INVESTIGATING COMMUNICATING SEQUENTIAL PROCESSES FOR JAVA TO SUPPORT UBIQUITOUS COMPUTING

KEVIN FRASER CHALMERS

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS OF NAPIER UNIVERSITY FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN COMPUTING

OCTOBER 2008

Abstract

Ubiquitous Computing promises to enrich our everyday lives by enabling the environment to be enhanced via computational elements. These elements are designed to augment and support our lives, thus allowing us to perform our tasks and goals. The main facet of Ubiquitous Computing is that computational devices are embedded in the environment, and interact with users and themselves to provide novel and unique applications.

Ubiquitous Computing requires an underlying architecture that helps to promote and control the dynamic properties and structures that the applications require. In this thesis, the Networking package of Communicating Sequential Processes for Java (JCSP) is examined to analyse its suitability as the underlying architecture for Ubiquitous Computing. The reason to use JCSP Networking as a case study is that one of the proposed models for Ubiquitous Computing, the π-Calculus, has the potential to have its abstractions implemented within JCSP Networking.

This thesis examines some of the underlying properties of JCSP Networking and examines them within the context of Ubiquitous Computing. There is also an examination into the possibility of implementing the mobility constructs of the π-Calculus and similar mobility models within JCSP Networking. It has been found that some of the inherent properties of Java and JCSP Networking do cause limitations, and hence a generalisation of the architecture has been made that should provide greater suitability of the ideas behind JCSP Networking to support Ubiquitous Computing. The generalisation has resulted in the creation of a verified communication protocol that can be applied to any Communicating Process Architecture.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my two supervisors, Jon Kerridge and Imed Romdhani, for providing the necessary skills and assistance to complete this work. I would also like to thank the School of Computing at Napier University for financing my studies.

Dedication

For Tracy. Look, all finished now.

Chapter 1 Introduction

Computers are everywhere. From mobile phones and watches, to corporate databases and industrial control systems, every day we interact with more and more computational devices in our daily lives. In one morning, between awaking and arriving at the office it is possible to interact with a plethora of computational devices in one form or another. Alarm clock, shower, radio, TV, MP3 player, mobile phone, bank machine, laptop. This is but a small list of devices with which we may interact with inside the first few hours of the day. But what does this mean for the world at large, and where are we going within this new technological age? Enter the era of Ubiquitous Computing.

1.1 Motivation

This motivation for this research came about from initial work within JCSP (Communicating Sequential Processes for Java) Networking to incorporate code mobility and thus lead to distributed mobile processes within JCSP [1]. By enabling code mobility within JCSP Networking in an easier and more concise manner, it became possible to investigate mobile agent scenarios with JCSP Networking [2], and likewise Ubiquitous Computing scenarios [3]. The ability to augment functionality and have dynamic architectural topologies in a distributed environment is an enabling factor of Ubiquitous Computing, and thus investigating JCSP Networking within the context of Ubiquitous Computing becomes interesting.

1.2 Ubiquitous Computing

Ubiquitous Computing is a research area concerned with not only the vast number of computational devices in the environment, but also with how they can be made to interact with one another. The introduction of this research field is generally attributed to Weiser [4], although the origins are in 1988 at the Xerox Palo Alto

Research Centre (PARC) [5]. At this time, an interactive whiteboard was developed which encouraged Weiser to look at how people interact with computationally enabled physical objects. This led to various scales of devices being developed, ranging from the whiteboard sized to early handheld computers and tags such as PARCTAB [6]. Simultaneously, early location aware systems were being developed [7] and the amalgamation of these ideas lead to Ubiquitous Computing.

The main aim of Ubiquitous Computing is to connect the real world with the computational, and also interlink the computational on a scale never before seen. For example, a door may be made to open (or not) automatically as a person approaches it. This is a simple example, but underlines the key idea of physical and computational merging. The connection of numerous varied devices comes into play when it is considered how the door knows who to open for. Sensors could be scattered around the environment and their readings sent to a centralised system which identifies the person and their intent and sends a message to the door accordingly. Another approach would be the use of a tag carried by the person which the door itself detects and acts upon accordingly.

1.3 Mobility

Dynamic interactions enable Ubiquitous Computing environments, due to the requirement of adaption within Ubiquitous Computing [8, 9]. Mobility is a key factor when considering dynamic interactions, both mobility of devices and logical mobility of the individual components of an application. This thesis focuses on the latter form of mobility.

Software, or logical, mobility requires runtime transfer of components between devices. Formal mobility models, and in particular the π-Calculus [10], have been proposed as enabling reasoning of Ubiquitous Computing applications [11]. The π-Calculus incorporates name passing within a process calculus, which enables dynamic topologies of interacting processes by allowing channel connections to be migrated between components. Channel mobility enables process mobility, and thus the mobility of channels and processes in a suitable software framework can be seen as enabling Ubiquitous Computing environments.

There are frameworks available that allow development of channel and process mobility models, such as **occam**-π [12] and JCSP [13, 14]. Both are based on another process oriented model – Communicating Sequential Processes [15, 16]. Work on JCSP has enabled simpler usage of the mobility features [1, 17], and the ubiquitous availability of Java – being available on a multitude of devices – encourages exploration of JCSP in a Ubiquitous Computing context.

