Communicating Process Architectures 2007
Alistair McEwan, Steve Schneider, Wilson Ifill and Peter Welch (Eds.)
IOS Press, 2007

Mobility in JCSP: New Mobile Channel and Mobile Process Models

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. The original package developed for network mobility in JCSP, although useful, revealed some limitations in the underlying models permitting code mobility and channel migration. In this paper, we discuss these limitations, as well as describe the new models developed to overcome them. The models are based on two different approaches to mobility in other fields, mobile agents and Mobile IP, providing strong underlying constructs to build upon, permitting process and channel mobility in networked JCSP systems in a manner that is transparent to the user.

Keywords. JCSP Network Edition, mobile processes, mobile channels.

Introduction

The network package of JCSP [1, 2] provides a framework for distributed parallel systems development, as well a level of networked mobility of processes and channels. Originally this was complex to implement, although it was rectified by the original `jcsp.mobile` package [3]. Recent extended usage of these features has uncovered some limitations, as well as previously unknown advantages, to the original application models. The aim of this article is to examine new models for both channel and process functionality in JCSP, which have been implemented in the newest version of the `jcsp.mobile` package, as well as current plans to improve the models further.

The rest of this article is structured as follows. In Section 1 we introduce the different types of mobility, before describing process mobility further in Section 2. In Section 3 we discuss how to achieve process mobility in Java, focusing on the class loading mechanism. Section 4 discusses the original class loading model of JCSP, and Section 5 presents solutions to the problems found in the original structure. In Section 6 we provide a brief evaluation of the model developed.

Section 7 discusses channel mobility, and in particular the Mobile IP model used to define the new architecture for channel mobility in JCSP, with Section 8 presenting this new model. In Section 9, we present a more robust future implementation for channel mobility that requires some modifications to the existing JCSP Network framework, and in Section 10 we evaluate our model. Section 11 and 12 presents future work and conclusions.

1. Mobility

We are concerned with two separate types of mobility – that of channel mobility and process mobility. These can both be split further between the general headings of localized mobility (single node) and distributed mobility (between nodes). Table 1 provides a

viewpoint of the complexity of these functional challenges. Also note that here process mobility is split between simple, single level processes, and complex, layered processes. The latter type of process is more difficult to migrate, and an implemented approach in JCSP has yet to be undertaken, although discussion concerning this shall be given in Section 11.

Table 1: Application Complexity of Mobility

Mobile Unit	Local Mobility	Distributed Mobility
Channel Mobility	Simple	Difficult
Simple Process Mobility	Simple	Moderate
Complex Process Mobility	Simple	Very Difficult

The argument presented in this table is that implementing mobility at a local level is simple. Although requiring additions to occam, leading to occam-π [4, 5], Java's nature meant this was possible in JCSP from conception.

These types of mobility (that we shall refer to as logical mobility [6]) allow us a new approach to developing systems, although a cost is incurred due to the level of abstraction we are dealing with. The cost the implemented models in JCSP inflict has yet to be measured. It is apparent in the pony framework of occam-π [7, 8] that overheads do exist when dealing with logical mobility at the network level. Therefore, we approached the design of these new models by examining existing applications in other fields, particularly mobile agents [9] and Mobile IP [10]. First of all we examine the concept of process mobility in the light of mobile agent approaches.

2. Process Mobility

To change the location that a program is executing requires various features, the most commonly cited being a method of code mobility [11]. This is exploited most of all in the field of mobile agents (MA). In fact, we make the argument that a MA is a specialized form of mobile process, an argument backed by a number of descriptions for agents, MAs, and their foundations.

2.1 Mobile Agents

Agents have their roots in the actor model, which are self contained, interactive, concurrently executing objects, having internal state and that respond to messages from other agents [12]. This description is similar to that of a CSP process – a concurrently executing entity, with internal state, which communicates with other processes using channels (message passing). Or another description is that an *"agent denotes something that produces or is capable of producing an effect"* [13]. Again, this is arguably similar to that of a process, which produces an effect when it communicates via a channel.

MAs are equally compelling when thinking of process mobility. Any process that can migrate to multiple hosts is a MA [13], although this may be a very broad claim, illustrated by the argument that *"the concept of mobile agent supports 'process mobility'"* from the same article. This appears to imply that mobile processes are MAs, although our argument is that this is not the case, the opposite being true.

MAs are also described as being autonomous (having their own thread of control) and reactive (receive and react to messages) [14], each of which enforce the belief that MAs and mobile processes have strong ties. Where they do differ is that process mobility is

more akin to adding functionality to an existing node, enabling it to behave differently. MAs are designed to perform tasks for us, although mobile processes can be enabled to do this also. Therefore our argument that MAs are indeed mobile processes is made.

As mentioned previously, MAs exploit a feature known as code mobility [15], and it is this area we examine next.

2.2 Mobile Code

Six paradigms exist for code mobility [11]:

- *Client-server* – client executes code on the server.
- *Remote evaluation* – remote node downloads code then executes it.
- *Code on demand* – clients download code as required.
- *Process migration* – processes move from one node to another.
- *Mobile agents* – programs move based on their own logic.
- *Active networks* – packets reprogram the network infrastructure.

Each of these techniques is aimed at solving a different functional problem in distributed systems. For example, client-server code mobility involves moving a data file to the node where the program resides, executing the program and transferring the results back to the user. This allows a client to exploit functionality available on the server. MAs use an opposite initial interaction; the program moves to where the data resides, and returns to the user's node with the results.

The term process migration as given does not fully incorporate our viewpoint on process mobility. The distinction between MAs and process migration lies in where the decision to migrate takes place. MAs are autonomous, and make their own decision to migrate. In process migration, the underlying system makes the decision. In a mobile process environment, either the system or the process may make the decision to migrate.

Efficiency of each technique can be determined by the size of the mobile parts, the available bandwidth between each node, and the resources available at each node. A large data file encourages the MA paradigm, whereas a large program encourages either the client-server or remote evaluation paradigm. These are concerns when considering what method of code mobility to approach, and the distinction between data state and execution state in the size of a MA shall be covered in Section 2.4.

2.2.1 Requirements for Code Mobility

There are four fundamental requirements to support code mobility [16]:

- *Concurrency support* – entities run in tandem with currently running processes.
- *Serialization* – the functionality to be converted into data for transmission.
- *Code loading* – the ability to load code during execution.
- *Reflection* – the ability to determine the properties of a logical element.

The support for concurrency is of prime importance, and most modern development frameworks have some support for a thread based model. Threads are not the safest approach however [17], and provide us with little structure.

Serialization of logical components is also common in object-orientated platforms. A point to consider is that not all objects are serializable, and may rely on resources specific to a single node. This is a consideration when sending an object to a remote node, and how to access a required unmovable resource after migration is a problem. Mobile channels linked to the fixed resource can possibly overcome this.

