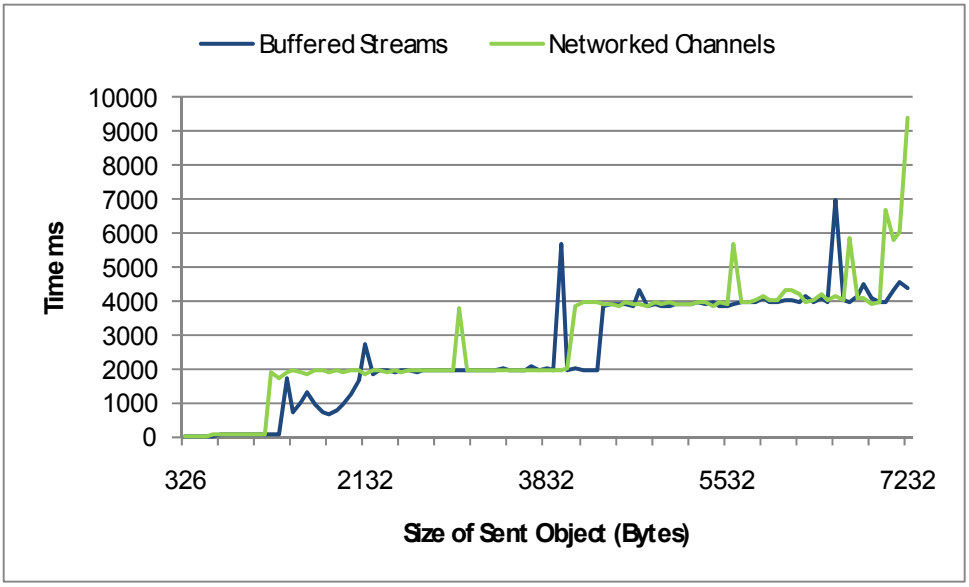

Figure 3: buffered streams versus networked channels.

These results allow us to show that JCSP does have an overhead within its communication mechanism. This overhead can have consequences, especially for resource constrained devices. As Figure 3 illustrates, an object that is sent via a channel may take two seconds on a roundtrip when the buffered stream equivalent has minimal time. This is due to sending extra data packets and if the overhead was reduced, the difference in performance could be compensated. The extra bytes sent for the message (249) and the Ethernet packet size (1492) imply that one sixth of a single packet is taken up by information beyond the sent data. Therefore there is a one in six chance that a message sent via JCSP will require an extra packet in a normal network. For a send-receive operation (as presented in Figure 3), this increases the time by two seconds within our test bed.

4.6 High Priority Links

Our next set of results illustrates a danger in the implementation of JCSP `Links`. These processes are given highest priority in JCSP to decrease latency by having the TX/RX processes start as soon as possible. The argument is that these processes may be blocked while trying to send or receive data if they are not given high enough priority. There is a converse to this argument. To illustrate the danger, we present the results of the PDA only receiving (and not sending back) messages from normal networked channels and unacknowledged network channels. These are given in Figure 4.

As this chart illustrates, sending unacknowledged messages takes more time than acknowledged ones. This should not be the case. First, there are fewer messages sent (no acknowledgements), and second the PC should be able to send messages faster than the PDA can read them, due to performance differences between the two machines and their network interfaces. What is happening here is that the PC is sending messages too fast for the PDA to cope. Data is appearing on the network before the PDA has time to process the previous message. The readings are taken from the application level process, and it is being superseded by the `LinkRX` process as it receives new messages. his results in the application process taking longer to receive messages as it must wait for the `LinkRX`. Underlying every network channel is an infinite buffer to receive messages upon to avoid a `LinkRX` process deadlocking. This leads to the `LinkRX` being capable of constantly writing to the channel if it has data ready, without the channel having time to respond.