1.4 Communicating Sequential Processes for Java

Enabling distributed mobility of channels and processes is difficult [17]. JCSP Networking allows construction of distributed channel and process models, and the inclusion of the mobility extensions enable basic channel and process mobility. By providing mechanisms to transparently create virtual networked channels across communication mechanisms, JCSP Networking provides a good initial platform to base an investigation into Ubiquitous Computing.

1.5 Aims

The aim of this thesis is to examine JCSP Networking within the context of Ubiquitous Computing. For this, there are two main research questions:

- *Is the current implementation of JCSP Networking a suitable framework for the development of Ubiquitous Computing systems?*
- *What are the practicalities of implementing the mobility abstractions of the π-Calculus within JCSP Networking?*

These two questions can be broken into further objectives.

1.5.1 Suitability of JCSP Networking for Ubiquitous Computing

To examine the suitability of JCSP Networking for Ubiquitous Computing, a number of properties of interest must be discovered, and experiments conducted to examine whether these properties are suitably supported in JCSP Networking. If these properties are not supported, then the problems with JCSP Networking that limit usage within Ubiquitous Computing must be discovered. Furthermore, an investigation into whether these problems can be overcome is also required.

1.5.2 Practicalities of Mobility

To examine mobility, there are three points to consider. Firstly, what are the advantages of taking such an approach to mobility in comparison to standard logical mobility models such as object-orientation? Secondly, can a suitable channel mobility model be developed that enables the type of dynamic interactions required by Ubiquitous Computing? Finally, can process mobility be enabled in such a manner that allows components to move freely through an environment such as Ubiquitous Computing?

1.6 Contribution

The work presented within this thesis contributes in a number of areas. Firstly, an examination of the current implementation of JCSP Networking within a resource constrained environment has been undertaken and various properties of the architecture calculated to provide expected performance of the underlying communication mechanism. The underlying messaging mechanism has been examined and layout and structure of sent messages extrapolated.

This thesis also describes a new implementation of JCSP Networking that overcomes the problems of the current implementation of JCSP Networking when considering Ubiquitous Computing scenarios. This new architecture is a reduced and refined version of the existing architecture. Importantly, a new protocol is proposed and developed that promotes inter-operability between different communicating process architecture frameworks. The new implementation is also examined by repeating the experiments performed on the original implementation, and thus showing improvements within the new implementation of JCSP Networking. The protocol has had a SPIN model created to verify its operation.

Certain properties of the original and new architecture are also examined against properties that are of interest to Ubiquitous Computing scenarios, which enables examination of the suitability of JCSP Networking for Ubiquitous Computing applications.

An analysis of different approaches to connection mobility in the context of practical distributed channel mobility is also presented. Seven different models of channel mobility are examined against properties of interest, allowing categorisation of the different models. This categorisation allows closer examination of the possible suitability of the different connection mobility models when considering the dynamic requirements of Ubiquitous Computing.

Finally, a method to transform JCSP processes into strongly mobile processes is also presented. This method allows active process networks to effectively be paused and subsequently resumed at a new location. The ability to pause process networks in this manner is novel, and builds upon existing approaches to capturing process network state.

1.7 Thesis Structure

This thesis takes the following structure. In Chapter 2 an investigation into the objectives is presented. Chapter 3 presents the current implementation of JCSP Networking and Chapter 4 analyses the current implementation by performing experiments within a suitably resource constrained environment. Chapter 5 proposes a new implementation of JCSP Networking to overcome highlighted problems, and Chapter 6 examines this new implementation by repeating the experiments conducted on the original implementation. Chapter 7 investigates possible channel mobility models, highlighting strengths and weaknesses of each and reflects these features back into the context of Ubiquitous Computing. Chapter 8 reviews techniques that have been proposed to permit process mobility, and then proposes an approach that may help processes exhibit the strong mobility aspired to by mobile agent systems, which are another proposed approach to Ubiquitous Computing. Finally, in Chapter 9 conclusions are drawn and future work proposed.

Chapter 2 Background

In this chapter, an investigation into Ubiquitous Computing is presented. Requirements and challenges are presented, and in particular software architecture properties are examined. Mobility, one of the key factors of Ubiquitous Computing, is also examined in depth. Finally, background information into Communicating Process Architectures is presented, focusing on JCSP and linking properties of Java to Ubiquitous Computing requirements.

2.1 Ubiquitous Computing

Historically, Ubiquitous Computing is attributed to Weiser [4, 5], the original focus being on computational devices of different scales being embedded within the environment. Ubiquitous Computing is also sometimes referred to as Pervasive Computing [18], although there are differences which shall be highlighted presently. First, general descriptions of Ubiquitous Computing are presented.

2.1.1 Describing Ubiquitous Computing

Numerous descriptions of Ubiquitous Computing exist, partially from the differing contexts that the description may come from. Ubiquitous Computing can be considered the availability of computational resources wherever we go [19], an extension of the mobile computing paradigm of all the time anywhere, to everywhere at all times with any device [20]. A common theme is the disappearance of technology into the background [21, 22], which allows focusing on the task at hand rather than the technology itself [23]. The general notion is that it moves computing forward to many devices to many users [21], a natural progression from the many users to one device mainframe era, through the one to one relationship of the PC era and the step through the Internet era of hybrid one to one and many to one relationships.