Finally, code loading and reflection are common in Just-In-Time (JIT) compiled and script based platforms such as Java and the .NET Framework, and this is important to mobile code. If a system cannot load code during runtime then mobile code is not possible, unless a scripting approach is used, which is slow and sometimes impractical.

2.2.2 Reasons to Use Code Mobility

The ability to move code between computational elements promises to increase the flexibility, scalability and reliability of systems [11]. Besides these possibilities, a number of actual advantages are apparent [11, 18]. Firstly, code mobility helps to automatically reconfigure software without any physical medium or manual intervention. Secondly, thanks to moving code from a stressed machine to an idle one, code mobility can participate in load balancing for distributed systems. Finally, code mobility compensates for hardware failure, increases the opportunity for parallelism in the application, and enhances the efficiency of Remote Procedure Calls (RPC).

Other reasons to utilise a code mobility system [6] include possible reduction in bandwidth consumption, support for disconnected operations and protocol encapsulation. Disconnected operation means we do not have to rely on fixed resources on each to node to operate, allowing the logical elements to move around in a transparent manner. Protocol encapsulation involves data and the method to interpret it being sent together.

However, all these considerations must be weighed against certain attributes such as server utilisation, bandwidth usage and object size [19]. It has been noted [20] that mobile code systems are not always a better solution than traditional methods, especially if the transferred code is larger than the bandwidth saved by not sending the data direct.

2.3 Other Requirements for Mobile Agent Systems

A number of further requirements can be considered when thinking of MA systems. The list below summarizes these [13]:

- *Heterogeneity* – MAs must have the ability to move between various hardware platforms. Therefore a cross platform system is needed, and one of the key reasons why Java has a strong representation in MA frameworks [6, 12].
- *Performance* – this is a very open ended term. Performance in a mobile agent system may refer to network transference or system resource consumption. What is clear is that a definite performance benefit needs to be apparent to justify the usage of a MA system.
- *Security* – as a MA is moving code into systems for execution, certain guarantees need to be in place, or security restrictions enforced to stop malicious agents affecting a system.
- *Stand alone execution* – MAs must be able to operate on their own, without definite resources at the receiving node.
- *Independent compilation* – the MA should not require definite linkages to other components that may not be available on receiving nodes. The agent should be logically distinct in the system.

2.4 Types of Migration

Mobility of code and agent migration can be split into two groups – strong and weak mobility [15]. Weak code mobility provides data and code transference, but no state information pertaining to it. Strong code mobility transfers state also, and may be

undertaken in a manner that is transparent to any communicating objects. State when referred to in this context includes execution state, and although desirable, Java lacks the ability to accomplish this. Therefore, in JCSP, only weak migration of logical elements is possible. The viability of strong migration has been questioned [11].

State itself can be split into two parts – data and execution – and it is the incorporation of execution state that dictates the type of migration. To clarify, strong mobility allows the resuming of a process directly after the migration instruction without any extra logic on the part of the developer [6]. In weak migration, the developer must create their own method of restarting the process after migration, which adds to the overall complexity of the process itself [13].

The case against strong migration is the amount of information that needs to be transferred. Both execution and data state must be transferred in this and this is time consuming and expensive.

Another consideration, in respect to how code is sent between nodes, is that of disconnected design. When migration occurs, all required code should also be sent at the same time [6]. This reflects on approaches using code repositories and the ability to accomplish this is questionable. For example, consider a mobile process that arrives at a node and then itself requires the receipt of another mobile process from its sending node. The ability to determine all code that may be required by a single mobile process is difficult, even if possible in all situations

Another viewpoint on migration is that of passive and active migration [18]. These terms are similar to those of weak and strong mobility. The difference is the amount of state transferred during the migration, with passive migration involving no state transference whatsoever. This description refers to the sending of data objects or code libraries. Active migration involves sending independently running modules.

The granularity of the code migrated is also a point to consider. Code can be thought of being either pushed to the receiving node all at once, or pulled by the receiving node as required [20]. The size of the sent component also varies depending on the type of architecture used. The approach to code mobility in Java is fine, sending only single classes at a time. A coarser grained approach is possible by utilizing JAR files. An even finer grained approach [20] has been suggested, which sends code on a per method basis instead of a per class one. The need for this in well designed software is questionable, and justification is based on unnecessary code loading occurring within Java. This has never been observed using either the original or new models for logical mobility in JCSP.

2.5 Migration Steps

There are six distinct steps involved during code migration [18]. Firstly, a decision to migrate must be made, either by the migrating unit itself or externally to it. The next step is the pre-emption of the application to avoid leaving both the application and the migrating component in an inconsistent state. The state of the component is then saved before it is sent to the next node. Any residual dependencies must then be taken into consideration, and dealt with accordingly. Finally the execution is continued when the migrating unit arrives at its new destination.

These steps represent a MA system more than just a code mobility system, and there are some issues in the form of communication used [21]. MAs use a mailbox communication system that does not guarantee instantaneous message delivery and this problem reflect upon the model for mobile channels which will be discussed in Section 7.

3. Achieving Process Mobility in Java

As was previously discussed, a key feature to achieve some form of process mobility is the usage of code mobility in a form similar to that of MAs. JCSP can take advantage of Java's code mobility mechanisms, and therefore we provide a discussion on code mobility in Java.

3.1 Java's Code Loading Model

Two objects are utilised in Java class loading – `ClassLoader` and `ObjectStream`. `ClassLoader` is responsible for loading classes for an application, and `ObjectStream` is used to serialize and deserialize objects.

Classes in Java are usually loaded by a single `ClassLoader` object – the `SystemClassLoader`. This object is responsible for loading class data for classes available on the local file system. It is also possible to develop custom `ClassLoaders` using the `ClassLoader` object as a base class.

Within the `ClassLoader` object itself, four methods are of note: `LoadClass`, `FindClass`, `DefineClass` and `ResolveClass`. Of these four methods, only `FindClass` needs to be overwritten.

3.1.1 ClassLoader Methods

The `LoadClass` method is called by any object that requires class data for an object that it uses. The `ClassLoader` being used by the object is determined by its class; whichever `ClassLoader` initially loaded the class is responsible for loading class data for that object. Consider Figure 1.

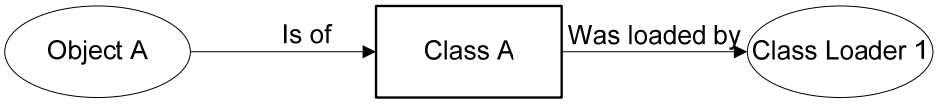

Figure 1: ClassLoader Used

If `Object` A is of `Class` A, which was loaded by `ClassLoader` 1, then any requests to load class data made by `Object` A will be carried out by `ClassLoader` 1.