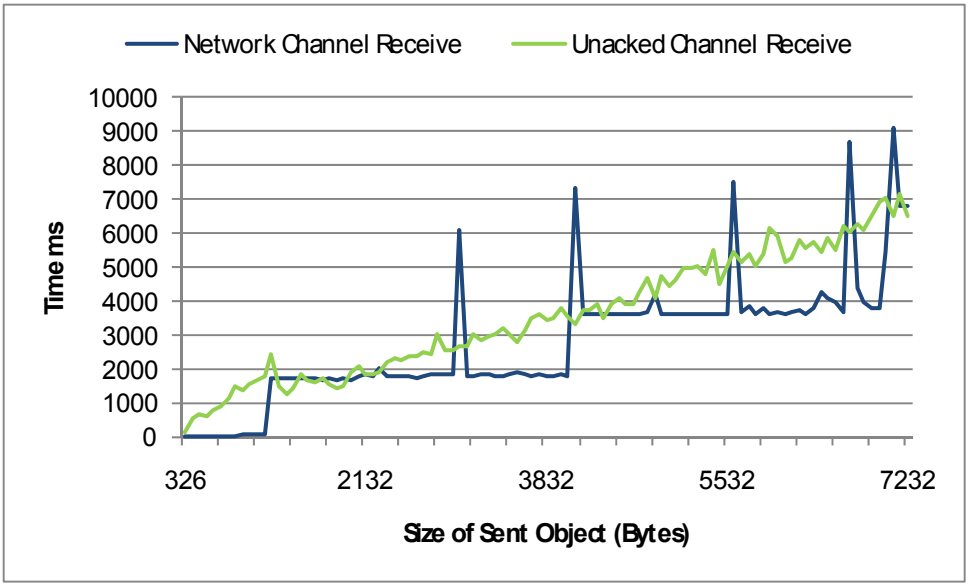

Figure 4: normal versus unacknowledged network channels receiving.

This may appear an unfair comment since unacknowledged channels should not be used in such a manner (they are used to avoid the Channel Name Server and networked connections from blocking).

The problem can also lie in channels which are buffered, have multiple incoming connections, and are accepting large packet sizes. This can lead to an application process waiting until data is received by the `LinkRX` processes. As the user has no control over the priority of the `Link` processes, they have no method to decide whether distributed I/O or application processes should be given highest priority.

A simple analogy of this is a producer-consumer program that operates with an infinitely buffered channel. If the sender is given higher priority than the receiver, then the receiver is theoretically starved as it cannot continue until the sending process has completed sending. Over time, the buffer in the channel grows and we are in danger of running out of memory. In practice this is not strictly true, as the receiver will be allowed to consume some of the messages. This may occur in JCSP over time using a standard infinitely buffered channel.

Buffered networked channels are also present in the current JCSP implementation. These are implemented by buffering the channel between the `NetChannelInputProcess` and `NetChannelInput`. Therefore the `NetChannelInputProcess` may not be blocked while writing a message to the `NetChannelInput`, depending on the buffer used within the channel. The standard JCSP buffers may be used within these channels, and thus there may in fact be two infinite buffers filling into one another.

The other interesting point that this graph illustrates is the repetition of the step function for the receiving results on the PDA. When comparing the results for communication from PC to PDA and from PDA to PC (Figure 5) we see that the greatest time is taken when the PDA is receiving data. Reducing this time would increase performance in our test bed considerably and is worth future investigation. The communication synchronizes, so the results should be the same if it were network capabilities restricting performance.