Ubiquitous Computing can also be considered as Everyday Computing [24], occurring within our everyday lives without our knowledge. This leads to the natural progression of Pervasive Computing, which focuses more on smart spaces and ambient intelligence [25]. In fact, Pervasive Computing can be thought of as the application of Ubiquitous Computing ideas, as Pervasive Computing extends the focus from small devices, network protocols and power consumption towards remote data access, smart spaces and context awareness [26].

However, the terms Ubiquitous Computing and Pervasive Computing are often interchanged, Pervasive Computing sometimes being referred to as research into mobile connected ubiquitous devices [27], or environments requiring little user interaction [28]. For this reason, Pervasive Computing ideas must also be considered when discussing Ubiquitous Computing, due to the tight coupling of the research areas.

Pervasive Computing is not only considered the outcome and application of Ubiquitous Computing ideas. It is also considered the natural evolution of distributed computing through mobile computing [18], and thus is considered an extension of distributed computing with devices augmenting the environment [29]. There is also the argument that it emerged from requirements for coping with heterogeneous mobile devices requiring interconnection, while abstracting from the technology required for interconnection [30].

It would appear that Ubiquitous Computing therefore comes from a number of different areas, but is particularly focused on mobile and distributed systems interacting with embedded computational infrastructure. There is also focus on the user being only lightly engaged in the computational environment, although users are an integral part of the Ubiquitous Computing infrastructure [11]. These descriptions are very vague however, and some more concrete examples are necessary to fully appreciate some of the ideas behind Ubiquitous / Pervasive Computing.

2.1.1.1 Example Scenarios

Examples of Ubiquitous / Pervasive Computing applications generally focus on augmenting existing everyday tasks with computing technology. Satyanarayanan [31] describes a scenario where the application determines that the current network infrastructure cannot support transferral of user files prior to a flight departing, and therefore finds nearby infrastructure that can support the transferral in time. Another scenario describes editing a presentation at a workstation and then taking the work onto a mobile device and editing using voice commands. Cheng [32] describes a similar scenario where a user reviewing images on a handheld device is automatically given higher resolution and colour depth when better network bandwidth is available.

Banavar [23] describes a scenario where someone attending a meeting automatically switches to video conferencing on a mobile device when they leave early, and the video feed transferring to a screen in a car from the device when the car is entered.

A common field of interest is healthcare [33, 34]. Accessing patient records electronically on mobile devices in a secure manner is foreseen as a goal, so much so that it is seen as a foothill project in the Grand Challenge in Ubiquitous Computing Research [35].

These scenarios and example applications help illustrate the application areas where Ubiquitous / Pervasive Computing is aimed at. From the scenarios it also becomes apparent that the current context of the user plays a key role in deciding how an application should behave. In particular, location and the services provided in a location are paramount.

2.1.1.2 Location Awareness

Location awareness appears to be one of the driving factors behind Ubiquitous / Pervasive Computing, particularly from the business point of view [36]. Location allows discovery of nearby services [37-39], although determining which service to

use and the protocol to discover nearby services are ongoing research problems [25, 40].

The interest in location can be directly related to the notion of Pervasive Computing smart spaces. A smart space is merely a location that provides services to users, and thus the services provided are Location Based Services [36, 37]. The focus on locality is important when considering some of the other requirements of Ubiquitous / Pervasive Computing.

2.1.2 Requirements

Weiser's initial view of Ubiquitous Computing requirements focussed primarily on low power and wireless hardware components [41], with network protocols to permit access to media. Weiser does state that Ubiquitous Computing reaches further than normal mobile computing, and incorporates autonomous agent ideas. The notion of small lower powered devices is continued further to incorporating thin clients and thin servers populating the environment [21], which provide only minimal capabilities as standard, and are designed to be augmented during operations. This implies a deal of adaption within the computational environment.

A number of authors have tried to list the requirements for Ubiquitous Computing. Banavar [23] states that dynamic tasks, device heterogeneity, constrained resources and social computing are the main requirements. Kindberg [42] focuses on requirements from different aspects of Ubiquitous Computing, mainly looking at software challenges. Again, resource constraints, heterogeneous devices and adaption are seen as key requirements, along with scalability, robustness and service discovery. Robustness is also a key concern stated by Sousa [43].

Niemela [26] lists interoperability, heterogeneity, mobility, security, adaptability, autonomy and scalability as requirements. Mobility in this sense is more than simple device mobility however, and requires mobility of software components between devices also. Software mobility allows dynamic binding of components, and thus promotes adaption. This idea is repeated by Lindberg [44], who states that handling heterogeneity and the dynamic nature of users, services and environments is a key challenge to overcome.