`LoadClass` is called using the binary name of the class as a parameter and a `Class` object is returned. The `LoadClass` method goes through up to four stages when it is called:

1. Check if the class has already been loaded.
2. Call `FindClass` on the parent `ClassLoader` (usually the `SystemClassLoader`).
3. Call `FindClass` relevant to this `ClassLoader`.
4. Resolve (link) the `Class` if need be.

`FindClass` is used to find the data for a class from a given name. As the method used varies, the technique to find the class differs widely between implementations. What this method must do is return the `Class` of the given name and the simplest way to do this is to call `DefineClass` on an array of bytes representing the class.

3.1.2 ObjectStreams

`ObjectStreams` operate like other streams in Java. As network communications are also represented by streams, it is possible to create a object transference mechanism between nodes by plugging these into `ObjectOutputStreams` and `ObjectInputStreams`.

`ObjectInputStream` must be able to ascertain the class of the sent object and load the class data if necessary. This is performed by the `ResolveClass` method within the `ObjectInputStream`. With the `ObjectInputStream` supplied with the Java platform, the `ResolveClass` method loads the local class of the object sent using the `SystemClassLoader`. To allow a customised approach an `ObjectInputStream` is necessary that utilises a custom `ClassLoader`. The `ResolveClass` method of the new `ObjectInputStream` can then invoke the `LoadClass` method of the relevant `ClassLoader`.

With a basic description of the underlying operation of Java class loading presented, we can move on to an analysis of the operation of the original class loading mechanism within JCSP.

4. Original Class Loading System in JCSP

The original class loading mechanism for JCSP was part of the `dynamic` package provided in the JCSP.*net*. The mechanism is part of the `DynamicClassLoadingService`, which incorporates two filters, one for the serilization of objects deserialization.

4.1 Structure

Figure 2 illustrates the structure of the original `DynamicClassLoadingService` in JCSP.

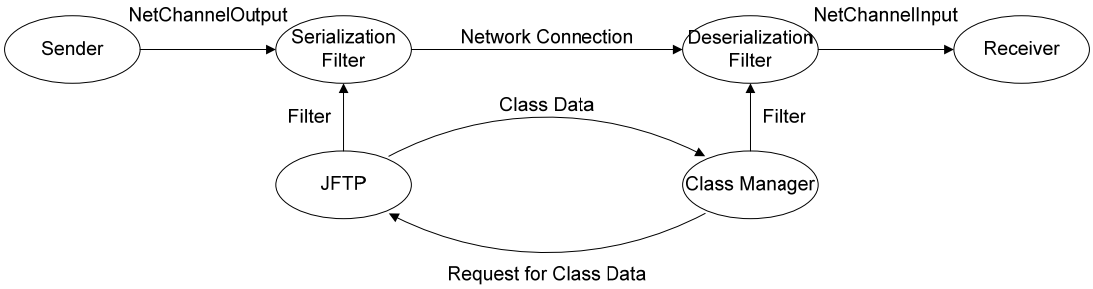

Figure 2: Original Dynamic Class Loading System

Two processes are utilised in the class loading service, `JFTP` and `ClassManager`. `JFTP` is responsible for processing requests for class data, whereas `ClassManager` requests, stores and loads class data.

4.2 Disadvantages and Problems

There are still limitations in the original `ClassLoader` itself. The most apparent is that class data requests do not propagate back through the system. This can lead to problems when trying to send an object that will traverse to another node before being utilised fully at a node on its path. For example consider the following:

```
01      public class MyClass {
02        public void run() {
03          MySecondClass obj = new MySecondClass();
04        }
05      }
```

Also consider a simple system consisting of three nodes, A, B and C. `Node A` creates an instance of `MyClass` and sends it to `Node B`. `Node B` reads in the object and then sends it onto `Node C` without calling the `run` method. `Node C` reads in the object and then invokes the `run` method.

Examining the interactions within the DynamicClassLoadingServices of these nodes we find a problem. When Node A sends the MyClass object to Node B, a request for the class data for MyClass is made from Node B to Node A. Node B now has the MyClass class file. When Node B sends the object to Node C, a request is made for the MyClass class data from Node C to Node B. When Node C invokes the run method of the MyClass object, it needs to discover what MySecondClass is. At this point it sends a request to Node B for the class data, which Node B does not have. An exception is thrown at Node C and our system fails.

The second limitation is poor handling of object properties. Consider the following:

```
06      public class MyClass {
07        private MySecondClass temp;
08      }
```

Both the MyClass and MySecondClass class representations should be loaded in one operation whenever an instance of MyClass is sent from one node to another. However, due to the method of interaction between the JFTP and ClassManager, only the MyClass data is requested and our system again fails.

The third limitation is due to the simplistic nature of the implementation of the DynamicClassLoadingService. As locally stored class data that has been received from another node is stored using the class name as a key, no two classes of the same name can be sent from two different nodes. When considering larger systems, it could occur that two classes use the same name, thereby creating an exception in the system.

One final 'fault' in the system was the handling of classes stored inside compressed JAR files. The comments inside the source code of JCSP point out that the existing method of reading in class data could not handle compressed archives. This is a trivial problem to solve, ensuring that we continue to read from the necessary stream until the number of bytes read in is the same as the number of bytes in the class file.

5. New Class Loader Model

To overcome the limitations of the current class loading mechanism, a new class loader has been developed for the jcsp.mobile package. This new model is presented as more robust, utilising a simpler method to retrieve class data, creating namespaces based on the node the class originally came from, and providing a class loading firewall as a first line of defense against malicious code attacks. The first area of development involves solving the problem of uniquely identifying classes with the same name.

5.1 Uniquely Identifying Classes

A simple solution to this problem is to link the class data with the node it originally came from. This can be achieved within JCSP by using the NodeID of the originating node that the object came from. This provides a method of creating class namespaces on nodes relevant to where the class data originated from. Consider Figure 3 for example.

Node A has sent an object to Node B and a class loading connection has been set up between them. This interaction has created a Node A class namespace within Node B. The same interaction has also occurred between Node B and Node C, creating a Node A class namespace at Node C also. Within these two separate namespaces it is quite possible to have two classes of the same name as each of the namespaces are distinct from the other.

5.2 New Class Loader Structure

The most important aspect of the new implementation is the inclusion of a method that allows class requests to propagate back through the series of nodes to the original. Essentially, the new class loader process combines both the `JFTP` process and `ClassManager` process into a single process.

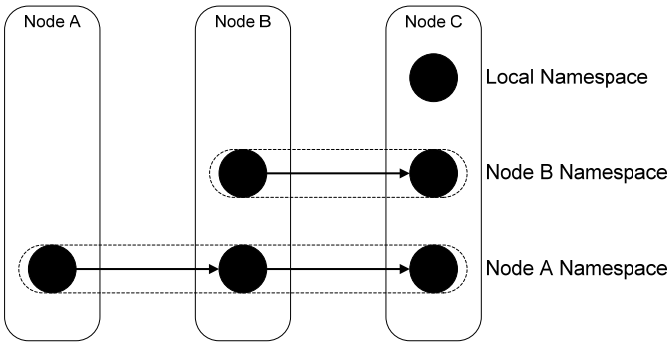

Figure 3: Node Namespaces

5.3 New Class Loader Operation

When considering the operation of the `ClassLoader`, there are four different usage scenarios that need to be examined. These scenarios are the interactions of single hop class loading between nodes for unknown internal classes and for classes that create unknown classes during their operation, and also the same two usages in a multi-hop interaction between several nodes.