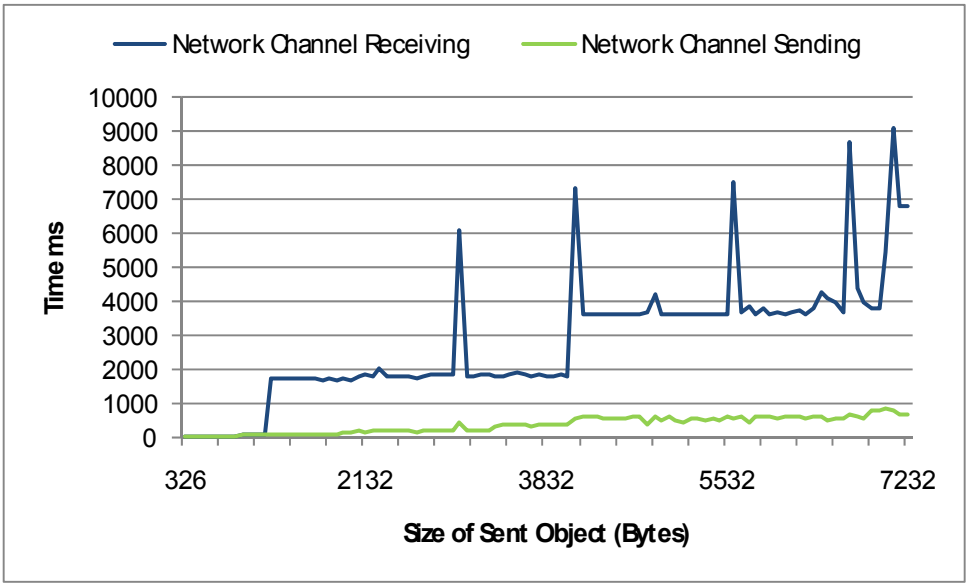

Figure 5: network channel sending and receiving on PDA.

4.7 Lack of Exceptions

The next limitation of JCSP is the lack of well documented exceptions being passed up to the application level processes. JCSP networking relies on I/O operations, and these can fail for reasons outside the control of JCSP. To combat this, exceptions such as `LinkLost` may be thrown, or a `LinkLostEventChannel` can be checked for any lost connections. These are not always caught. An example of such an operation is when an output end sends a message to the corresponding input end. If, prior to the acknowledgement being sent, the connection between the two nodes fails, the output end is not informed, and is left to hang waiting for the acknowledgement that will never come. As we do not have guarded output in JCSP networking we cannot recover from such an eventuality and must restart the system. A simple solution to overcome this problem is given in Section 5.

We may get an exception when something bad happens in the form of a `ChannelDataRejectedException`. This is a strange exception to be thrown for failed I/O operations. `RejectableChannels` are a deprecated construct within the core library and the reliance of JCSP networking on them should be removed. It would appear that the reason for having `RejectableChannels` was originally to handle I/O exceptions so that exceptions could be passed to the application level. As these I/O exceptions may still occur, a mechanism must be put in place to pass the exceptions onto the application process in a manner that allows a networked channel end to still appear as a normal channel end if required. This is described in Section 5.

4.8 Lack of Universal Protocol

Our final concern reflects on the issues raised in sections 4.3 and 4.4. JCSP utilizes objects as messages between distributed nodes. This is not a concern if we only wish JCSP to communicate with itself. However, we now have a numerous implementations of networked CSP inspired architectures across a great number of platforms. It is impossible for JCSP to communicate with pony or PyCSP in its current form without some form of Java object interpreter built into the respective frameworks. This is a limitation. When concerned with distributed systems, we should strive to allow inter-system communication whenever possible. It is reasonable for JCSP to communicate with pony, and vice-versa,

but this is not possible. A universal communication mechanism and protocol should be developed to allow these separate frameworks to communicate as much as possible. This is covered in Section 5.

5. Conclusions and Ongoing Work

We have shown that JCSP networking currently has a number of problems – especially when considering small power/memory devices – and our hope is to address these with a new implementation of JCSP networking. In summary, we have argued that:

- The architectural implementation leads to high resource overhead
- The architectural implementation is complex when compared to the basic premise of JCSP networking, making extension difficult
- Message packets are large in comparison to the amount of information actually sent within them
- The current implementation by default only allows serializable objects to be sent
- Performance of the basic communication mechanism is almost on par with the underlying stream, but message overheads have an effect
- The default high priority link is restrictive as some applications may require lower priority I/O
- Exception raising is not guaranteed
- There is no interoperability between frameworks due to JCSP relying on objects as transmitted messages

Our new implementation of JCSP networking aims to overcome these problems while also trying to bring the new architecture to the same level as the core for functionality.

5.1 A New Architecture for JCSP Networking

Our new architecture is based on the existing JCSP networking implementation, as well as taking inspiration from C++CSP Networked and pony. We have aimed to retain the existing interfaces whenever possible to allow existing users the same familiarity with the library.

The new architecture has the initial aim of reducing the resource overheads discovered in JCSP networking by removing unnecessary processes. To support this, we have removed `NetChannelInputProcess` and `LoopbackLink`, as well as converting the management processes into shared data objects.

Our new approach is based on a layered model. This allows functional separation and allows simple extension / modification. The underlying process model is still in effect, and the new architecture model is almost exactly the same. Figure 6 illustrates the new architectural model.