The Grand Challenge in Ubiquitous Computing Research [35] approaches Ubiquitous Computing Science, stating that focus should be on system and software architectures, mobility, context awareness, language design, protocol design, and support tools and verification of these factors. In particular, a communication infrastructure beyond standard TCP/IP protocols is called for. Milner continues the modelling and scientific argument [11], discussing whether the inherent complexity and scale of Ubiquitous Computing can be modelled, and what this implies for engineering such applications. Sufficient models of abstraction are required to enable understanding of the underlying architectures.

da Costa [20] lists scalability, heterogeneity, dependability, security, integration, invisibility of the underlying infrastructure, and context awareness and management as requirements, stating that a sufficient middleware is required to support these features. Many of these requirements can be attributed from existing computing fields. Heterogeneity, scalability, dependability and security can be attributed to distributed computing, and spontaneous interoperation, mobility, and context awareness and management can be attributed to mobile computing. These ideas fit into the idea of Pervasive Computing extending distributed and mobile computing.

Examining requirements for Pervasive Computing repeats the common notion of extending distributed and mobile computing. Satyanarayanan [31] lists remote communication, fault tolerance and high availability from distributed systems, and mobile networking, adaptive applications and location sensitivity from mobile computing as requirements. These ideas are extended with Pervasive Computing requiring smart spaces and invisibility of the environmental architecture.

Henricksen [8] states that Pervasive Computing requires examination of four key areas: devices, software components, users and user interfaces. Of these four, devices require heterogeneous support and mobility, and software components require mobility, adaption, interoperability, scalability and component discovery and deployment. Henricksen focuses further on middleware [45], stating that support is required for heterogeneity, mobility, scalability, and fault tolerance.

Cardoso [29] adds Quality of Service (QoS) requirements also, due largely to the scale and user interaction requirements.

Cheng [32] considers the minimal human oversight requirements of Pervasive Computing spaces in relation to the dynamic requirements of user movement and changing resources such as bandwidth and service availability, and thus fault tolerance is a major consideration. Saha [18] repeats the call for a suitable middleware to interface between the hardware and applications within the environment, and also support the heterogeneous nature of these interactions. However, Edwards [46] states that total inter-operability between components and devices is not possible, due to the inability to predict future requirements and standards. Thus, limited interoperability is required and sensible extensions built upon it.

From the brief overview of requirements for Ubiquitous Computing and Pervasive Computing architectures, it can be seen that there are a number of common themes. In particular, the following properties seem to be of interest:

- *Interoperability* – to support heterogeneous devices and software components.
- *Performance* – to support Quality of Service and scalability, although a strict requirement on performance is not in itself a requirement.
- *Scalability* – due to the large number of device interactions envisioned.
- *Stability* – robustness and fault tolerance.
- *Adaptability* – the ability to adapt to different operating conditions.
- *Mobility* – to help support adaptability, both device and software mobility is a consideration.

Another common argument is the requirement of a software middleware to support these properties. Therefore, an analysis of software architecture properties is also required.

2.1.3 Software Architecture for Ubiquitous Computing

Proposals for Ubiquitous / Pervasive Computing software architectures generally focus on component oriented architectures. Garlan discusses the Aura framework [47], repeating the calls for mobility, adaptability and resource awareness within software components. The Aura framework works on the idea of tasks which follow users throughout the environment, tasks themselves being made of various components. The idea of tasks following users returns to the fundamental ideas behind Ubiquitous / Pervasive Computing. Garlan also calls for refocusing of software from monolithic enterprise applications to dynamic components, and states that a rethinking of how components are specified and implemented is required. A foreseen challenge is deciding on the types of interactions between low level infrastructure and the upper application task layer. Edwards [46] states that this interoperation layer must be minimal, and provide few fixed parameters, allowing the user / developer the ability to join devices together in sensible manners. However, Henricksen [45] approaches the problem by creating a transparent communication layer that is similar to CORBA, and allows various frameworks to create the required connections between components automatically.

Sousa returns to the Aura framework [43], calling for a rethinking towards activity oriented computing, which supports the notion of tasks being important. This requires dynamic reconfiguration of software architecture to support user needs, which relates to software mobility. A main argument is that application models aimed at Ubiquitous / Pervasive Computing do not consider that user tasks are generally defined at runtime, and therefore packaging for all user requirements at design time is bound to fail. The ability to suspend and resume existing tasks is simply not enough to support Ubiquitous Computing. da Costa [20] complements the inability to package all possibility at design time by arguing that a common API within a single framework will not support heterogeneity due to the lack of a common framework that can operate on all devices.

Hoareau [48] argues on implementing strict hierarchical component architectures with well defined interfaces connecting the components together. By conforming

to these requirements, applications can be made to adapt based on architectural rules built within an Architectural Description Language (ADL) and observed environmental resources.

A common approach to coping with autonomy and the idea of smart spaces in Pervasive Computing is to apply agent oriented architectures to Ubiquitous / Pervasive Computing. Zambonelli [49] promotes the usage of agents in Ubiquitous Computing applications, but does also warn of some dangers. Niemela [26] also supports agents as suitable prototypes for Ubiquitous Computing, and Jung [50] considers Ubiquitous Computing as a multi-agent system which is targeted at everyday life. This is also supported by the Grand Challenge in Ubiquitous Computing Research [35], where agents are considered the base platform to build Ubiquitous Computing systems upon. Molina [51] argues that multi-agent systems are becoming more relevant in Pervasive Computing environments, particularly as they provide an interface between users and the environment.