5.3.1 Single Hop Operation

The first usage scenario has an object being sent from the Sender process to the Receiver process, and the sent object then creating an instance of a previously unknown class at the Receiver process's node. This is illustrated in code lines 1 to 5.

Here, the Sender first creates a new `MyClass` object and sends it to the Receiver. When the Receiver process reads the data it tries to recreate the original object using the customised `ObjectStream`. When `FindClass` is invoked, the system discovers that it cannot load the class locally and sends a request back to the node of the Sender process. This `ClassLoader` then retrieves the class for `MyClass` and sends it back to the Receiver's node. The Receiver can now invoke `run` upon the instance of `MyClass`. At this point, the `MyClass` object must create an instance of `MySecondClass`, and the same sequence of events occurs, with the same `ClassLoader` being invoked as before. The Receiver can now continue execution. Figure 4 presents a sequence diagram of this interaction.

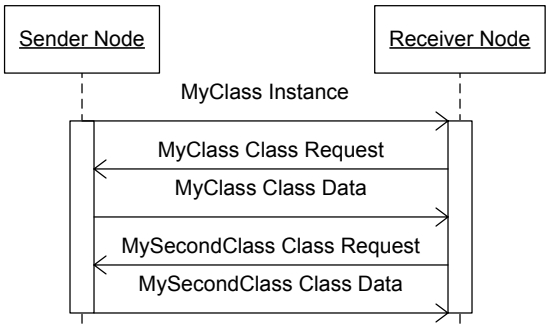

Figure 4: Single Hop Sequence

The second usage scenario for a single hop system involves an unknown object being an internal property of the sent object, as illustrated in code lines 6-8. This has the same interaction sequence, except this time `MySecondClass` is requested before a call to the `run` method is made. Our sequence diagram is the same, although the Receiver Node does not actually make a call to the Sender Node for `MySecondClass` in the latter instance until the call to `run`.

As this diagram illustrates, the interactions occurring between the Sender and Receiver are fairly simplistic, and the original model for the `DynamicClassLoadingService` could handle these, except for the internal properties of a class. The more complex interaction occurs when considering an architecture that incorporates one or more intermediary nodes.

5.3.2 Multi-hop Interaction

When class data has to pass through one or more nodes to reach its destination, we term this a multi-hop interaction. From the point of view of the Sender and Receiver, nothing has changed. The difference is that one or more remote processes sit between them, which we shall refer to as `Hop`. The purpose of the `Hop` process is to read in an object from the Sender and forward it to the Receiver.

For the usage scenario involving unknown classes the Sender creates an instance of the `MyClass` object and sends it to `Hop`. When the `Hop` process tries to recreate the object, its `ClassLoader` requests the class from the `ClassLoader` at the Sender node. When the class is received, the `Hop` node can then recreate the object and send it on to the Receiver. When the Receiver tries to recreate the object, it too requires the class for `MyClass`, and makes a request back to `Hop`. The `Hop` node can retrieve the class data it previously received and forward it onto the Receiver node. The Receiver can now invoke the `run` method upon `MyClass`. The Receiver node requires the class for `MySecondClass` at this point, and sends a request `Hop`. `Hop` does not have the required class for `MySecondClass`, and sends a request back to the Sender. It receives the necessary class back from Sender, and then stores it locally before forwarding it onto the Receiver. The Receiver can now define and resolve `MySecondClass` and allow the Receiver to continue. Figure 5 illustrates this.

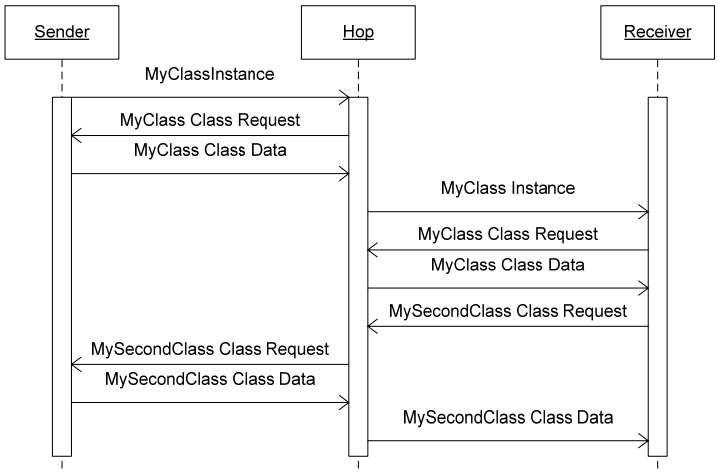

Figure 5: Multi-hop Object Creation Sequence

The interaction involving unknown classes as properties of a sent object again involves the Sender creating an instance of `MyClass` and sending it to `Hop`. When `Hop` receives the instance, it requests the class for `MyClass` from the Sender node. When this is received and an attempt to define and resolve the class is made, `Hop` discovers that `MySecondClass` is

also required, and a request is sent for the `MySecondClass`. When the instance of `MyClass` is sent to the Receiver, the same requests are sent back to `Hop`. As `Hop` has both the classes, it can handle the requests itself. Figure 6 presents the sequence diagram.

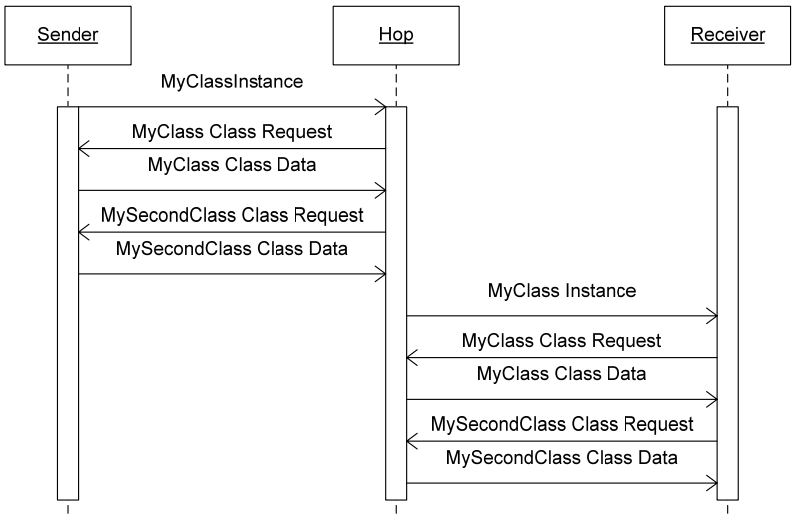

Figure 6: Multi-hop Object Property Sequence

5.4 Class Loading Architecture

In this section we demonstrate how `Hop` knew where to request the class data for `MySecondClass` from. Java has the ability to determine which `ClassLoader` originally defined the class of an object, and when sent from one node to another, the `DynamicClassLoadingService` determines which class loader originally defined the class. If it was the local `SystemClassLoader`, then the service retrieves the `ClassLoader` process responsible for handling requests for locally available classes. If the `ClassLoader` is a `DynamicClassLoader`, this is used instead. As each `ClassLoader` has a channel to accept requests, it's location can be provided as a property of the `DynamicClassLoaderMessage` that encapsulates a sent object.