The key difference is how the components communicate together. Each layer only understands certain message types, thus promoting separation. As a layered approach is taken, messages only travel as far up or down the layered model as required, providing a further degree of separation. This allows simple additions and modifications in specific components to allow extension of the architecture. For example, networked barriers have been implemented by providing the same mechanisms in the Event Layer as there are for channels.

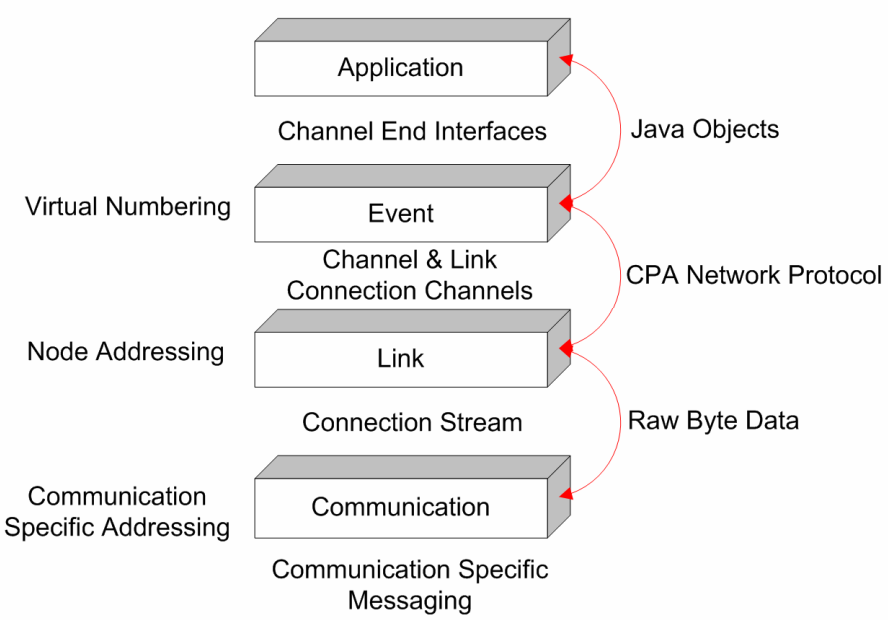

Figure 6: new JCSP architectural model.

5.1.1 Networked Barrier and AltingBarrier

The barrier operates on a client-server basis, with one JCSP node acting as a server for *n* client barriers. Each barrier end may have a number of processes synchronizing upon it, and for optimisation purposes it is only when all locally enrolled processes have synchronized does a client barrier end communicate with the server end. Once all client ends have synchronized, the server end releases them. Thus, the networked barrier operates in two phases; local synchronisation and then distributed synchronisation

We are also considering how to implement a networked multi-way synchronisation within the architecture, but this is far more difficult. The two phase approach used in the standard barrier cannot be reused for a direct networked implementation of `AltingBarrier` due to the implementation within the JCSP core. Here, a central control object is used, which ensures that only one process is actively alting on any `AltingBarrier` at any one time. This is irrespective of the number of `AltingBarriers` in the system. This controls the multi-way synchronisation in a manner that allows fast resolution of choice between multiple multi-way synchronizing events.

The problem with this controlled model is that it does not scale to multiple processors. If each process must wait to access this coordination object, then only one process is in operation at any time. This is fine in a single processor, concurrent system as only one process can only ever be in operation. With a multi-processor environment the problem is that all processors must wait while one accesses the coordination object. This is a worst case scenario, but does highlight the problem faced. A distributed event based system faces the same problem. However, we currently believe the two phase approach used for networked barriers is the most likely approach for efficiency reasons when dealing with a distributed multi-way synchronisation. A possible approach is to use a process within the networked `AltingBarrier` client end to control the synchronisation and communicate with the declaring `AltingBarrier` server end. This approach should allow a single networked `AltingBarrier` to exist on a single node. However, the goal would be to allow multiple networked `AltingBarriers` on a node.

Another key feature is the use of a communication protocol independent of Java or any other platform. Instead of relying on Java objects, data primitives are used.