Another common viewpoint on software architecture is the requirement of mobile software architectures. For example, Henricksen [8] argues on transparent mobility supported by the underlying software architecture, and Cardoso [29] believes mobile agents support the adaption, performance and scalability requirements of Pervasive Computing applications. Milner [11] argues that modelling of Ubiquitous Computing applications should also be supported by formal mobility models, particularly due to the inherent mobility of users, devices and software components. Mobility of software is considered especially difficult to deal with, due to the lack of physical constraints placed on software mobility.

From a software architecture view point, it can be seen that agents are considered an interesting area for Ubiquitous Computing, coupled with mobile and dynamic architectures. These two facets shall be examined in greater detail in Sections 2.2 and 2.3 respectively. First a brief analysis of hardware requirements is presented.

2.1.4 Hardware for Ubiquitous Computing

Weiser's initial description of Ubiquitous Computing hardware [41] focussed on three different sizes of device – the tab, the pad, and the board. The tab can be

considered a small, pocket sized device, similar to the Active Badge system developed by Want [7], which enabled tracking of a badge wearer via infra-red sensors, and the PARCTAB device described by Schilit [6], which provided services that augmented a small handheld device. Pad sized devices were envisioned as small scrap computers – similar to pieces of paper or notepads – and board sized devices covered such items as electronic whiteboards.

Modern viewpoints on Ubiquitous / Pervasive Computing hardware focus largely on small scale devices with wireless connectivity. Hartwig [52] has argued on augmenting the environment with small wireless servers that provide services for users as a viable model, whereas Want [53] considers a small personal wireless server with no user interface as a more viable option. Cardoso [29] merely states that mobile devices are essential for Pervasive Computing, and Moors [54] integrates wireless technology with the service and adaption ideas to control how mobile devices behave based on locations determined by wireless beacons. For example, a phone can be made to go into silent mode when a beacon signals that the user is within a cinema.

Thus, modern Ubiquitous Computing hardware seems to focus on small mobile devices that are wireless enabled. The smart phone is seen as the first real world Ubiquitous Computing device [55], and mobile telephony and SMS text messaging are considered the first real world Ubiquitous Computing applications. Relating this to the software architectural considerations, any proposed framework should at least initially be examined within the context of wireless enabled mobile devices.

2.2 Agent Oriented Systems

The term agent within software is an often overused term based on the field of computing that is examining agent properties. Tokoro [56] takes a viewpoint where agents are concurrent objects that are autonomous so that they can perform tasks. The agent is considered capable of reacting to incoming events and reacting accordingly. Lange [57] also takes an object view point of software agents, stating that they are autonomous, reactive and goal driven. Iglesias [58] considers objects and agents similar due to both relying on message passing communication.

The combination of object ideas and concurrency come together in the description provided by Bauer [59, 60], where agents are considered more akin to active objects, with autonomy, reactivity (responds to events) and pro-activity (generates events) being the fundamental differences between standard objects and agents, or adding the ability for an object to autonomously say go and no when communicating with other computational entities. This idea is cemented by Wooldridge [61], who states that *"objects do it for free; agents do it because they want to."*

2.2.1 Describing Agents

Agents have a strong background in artificial intelligence. Nwana [62] distinguishes agent types based on three properties – the ability to learn, the ability to cooperate, and the ability to be autonomous. Depending on these capabilities, an agent may be considered as smart, collaborative or some other category. Silva [63] has defined that the artificial intelligence capabilities of an agent based framework depends on how strong the sense of agency is within the framework. A strong sense of agency provides an AI agent framework, a weak sense of agency is simply an agent based framework, and the object-orientated viewpoint of agency is really middleware. Silva considers agents to be active components that perform tasks on behalf of others.

Wooldridge [61] also considers the autonomy of agents as the important distinction, and considers this is accomplished by agents having encapsulated state and the ability to make decisions based on this state. Agents are also considered to be reactive and proactive, and must have social capabilities, or the ability to communicate.

Kendall [64] also considers the same properties as Wooldridge to be important, and also provides a layered model of capabilities of an agent, considering mobility as the highest level capability due to the provision of dynamic architectures. As already stated, dynamic architectures are important to Ubiquitous Computing applications.

Molina [51] takes a Ubiquitous Computing viewpoint on agents, and, as well as repeating the need to respond to events and communicate, states that agents

provide adaption and reasoning to allow usage within Ubiquitous Computing applications.

2.2.1.1 Autonomous Agents

When agents are described within the context of artificial intelligence, the autonomous behaviour of agents is usually the main focus of interest. The most common architecture for defining agent behaviour is the Belief, Desire and Intention (BDI) model [65, 66]. BDI defines that agents are given a set of goals (Desires), plans, some of which have been committed to (Intentions), and some internal state (Beliefs) that is used to make decisions. This model follows closely to the idea of encapsulated components that are reactive and proactive.

Another common method to define behaviour is the active object [67], which extends object orientation by allowing objects to have their own thread of existence. Behaviour must be added to the object to allow the active object to be more proactive. Garcia [68] attempts to incorporate behaviour by injecting code into objects using aspect oriented techniques, which may lead to some form of adaptive behaviour.