When a node receives a `DynamicClassLoaderMessage`, it first extracts the request location from the message. It uses this location to check if it has already set up a `ClassLoader` for this object, and if not creates one. An attempt is then made to recreate the object contained within the `DynamicClassLoaderMessage`, using the relevant `ClassLoader`. This `ClassLoader` can request classes from the `ClassLoader` further down the chain, using the location provided. This allows the creation of code namespaces on each node based on where an object originally came from, as illustrated in Figure 3. As this diagram shows, each node has its own `ClassLoader` process which is relevant to its own `SystemClassLoader`, handling requests for classes stored locally at that node. Requests travel to the originating node, and responses travel forward to the destination that requested them. At a basic level, classes are always obtained from the originating node.

This structure has the further advantage of creating a trusted path from sender to receiver. Every time a request comes in, the `ClassLoader` checks its local file system first. If it can obtain the class locally, then it will send the local version. The version sent may not be the original version of the class that was sent through the node, but it does prevent malicious attempts to override trusted classes at the receiving node. In effect, this has created a code firewall between each node, with the previous node being trusted to pass on safe code only. Adding the ability to only load classes from certain channels into the node or from nodes with specific addresses enhances this level of security further.

5.5 Storing Class Data

One decision that was made in the design of this architecture was the storing of class data at each node as it is received. The contrary decision was to always allow requests to travel back to the origin node and then the class to travel back to the requesting node through all intermediaries. The latter approach would require more interactions between nodes, but fewer resources for managing classes at each node. However, the problem exists that a node may leave a system while classes it sent are still present, and if copies of classes were not retained locally, any further requests by other nodes will be unsuccessful. Also, when we consider that we wish to achieve some level of disconnected design, the approach of having classes available throughout the system is better.

The storing of code is also a major consideration for the pony framework [8]. The advantage in pony is that each system has a single master node, which could store code and distribute it to slave nodes. Although this creates a single point of failure, this is no different than if the master node went down anyway. Storing class data in a single location also builds structure and simplicity into the interactions. As pony is more controlled than the networked features of JCSP, this makes sense. The architecture for code mobility within JCSP reflects more on the fact that it is more disconnected and less controlled than pony. Therefore we make a recommendation that pony should adopt a code repository style approach for code mobility.

6. Evaluation of the Code Mobility Model

The architecture described for code mobility, and thereby mobile processes, has been developed with Java, JCSP and mobile agents in mind, and this does lead the limitation of having weak mobility only. However, the architecture developed is sufficiently disconnected to allow systems where the mobility of code between any nodes is almost at a very high level.

There are some other considerations. Our approach does allow mobile processes to be sent to any node that we are directly connected to, and code to be loaded between nodes as long as a connection back to the origin exists, or one of the nodes has the code required. If this is not the case then a system will fail. Although we have developed an architecture that exhibits a level of disconnected design, this serious vulnerability is still in place.

Consider a comparison between the JCSP approach and the recommendation for pony given above. We shall use the idea of locations and abstract locations [14], a location being a single node in which a mobile process can execute, and an abstract location being a collection of nodes within which a mobile process can execute. Within JCSP, only the former viewpoint is possible, as our nodes and architecture have left the system loosely coupled. Within pony, a mobile process can be considered at either the single node level or multiple nodes in the form of a mobile process within an application. The more rigid control offered by pony allows this. However, a mobile process wishing to move outside of the application has more difficulty in pony than in JCSP. This difference is inherent in the structures developed for JCSP and occam.

Mobile Agents and mobile processes also provide us with a very advanced abstraction, allowing a different viewpoint for systems development and understanding. Although it has been stated that there is no ‘killer application’ for mobile agent systems [6], as anything achievable with a MA system is also achievable using traditional methods, the simplicity of the abstraction is very powerful.

Finally, handling connection to, and management, of resources is an issue [6]. The proposal we make for this is the use of mobile channels that allow both our process and

channel ends to move together, allowing the process to stay connected to a resource. This is also true for mobile processes connected to other mobile processes. Communication between agents is seen as a problem in MA systems, and sent messages may end up chasing a mobile element around a system. This has been taken into consideration when designing the mobile channel model.

7. Channel Mobility

The model we have developed to allow channel mobility in JCSP is partially based on that of Mobile IP [10]. As this model was developed to allow mobility of devices in standard IP based networks, it was determined that it was a worthwhile model to replicate for mobile channels. We will first describe the shortcomings of the original model for migratable channels in JCSP.

7.1 Migratable Channels

Migratable cannels allowed a form of channel mobility in a rigid manner. The `MigratableChannelInput` had to be prepared prior to an actual move occurring. This was not a major issue, as this can be done by the movement function. The major flaw was data waiting to be read on the channel being rejected, and therefore lost. This is a dangerous system; losing messages is not safe. Also, the exception thrown by the JCSP system when data is rejected is not catchable, but handled by the underlying framework. So, although we can move a networked channel, the likelihood is the system will break during the process.

This problem leaves us with the need to create a new model to achieve channel mobility within JCSP. Therefore, the existing model for Mobile IP was examined and then replicated in a manner to allow channel mobility within JCSP.

7.2 Mobile IP

For Mobile IP to operate, three new functional entities are introduced [10]:

- *Mobile node* – the mobile device, which has a constant IP address.
- *Home agent* – resides on the mobile node's home network and forwards messages onto the mobile node's actual location.
- *Foreign agent* – resides on the network the mobile node is currently located, and receives messages from the home agent for the mobile node.

Also of import are the care-of-address (COA), which is where the mobile node is currently situated, and the home address, which is where the mobile node originated from. The Mobile IP model is presented in Figure 7.

When a mobile node is started, it is registered with a normal IP address on its home network. When it leaves its home network, the home agent is informed and is prepared to intercept any messages sent to the mobile node's address. The node then moves and when it arrives at its new location, informs the foreign agent, which registers the mobile node. The home agent is informed of the address of the foreign agent (the COA) and forwards messages onto this address.