5.1.2 A Universal Protocol for CPA Based Networking

The aim of our new protocol is to promote communication independent from the data being sent. Through this we hope to achieve a standard mechanism that can be exploited across the various CSP based network architectures. It is perfectly reasonable to expect JCSP to communicate with KRoC, and thus we have aimed at a simple protocol that is easily portable to other platforms. The standard message types can be well defined, and thus far we have encountered only message types that require three values: a *type*, and two *attributes*. These can be expressed using a byte and two 32 bit signed integers. We are also removing the need for object messages which reduces the message overhead. At present, this reduces the 249 byte message header to 9 or 13 bytes.

This does not take into account data passed within the message itself, and this is considered a special case. We define message types using values, therefore the message type can also be used to determine whether or not the message has a data segment. If it does, the number of bytes can be sent as a 32 bit integer, and then the data itself can be transferred. The new key feature is that channels are now responsible for converting objects to and from bytes, and the messages themselves must only contain byte data. The conversion method can be specified by the JCSP user, thus providing data independence as there is no longer a reliance on Java serialisation. For example, a JCSP system could send a message to a pony system by utilising a converter that implemented strict **occam** rules for data structures. Schweigler's [6] work in this area provides a strong basis to build upon.

5.1.3 Channel Mobility

We are also hoping to build channel mobility directly into the protocol to allow mobility of channels between platforms. Unfortunately, at present this cannot be fully accomplished due to conflicting approaches for mobility proposed for JCSP [11] and pony [6]. Both of these approaches have advantages and disadvantages, and a coming together is required for mobility to be implemented directly into the protocol.

5.1.4 Other Features

One shortcoming overcome in the new model is the exception handling mechanism, as specified in Section 4.7. There is now a unique exception that can be thrown by a networked JCSP system; the `JCSPNetworkException`. This is a silent exception, in that it does not need to be caught explicitly by the JCSP user, thus permitting channel operations to throw the exception but not break the existing core channel interfaces.

We have also implemented a solution to the deadlock caused by an output channel waiting indefinitely for an acknowledgement from a broken connection. `NetChannelOutput` objects now register with their respective `Link` components on creation, and unregister on destruction / failure. This allows a `Link` to send a message to the `NetChannelOutput` on failure. As this message is written to the same channel as an acknowledgement would be expected the `NetChannelOutput` can receive this message and act accordingly. As this message may also be sent prior to an initial write, the channel can be checked at the start of a write operation, avoiding unnecessary attempts to write to a dead `Link`.

5.1.5 Verified Model

Our design has been approached with model checking in mind. We have performed some preliminary investigation with Spin [23], with promising results thus far. We do have a number of properties we still wish to examine. The reason to use Spin as opposed to FDR is due to the usage of mobile channels within the JCSP architecture, and Spin allows channel mobility explicitly within its models. The model built within Spin can be directly composed into Java due to the core of JCSP being in place.

5.1.6 Ongoing Work

There is still work to do on the new implementation of JCSP networking. In particular, if the defined protocol is to be considered as a method to allow intercommunication between different platforms, then further investigation must be undertaken for other common / expected messages within these frameworks. As an example, the protocol currently does not implement any notion of claiming a channel end, although this is used within pony for shared channel ends.

Connections are currently not implemented in the new architecture. It is possible to implement connections using normal networked channels, but this requires building a connection message protocol that will be sent via the communication protocol. A more practical approach is to implement these message types directly into the communication protocol, and then develop management and component ends within the Event Layer.

5.2 Future Work

We have highlighted a number of other pieces of future work beyond the new implementation of JCSP networking. Work on enhancing the serialisation capabilities of Java to accommodate JCSP will likely lead to an increase in performance for both small factor and desktop applications. We also hope to perform comparisons with RMI, taking into account simplicity, code mobility and performance. With networked connections, it should be possible to create remote interfaces that are externally similar to RMI. Finally, for our own experimental test bed, the possibility of increasing performance for the PDA receiving data is interesting.

5.3 Conclusions

We have shown that JCSP networking has a significant number of problems which lead to certain impracticalities when considering JCSP in small factor devices. Particularly, we have shown that that there are overheads due to excess process creation and destruction, as well as overheads for message transfer. We have also illustrated some dangers and argued on the complexity of the implementation.

Finally, we have discussed a new approach for JCSP networking which will lead to a more ubiquitous approach for CSP networking as a whole. We hope that this approach can be replicated across the various CSP based frameworks to allow stronger integration, allowing simpler exploitation of multiple platforms.