2.2.2 Modelling Agents

Considering agent orientation as a possible architecture for Ubiquitous Computing, the question arises on how agent oriented architectures are modelled. Kinny [66] describes an agent as having two models: internal and external. The internal model incorporates the internal state of the agent, and can incorporate such ideas as BDI. Externally, an agent utilises services and is also provided with a type based on a hierarchy.

Iglesias [58] focuses on the external service viewpoint of agents, considering the similarities between objects and agents when the communication mechanisms are compared. In particular, Iglesias argues that although both utilise message passing, agents have the ability to analyse messages and determine whether to execute them. This idea returns to the notion of an agent having the ability to say no, although Garcia's work on applying aspect oriented ideas to implement agent

behaviour in object orientation would allow an object some capability to refuse messages.

Bauer [60, 69] utilises UML to try and model agent based systems, providing extensions to help model the autonomy and active runtime of agents. However, Bauer does note that UML is not entirely suitable to model agent orientated architectures.

Others have tried a more formal approach to modelling agents. Luck [70, 71] has used Z Specifications to model agent oriented systems, and notes that there is difficulty due to the overuse of the term agent. Luck defines a hierarchy of properties that allows an agent to be defined. An entity is a set of attributes, and an object is an entity with a set of actions. An agent is an object with a set of goals, and an autonomous agent is an agent with a set of motivations, which allows goals to be modified.

Duvigneau [72] examines agents by utilising Petri-Nets, and argues that agents form hierarchies, thus requiring a *"nets within nets"* paradigm. Xu [73] also examines Petri-Nets as a method to model agent oriented applications.

Gonzalez [74] has approached agent development using Communicating Sequential Processes (CSP) [15, 16], and has argued that various behavioural aspects can be modelled using the CSP formalism. Gonzalez argues that the notion of agents and processes with CSP are strongly related.

Yu [75] utilises another process calculus, the π-Calculus, to model agent systems. Yu's argument centres on the dynamic architectures that the π-Calculus enables, and argues that the π-Calculus process is also very similar to an agent.

2.2.3 Summary

Although agents have been proposed as a possible architecture for Ubiquitous Computing, the fact that the term agent is over used does lead to questions on what Ubiquitous Computing views as an agent. The common features of agent descriptions focus on the ability to perform a task for another entity, autonomy, and activeness. However, autonomy is also ill-defined, and it is unclear whether

autonomy simply means being active and performing a task, or whether an agent should have intelligence and adapt to its environment. Considering one of the properties of Ubiquitous Computing is adaptability, it is likely the latter description that is being used to define Ubiquitous Computing agents.

The following section examines mobile software architectures, which is the second architectural viewpoint considered for Ubiquitous Computing. Mobility is also one of the requirements of Ubiquitous Computing, and therefore requires investigation.

2.3 Mobility

Software, or logical, mobility is different from the more commonly thought of physical mobility, and has a number of different challenges. Baude [59] distinguishes between mobile computing (physical devices) and mobile computation (mobile software components), and in particular looks at mobile active object systems. As Section 2.2 described, active objects have similarities to agents, thus there is a commonality between the mobile and agent architectures. Baude also distinguishes between strong and weak mobility, with strong mobility capturing the execution state of the component, and weak not doing so.

Fuggetta [76] argues that there is general confusion on what state mobility actually means. The state of a component may or may not include the current execution point, or program counter, of the mobile component. To be strongly mobile, this information must be captured and transferred transparently, without programmer intervention. Data state contains no execution state, and mobile data state allows weak component state to be transferred.

These views on logical mobility generally focus on the component, and the following section examines these ideas in greater detail.

2.3.1 Logical Component Mobility

Tröger [77] redefines strong and weak mobility to active and passive component mobility. A passive component can be considered one that has no path of execution, and can be considered to be data or code library mobility. Many frameworks provide this mechanism using serialization, which allows passive object

replication. Active component mobility describes components that have a path of execution, and Tröger mentions the possibilities of such components within Pervasive Computing environments.

Bettini [78] provides a further type of migration beyond weak and strong, that of full mobility. Full mobility allows an entire operating system process to migrate, and would be akin more to an entire application migrating rather than just an individual component. Ghezzi [79] considers a different third type of mobility, that is communication based rather than component based. For example, Remote Procedure Calls (RPC) can be rebound to enable a component to move and reconnect to existing components.

The idea of RPC enabling mobility is also considered by Cardelli [80], who also defines five separate mobile component properties. A component may provide control mobility, which allows the thread of control of a component to virtually transfer to another location, using RPC. Data mobility allows the transfer of data from one network host to another, and link mobility allows the migration of a connection between two components to be migrated. Object mobility allows the migration of objects, whereas remote evaluation permits an object to migrate to another location, execute and return. The interlinking of location and migration has been addressed by Roman [81], who states that location defines the position of the logical component, and thus a change of position is a change of location and therefore migration.

Phillips [82] has argued further on the notion of location, stating that a means of expressing a process location is required, both physically and logically. Phillips has also argued that the very nature of distributed mobile components requires concurrent behaviour, and that communication between components must be modelled.