Let us consider more fully the model in Figure 7. The communicating host is initially communicating with the mobile node using the home address. When the node moves, it informs the home agent to preempt messages sent to its home address and prepare them for forwarding to a new address. When the mobile node arrives at the foreign network, the

COA is determined and sent to the home agent, which forwards all messages onto this new address. From the point of view of the communicating host, nothing has changed, and messages are still sent to the same home address. When the mobile node wishes to communicate directly to the communicating host, it does so through the foreign agent.

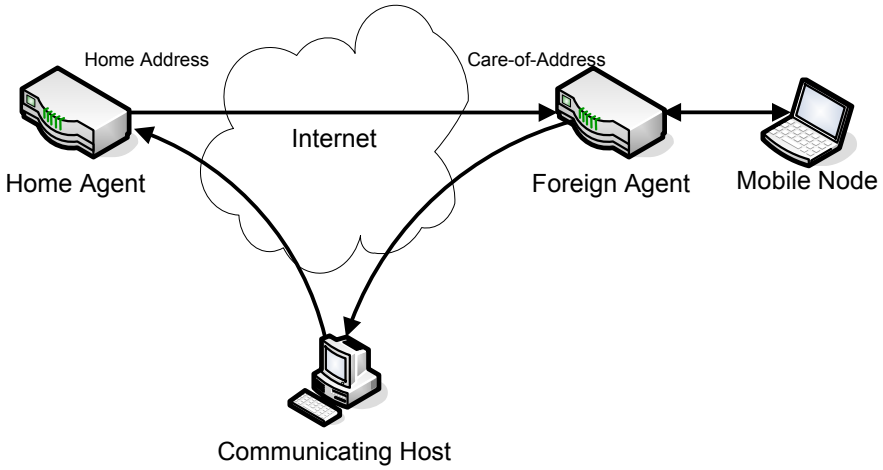

Figure 7: Mobile IP Model

Within JCSP we have the advantage that channels are one way only, meaning we can simplify our mobile channel model from that of Mobile IP. However, first we provide a basic overview of how JCSP networking operates.

7.3 JCSP Networking

A number of functional elements provide us with networked channel capabilities in JCSP. When we describe a networked channel in this respect, we mean an input end and output end connected together across a communication medium, and these are two distinct objects usually located on different nodes. We have:

- *Node* – a JVM where a JCSP application resides. Multiple nodes may make up an entire JCSP system, and multiple nodes may reside on a single machine.
- *Node address* – the unique identifier for a node.
- *Link manager* – listens for new connections on the node address. It is also responsible for creating new connections to other nodes.
- *Link* – a connection between two nodes. A node may have multiple links. The link is a process that listens for messages on its incoming connection and forwards them onto the net channel input.
- *Net channel input* – an input end of a networked channel.
- *Net channel location* – the unique location associated with a net channel input. The location takes the node address and a channel number as unique identification.
- *Net channel input process* – receives messages sent by links and passes them onto the net channel input. This process accepts messages from all links, which must determine which net channel input to pass the message onto by the unique channel number passed in the original address.

When a networked JCSP system is started, the node is initiated and allocated a unique address. The `LinkManager` on the node is started and listens for new connections from other `Nodes`. When a connection occurs, or the application requests that a new connection

to a remote `Node` be made, a new `Link` process is started. When a new `NetChannelInput` is created, a new `NetChannelInputProcess` is started, and a unique number allocated to the channel. Now when a `Link` receives a message from its connected `Node`, it extracts the unique channel number destination, and uses this to forward the body of the message to the `NetChannelInputProcess` which reads in the message and immediately writes to the `NetChannelInput`. The `NetChannelInput` is buffered so the `NetChannelInputProcess` does not block. These parts are placed together to provide the model presented in Figure 8.

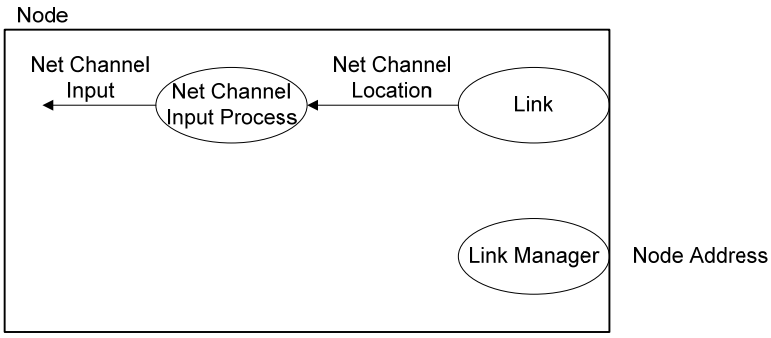

Figure 8: JCSP Networking

8. Mobile Channel Model

If we compare the two models of Mobile IP and JCSP Networking, we can see that there are similarities between them. For example the `Link` process acts as a router for messages received from a connection to a `NetChannelInputProcess`, similar to the foreign agent in Mobile IP, which forwards messages onto the mobile node on a foreign network. Therefore we can remove the foreign agent from our interpretation of the Mobile IP model, and implement the home agent to provide the functionality we need. The home address is a normal `NetChannelLocation`, and the COA likewise. This gives us the model presented in Figure 9.

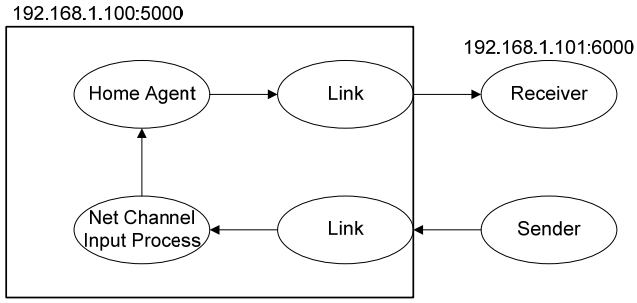

Figure 9: Mobile Channel Model

We have condensed the Sender and Receiver nodes down to single processes to keep the diagram simple. When a `NetChannelInput` migrates, it leaves a `HomeAgent` process that receives messages from the `NetChannelInputProcess`, and forwards them on accordingly. The sequence of interactions is:

1. Sender sends a message to the home address of the `NetChannelInput`. In Figure 9, the Sender would send to 192.168.1.100:5000/7 – 7 being the unique number associated with the channel.
2. The `Link` connecting to the Sender's `Node` reads the message, extracts the location, and forwards the message to the `NetChannelInputProcess`.

3. The `NetChannelInputProcess` reads in the message and sends it to the `HomeAgent`.
4. The `HomeAgent` reads the message and sends it to the new location of the Receiver, via the `Link` connecting to the `Node` that the Receiver is located. In Figure 9, the `HomeAgent` would forward the message to 192.168.1.101:6000/12.
5. The Receiver can now read the message sent by the Sender.

This model permits channels to move from one `Node` to another transparently, allowing multiple senders to send messages to a single address, instead of the system trying to inform them of the new location. As networked channels are Any-2-One, this is a simpler solution. If channels were One-2-One, we could inform a single sender. As this is not so, we implement a system allowing all senders to remain virtually connected to the receiver.