References

[1] P. H. Welch, "Process Oriented Design for Java: Concurrency for All," in H. R. Arabnia (Ed.), *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '2000)Volume 1*, pp. 51-57, CSREA Press, 2000.

[2] D. Lea, "Section 4.5: Active Objects," in *Concurrent Programming in Java: Second Edition*, Boston: Addison-Wesley, 2000, pp. 367-376.
[3] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, Inc., 1985.
[4] P. H. Welch, N. Brown, J. Moores, K. Chalmers, and B. H. C. Sputh, "Integrating and Extending JCSP," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), *Communicating Process Architectures 2007*, pp. 349-370, IOS Press, Amsterdam, 2007.
[5] P. H. Welch, J. R. Aldous, and J. Foster, "CSP Networking for Java (JCSP.net)," in P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra (Eds.), *International Conference Computational Science — ICCS 2002*, *Lecture Notes in Computer Science 2330*, pp. 695-708, Springer Berlin / Heidelberg, 2002.
[6] M. Schweigler and A. T. Sampson, "pony - The occam-π Network Environment," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), *Communicating Process Architectures 2006*, pp. 77-108, IOS Press, Amsterdam, 2006.
[7] N. Brown, "C++CSP Networked," in I. East, J. Martin, P. H. Welch, D. Duce, and M. Green (Eds.), *Communicating Process Architectures 2004*, pp. 185-200, IOS Press, Amsterdam, 2004.
[8] J. M. Bjørndalen, B. Vinter, and O. Anshus, "PyCSP - Communicating Sequential Processes for Python," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), *Communicating Process Architectures 2007*, pp. 229-248, IOS Press, Amsterdam, 2007.
[9] Inmos Limited, "The T9000 Transputer Instruction Set Manual," SGS-Thompson Microelectronics 1993.
[10] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Transactions on Software Engineering,* 24(5), pp. 342-361, 1998.
[11] 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.
[12] N. Brown and P. H. Welch, "An Introduction to the Kent C++CSP Library," in J. F. Broenink and G. H. Hilderink (Eds.), *Communicating Process Architectures 2003*, pp. 139-156, IOS Press, Amsterdam, 2003.
[13] N. Brown, "C++CSP2: A Many-to-Many Threading Model for Multicore Architectures," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), *Communicating Process Architectures 2007*, pp. 183-205, IOS Press, Amsterdam, 2007.
[14] A. A. Lehmberg and M. Olsen, "An Introduction to CSP.NET," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), *Communicating Process Architectures 2006*, pp. 13-30, IOS Press, Amsterdam, 2006.
[15] N. C. Schaller, S. W. Marshall, and Y.-F. Cho, "A Comparison of High Performance, Parallel Computing Java Packages," in J. F. Broenink and G. H. Hilderink (Eds.), *Communicating Process Architectures 2003*, pp. 1-16, IOS Press, Amsterdam, 2003.
[16] B. Vinter and P. H. Welch, "Cluster Computing and JCSP Networking," in J. Pascoe, P. H. Welch, R. Loader, and V. Sunderam (Eds.), *Communicating Process Architectures 2002*, pp. 203-222, IOS Press, Amsterdam, 2002.
[17] S. Kumar and G. S. Stiles, "A JCSP.net Implementation of a Massively Multiplayer Online Game," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), *Communicating Process Architectures 2006*, pp. 135-149, IOS Press, Amsterdam, 2006.
[18] M. Weiser, "The Computer for the 21st Century," *Scientific American,* September, 1991.
[19] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes, I," *Information and Computation,* 100(1), pp. 1-40, 1992.
[20] R. Milner, "Ubiquitous Computing: Shall we Understand It?," *The Computer Journal,* 49(4), pp. 383-389, 2006.
[21] C. G. Ritson and P. H. Welch, "A Process-Oriented Architecture for Complex System Modelling," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), *Communicating Process Architectures 2007*, pp. 249-266, IOS Press, Amsterdam, 2007.
[22] C. L. Jacobsen and M. C. Jadud, "The Transterpreter: A Transputer Interpreter," in I. East, D. Duce, M. Green, J. Martin, and P. H. Welch (Eds.), *Communicating Process Architectures 2004*, pp. 99-107, IOS Press, Amsterdam, 2004.
[23] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, 2003.