Roman [81] also considers coordination between components, and believes that coordination and location are the two most important factors for a logically mobile framework. The consideration of coordination separately from the components

allows a decoupling of the components, and coordination should be considered separately to the actual mobile component behaviour.

2.3.2 Properties and Requirements

There are a number of considerations when developing a mobile architecture. Fortino [83] has argued that existing technologies such as RPC must be considered to allow interoperability between frameworks. As interoperability is seen as a key feature of Ubiquitous Computing, the argument is justified within the Ubiquitous Computing context. Openness to new architectures must also be available however.

Roman [81] has argued that component code and component state must be considered as first class elements of the component, and Welch [12] agrees that some form of passive state must be considered as part of a mobile component for there to be a reason for mobility. Roman has also stated that disconnection of components and subsequent reconnection is required to allow mobility, and thus algorithms to support message passing between mobile components while they move is also required.

A common requirement for logically mobile components is code mobility, and the following section examines this is more detail.

2.3.2.1 Code Mobility

Code mobility is the ability to transfer code from one host to another, and allow the dynamic loading of this code into an already running process. Ghezzi [79] has summarised a number of different applications of code mobility, providing the different behaviours each exhibit. These applications are:

- *Client-server* – server has the knowledge, resources and processors to execute the task. There is really no migration of actual code in this application.
- *Remote-evaluation* – client has the knowledge, whereas the server has the resources and processor.

- *Code on demand* – client has the resources and processor, and the server has the knowledge.
- *Mobile agent* – client has the knowledge and processor, and the server has the resources.

Fuggetta [76] has taken a view where there are two types of code mobility – process migration and object migration. The former is akin to Bettini's [78] full mobility. Fuggetta also considers some applications of code mobility, and metions location aware programming as an advantage. As described in Section 2.1, location is a key idea within Ubiquitous / Pervasive Computing. This helps to reaffirm mobility as a suitable architecture for Ubiquitous Computing applications.

Active networks are added as a code mobility application by Brooks [84]. An active network is a distributed system with the ability to adapt to environmental conditions by modifying the communication structure. Again, this dynamic nature is a requirement for Ubiquitous Computing, reaffirming mobility as a suitable Ubiquitous Computing architecture.

2.3.3 Mobility Architecture

So far, the discussion on mobility has focused on the description and requirements of logical mobility, and this has highlighted two separate mobile constructs – component mobility and the mobility of connections between components. In this section, a further analysis of mobile software architectures is presented.

Fuggetta [76] considers components and their interactions as the architectural constructs to consider within a logical mobility architecture, and Lopes [85] has stated that a clear separation of components via connectors is required to allow adaptation. Zheng [86] has also called for separate coordination and computation, with clear input and output interfaces defined.

Zheng and similarly Oquendo [87] have utilised Architectural Description Languages (ADLs) based on the π-Calculus [10] to help define the dynamic architectures of logical mobility systems. As stated, the π-Calculus has also been used to help describe agent based architectures (Section 2.2). Oquendo also considers a

separation of components and connectors within the mobility architecture, and considers the configuration of these components and connectors to be the high level architectural view of a logical mobility system.

Connection migration is examined by a number of authors. Milner [88] has called a mature connection mechanism as being able to communicate itself – a communication that can communicate another method of communication. Zhong [89] argues that connection migration must accommodate interacting components migrating simultaneously, and this should occur transparently and reliably. Molina [51] agrees with the notion of transparency, as well as location transparency as a whole. Molina also argues that mobility and communication are interrelated, requiring one another to operate.

May [90] has analysed the different types of mobility that both components and connections can exhibit, and how these effect a communication mechanism. May describes copying, moving and borrowing – copying replicates the sent entity at a new location, moving copies an entity and destroys the original, and borrowing is similar to moving, but with the mobile entity returning.

There is therefore a requirement for mobility of both components and connectors in the software architecture to support logical mobility. Mobile and agent oriented architectures have both been described as potential models for designing Ubiquitous Computing applications, and therefore examining these ideas in unison is desirable. Section 2.4 will discuss mobile agents. In the following subsection, object orientation is examined as logical mobility architecture. As the argument has been made to the similarities between agents and objects, mobile objects require further examination.

2.3.3.1 Object Oriented Architectures

There are problems when considering objects as mobile. Barnes [91] has highlighted that object orientation naturally supports mobility within the structure of its architecture, and mobile design is used extensively in single machine based applications. However, the problem stems from the inability to ensure that a mobile object can safely move all its parts.

Hoare [92] has highlighted the key problem when considering mobile objects. Object orientation allows aliasing of objects, in that an object may be accessible via more than one path of references from the root object. In fact, Hoare has stated that there is no method to explicitly name an object, rather just the connections to objects. With this in mind, when examining mobility it can be considered that objects are not first class elements, as there is no tangible method to own an object.

Locke [93] has argued that objects also have no location due to this lack of ownership. As mobility and location are intertwined, there is further evidence of a problem when considering objects as a mobility architecture. Vitek [94] goes further, and points out security flaws when considering mobile object systems that arise from covert channels (aliases) crossing protection domains.