8.1 Shortcomings

Although this model does allow mobile channels, we still have a number of problems to overcome. Firstly, we still run the risk of losing a message whenever a channel moves to another location. Secondly, we lose synchronization between sender and receiver due to passing through the `HomeAgent`. Thirdly, we are creating a chain of `HomeAgent` processes between the mobile channel's home location, and all the locations it has visited during its lifetime. This problem also causes difficulties for the second shortcoming.

To overcome this, we introduce another channel to the `HomeAgent`, a configuration channel, which is responsible for receiving movement notifications from the mobile channel end, as well as being responsible for requesting messages from the `HomeAgent` when the mobile channel end is ready to read. The `HomeAgent` process waits for a request from the mobile channel end before forwarding the next message from its input channel. The configuration channel can also be used to send updates for locations; resulting in only one `HomeAgent` for a mobile channel. The operation of the `HomeAgent` is now:

```
09      Message msg = (Message)config.read();
10      if (msg instanceof MoveMessage) {
11        NetChannelLocation newLoc = (NetChannelLocation)config.read();
12        toMobile = NetChannelEnd.createOne2Net(newLoc);
13      }
14      else {
15        toMobile.write(in.read());
16      }
```

`toMobile` is connected to the mobile channel end, and `in` is the original channel set up to receive incoming messages to the mobile channel.

Using this extra channel overcomes the three problems mentioned above. We will no longer lose messages as no messages are stored at the mobile channel end; they are forwarded when a read occurs. Synchronization between reader and writer has been regained, as the writer is not released until the reader has committed to a read, and we no longer need to have multiple `HomeAgent` processes, as the original one is kept informed of the location of the mobile channel.

8.2 Comparison to pony

The mobile channel model in pony is implemented differently to the one we have described here for JCSP. The major difference is that pony takes a controlled approach to channel mobility, with a process controlling access to a networked channel, and ensuring messages are received by monitoring the location of channels. The model we have proposed for

JCSP has less centralised control, and acts in a peer-to-peer manner. Our approach therefore has certain advantages over pony. For example, pony has a single point of failure in its controlling process, although this makes channel mobility easier to control. The failure of a single `HomeAgent` process will only bring down a single mobile channel, but it does mean that the control of channel mobility is distributed, and harder to maintain.

There are still two major problems with the approach taken in JCSP. Our model does not allow selection of a mobile channel in an alternative, and each mobile channel requires an extra process to operate. As Java cannot handle a large number of processes (around 7000 threads is the maximum on a Windows platform [22, 23]), this can lead to a problem when large number of mobile channels are in operation. Our model requires three processes, a `NetChannelInputProcess` at the home location and at the new location, and a `HomeAgent` process at the home location, to provide the functionality of a single channel. Therefore we need to refine our model to provide more functionality, and use fewer resources at the same time. We therefore also propose a, as yet unimplemented, further model for channel mobility.

9. Future Mobile Channel Model

To implement a `MobileAltingChannelInput`, we can take advantage of recent additions to JCSP in the form of Extended Rendezvous [24]. This allows us to read a single message from a `NetChannelInput` into the `HomeAgent` and not release the writer until the forwarded message has been successfully received at the mobile channel's actual location. The code to achieve such an operation would be:

```
17      public void run() {
18        Object message = in.beginExtRead();
19        toMobile.write(message);
20        in.endExtRead();
21      }
```

This blocks the writer until the full forwarding operation has been completed, but does not take into account rejection of messages at the mobile end that occurs when a channel moves. Our `HomeAgent` process therefore becomes:

```
22      public void run() {
23        Guard[] guards = {config, in};
24        Alternative alt = new Alternative(guards);
25        while (true) {
26          switch (alt.priSelect()) {
27            case 0:
28              NetChannelLocation newLoc = (NetChannelLocation)config.read();
29              toMobile.recreate(newLoc);
30              break;
31            case 1:
32              boolean writing = true;
33              Object message = in.beginExtRead();
34              while (writing) {
35                try {
36                  toMobile.write(message);
37                  writing = false;
38                catch (ChannelDataRejectedException cdr) {
39                  NetChannelLocation loc =(NetChannelLocation)config.read();
40                  toMobile.recreate(newLoc);
41              }
42              in.endExtRead();
43              break;
44      }}}
```

This code also defines how to handle the location of the channel changing. First we check if the configuration channel or the `in` channel into the `HomeAgent` is ready (lines 24-26). If the configuration channel is ready, we read in the new channel location and recreate the channel to the mobile channel end, allowing the `HomeAgent` to communicate with the new location (lines 27-30).

If the `in` channel has data ready (line 31), we begin reading from the channel and try to write to the mobile channel. There are two possible outcomes. Either, the data is successfully read by the mobile channel and we can continue (lines 36-37), or the mobile channel end could move before the data is read and therefore the data is rejected (line 38). In this case, we read the new channel location and recreate the channel connected to the mobile end (lines 39-40) and try to write again. Once the data has been successfully written, the writing process can be released (line 42) and the process continued.

This proposed `HomeAgent` process can also accommodate the functionality of the `NetChannelInputProcess`, thereby removing the need for two processes on the home node and reducing the amount of required resources. Thus, we have solved both the problem of selection and resource usage. The mobile end can alternate as it knows if it has received a message on its buffer.

10. Evaluation of Mobile Channel Model

The model for mobile channels presented coincides with the distributed nature of JCSP, and is therefore deemed a suitable approach for providing mobility of channels. It differs from that of pony, although this is due to the different approach taken for distributed systems. Although each has their good and bad points, it is left to the individual to choose whether the controlled approach of pony or distributed approach of JCSP is better for the system they wish to implement. As both systems have fairly similar performance in their networked channels, this is justified.

The JCSP approach as currently implemented does have its shortcomings however. Selection of a mobile channel and the number of resources used are an issue. A plan for the future approach to channel mobility does overcome these shortcomings, but does require some changes to the underlying functionality of JCSP as it stands.

There is still a question of performance of mobile channels in JCSP, as each sent message has to be forwarded onto the actual location of the channel end. This doubles the amount of time taken to send a message, as it must make a hop before arriving at its destination. The performance of the mobile channels has yet to be evaluated, and this is left for future work.

As the model developed is based on an already implemented model, that of Mobile IP, we can argue that it is sufficiently sound as an approach for channel mobility. Although we have not described how output channel mobility is achieved, this is trivial. Networked channels are Any-2-One, so when an output channel end moves we need only send the location of the input end and reconnect at the new location. No data is kept at the output end, and the channel blocks until the write operation is completed, so the output end cannot move. Therefore, we have both input and output channel end mobility across the network, implemented in a manner that is transparent to both reader and writer.

11. Future Work

Although we have presented models for both channel mobility and process mobility, we have not discussed how more complex processes can be moved from one node to another

due to the complexity of suspending a running network of processes. This has been discussed at length in terms of graceful resetting [25] and how to place processes into a state so that they can be migrated safely [5]. These ideas have been presented for occam, although it is also possible to implement them in JCSP with the ideas presented for JCSP poison [26]. These principles have been implemented in the new edition of JCSP [24].

Also, what we can exploit specifically in Java to allow complex process migration has not been investigated. As Java uses object serialization to send objects to remote machines, the mechanism can be taken advantage of to allow signaling of a process when it should start preparing itself to move. This has also been taken advantage of in the channel mobility model to a certain degree. Any serializable object in Java can override methods to allow the dictation of how the object behaves when it is being written to or read from an object stream. The write method can be used to allow the signaling of the process that it should prepare itself to move. However, this will require a great deal of extra logic built into existing processes. This can be directly linked back to Java's lack of support for strong code mobility.

Other future work includes the evaluation of the mobile models against other approaches to code mobility, especially mobile agent platforms. It is hoped that this will allow JCSP to be put forward as a possible framework for mobile agent systems in the future, with the advantage of taking a more formal and structured approach than the standard remote procedure call and object-orientated one.

12. Conclusions

By utilising existing features of both Java and JCSP, we have shown that it is possible to develop transparent methods of both process and channel mobility between distinct networked nodes. By developing our models from those of mobile agents and Mobile IP, we have taken into account some of the pitfalls already discovered for other logical mobility systems. Our models still need to be thoroughly tested and examined, and for mobile channels in particular additions need to be made to JCSP itself to allow the best use of the resources available, as well as allow selection of mobile channels in an alternative.

References

[1] P. H. Welch and J. M. R. Martin, "A CSP Model for Java Multithreading," in Proceedings International Symposium on, Software Engineering for Parallel and Distributed Systems, 2000, pp. 114-122, 2000.

[2] 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.), Proceedings International Conference Computational Science - ICCS 2002, Part II, Lecture Notes in Computer Science 2330, p. 695, Springer Berlin / Heidelberg, 2002.

[3] K. Chalmers and J. M. Kerridge, "jcsp.mobile: A Package Enabling Mobile Processes and Channels," in J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood (Eds.), Communicating Process Architectures 2005, pp. 109-127, IOS Press, 2005.

[4] F. R. M. Barnes and P. H. Welch, "Prioritised Dynamic Communicating Processes: Part 1," in J. Pascoe, P. H. Welch, R. Loader, and V. Sunderam (Eds.), Communicating Process Architectures 2002, pp. 321-352, IOS Press, 2002.

[5] F. Barnes and P. H. Welch, "Communicating Mobile Processes," in I. R. East, D. Duce, M. Green, J. M. R. Martin, and P. H. Welch (Eds.), Communicating Process Architectures 2004, pp. 201-218, IOS Press, 2004.

[6] G. P. Picco, "Mobile Agents: an Introduction," Microprocessors and Microsystems, 25(2), pp. 65-74, 2001.

[7] M. Schweigler, A Unified Model for Inter- and Intra-Processor Concurrency. PhD Thesis, The University of Kent, Canterbury, UK, 2006.

[8] M. Schweigler and A. Sampson, "pony - The occam-π Network Environment," in P. H. Welch, J. M. Kerridge, and F. R. M. Barnes (Eds.), Communicating Process Architectures 2006, IOS Press, pp. 77-108 2006.
[9] J. White, "Mobile Agents White Paper," General Magic, 1994. Available from: http://citeseer.ist.psu.edu/white96mobile.html
[10] C. E. Perkins, "IP Mobility Support for IPv4," IETF, Technical Report RFC 3334, 2002.
[11] R. R. Brooks, "Mobile Code Paradigms and Security Issues," IEEE Internet Computing, 8(3), pp. 54-59, 2004.
[12] H. S. Nwana, "Software Agents: An Overview," Knowledge Engineering Review, 11(3), pp. 205-244, 1996.
[13] K. Rothermel and M. Schwehm, "Mobile Agents," in A. Kent and J. G. Williams (Eds.), Encyclopedia for Computer Science and Technology, New York, USA: M. Dekker Inc., 1998.
[14] V. A. Pham and A. Karmouch, "Mobile Software Agents: an Overview," IEEE Communications Magazine, 36(7), pp. 26-37, 1998.
[15] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding Code Mobility," IEEE Transactions on Software Engineering, 24(5), pp. 342-361, 1998.
[16] M. Delamaro and G. P. Picco, "Mobile Code in .NET: A Porting Experience," in N. Suri (Ed.), Proceedings of Mobile Agents: 6th International Conference, MA 2002, Lecture Notes in Computer Science 2535, pp. 16-31, Springer Berlin / Heidelberg, 2002.
[17] P. B. Hansen, "Java's Insecure Parallelism," SIGPLAN Notices, 34(4), pp. 38-45, 1999.
[18] P. Troger and A. Polze, "Object and Process Migration in .NET," in Eighth IEEE Workshop on Object-Orientated Real-Time Dependable Systems (WORDS '03), p. 139, IEEE Computer Society, 2003.
[19] K. A. Hummel, S. Póta, and C. Schusterreiter, "Supporting Terminal Mobility by Means of Self-adaptive Communication Object Migration," in WMASH '05: Proceedings of the 3rd ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots, pp. 88-91, ACM Press, 2005.
[20] P. Braun, S. Kern, I. Müller, and R. Kowalczyk, "Attacking the Migration Bottleneck of Mobile Agents," in AAMAS '05: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1239-1240, ACM Press, 2005.
[21] X. Zhong, C.-Z. Xu, and H. Shen, "A Reliable and Secure Connection Migration Mechanism for Mobile Agents," in Proceedings 24th International Conference on Distributed Computing Systems Workshops - W4: MDC (ICDCSW'04), pp. 548-553, IEEE Computer Society, 2004.
[22] N. C. 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, 2003.
[23] K. Chalmers and S. Clayton, "CSP for .NET Based on JCSP," in P. H. Welch, J. M. Kerridge, and F. R. M. Barnes (Eds.), Communicating Process Architectures 2006, pp. 59-76, IOS Press, 2006.
[24] P. H. Welch, N. Brown, J. Moores, K. Chalmers, and B. Sputh, "Integrating and Extending JCSP," in A. McEwan, S. Schneider, W. Ifill, and P. Welch (Eds.), Communicating Process Architectures 2007, IOS Press, 2007.
[25] P. H. Welch, "Graceful Termination - Graceful Resetting," in A. W. P. Bakkers (Ed.), oUG-10: Applying Transputer Based Parallel Machines, pp. 310-317, 1989.
[26] B. Sputh and A. Allen, "JCSP Poison," in J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood (Eds.), Communicating Process Architectures 2005, IOS Press, 2005.