However, there has been work on trying to protect against aliasing (for example [95]) by imposing ownership on objects. By enforcing strict encapsulation instead of weak encapsulation, many of the problems associated with object mobility can be overcome.

2.3.4 Formal Modelling of Mobility

As objects appear to be unsuitable for distributed mobile architectures, a more formal approach to modelling mobile architectures is required. There are two main approaches – state based and communication based. Mobile UNITY [85, 96, 97] provides a state based model of mobility, and is an extension to the CommUNITY formalism. Mobility is modelled by allowing components to change a location state variable, and also define behaviour based on this variable [98]. Locations can also be transferred between components [99]. Mobile UNITY has been used to model mobility protocols such as Mobile IP [100].

The π-Calculus [10] promotes the mobility of connections between components to a first class construct. Although the π-Calculus covers only part of the mobile capabilities of a mobile architecture, further work on mobility formalisms attempts to capture all aspects of mobility (for a summary see [101]).

Communication within the π-Calculus is synchronous in nature, and Phillips [102] has argued that distributed systems do not exhibit synchronous but asynchronous behaviour. Therefore Phillips proposes an asynchronous version of the π-Calculus to model real distributed mobile architectures.

Cardelli [80, 103] has extended the ideas presented in the π-Calculus and developed the Ambient Calculus. Unlike the π-Calculus, the Ambient Calculus promotes ambient as the first class mobile entity. An ambient is considered to be a bounded, nested collection of processes that migrates as a single entity. The idea of a bounded entity assists in modelling protection domains for applications.

As the π-Calculus has been proposed as both a model for agent orientation and mobile architectures, it would appear to provide a suitable model to investigate Ubiquitous Computing. This is also proposed by Milner [11], who states that to sufficiently support Ubiquitous Computing, protocols that enable communication mobility are required.

Mobile agents also provide a mechanism to combine the ideas of agents with the ideas for mobile architectures. In the following section, this area is further examined to determine the suitability of mobile agents for Ubiquitous Computing.

2.4 Mobile Agents

As with agents, the term mobile agent is often overused and lacks a particular definition. Lange [57], in a similar manner to normal agents, considers a mobile agent to be a software object that is reactive and proactive, but with the added capability of being able to change its execution environment. Contrary, Papastavrou [104] does not consider agent capabilities at all, and states that a mobile agent is a process that is dispatched from a source device to perform a task within another execution environment, and upon completion returns to the source. This description is also followed by Gray [105].

Spyrou [106] considers the active object view of agents and applies it to mobile agents. Data can be viewed as a collection of objects, and a mobile active object can therefore be seen as a mobile agent, or a mobile set of data with some

execution capabilities. Cabri [107] on the other hand sees data as the unit of exchange between several mobile agents which make up an application.

2.4.1 Using Mobile Agents

Picco [108] states that the key contribution of mobile agents (and code mobility in general) is that they allow location to be considered as a first class construct, and promotes functionality based on location. Returning to the Ubiquitous / Pervasive Computing argument, this is rather important as locality is seen as an important construct. Picco also discusses different analogies for mobile agents. A single move agent is simply a migratory service, whereas an agent that migrates many times is an actual mobile agent. Both of these analogies fit within the scope of Ubiquitous / Pervasive Computing, where services are also seen as an area of interest. Picco does warn that mobile agents are not always the best solution in all cases however, and Lange [57] also states that there has to be a reason for mobility.

Picco also returns to code mobility when describing mobile agents. A mobile agent consists of code, data state, and behavioural state. However, most agent systems are built utilising Java, which cannot promote strong mobility by capturing behavioural state, and modifications are usually required to allow strong mobility. Picco also points out that connections to resources can also be a problem, and this stems back to the unsuitability of objects for mobility purposes. Picco does not seem to consider connections between agents as being part of the mobile unit.

A number of authors have also overviewed mobile agent requirements and applied them to current mobile agent platforms (for example, Gray [105] and Silva [63]). In general, it has been argued that no mobile agent platform has suitably met all required properties, mainly stemming from the lack of strong mobility because of a reliance on Java, or lack of interoperability between different frameworks (e.g. Java and .NET agents). Silva does mention that there are commonalities between mobile agent platforms, which include usage of agent servers, autonomous active components, and distributed agent communication.

2.4.2 Advantages of Mobile Agents

Many authors have listed advantages of mobile agents, although authors such as Chess [109] have noted that there is no real application made possible only by mobile agents. Chess argues that the main advantage for mobile agents is a software engineering one, as they enable design of development of certain applications in a simpler manner.

Other authors have focussed on implementation advantages of mobile agents. Lange [110] lists reduced network load, reduced network latency, protocol encapsulation, asynchronous and autonomous execution, dynamic adaptability, heterogeneous applications, and robust fault tolerant applications. Returning to the requirements of Ubiquitous Computing, Lange's advantages of mobile agents cover interoperability, performance, stability and adaptability, leaving only scalability as an unanswered requirement. Lange also lists applications suitable for mobile agent systems, many of which can fit into the sphere of Ubiquitous / Pervasive Computing.

Gray [105, 111] considers mobile agents useful within dynamic, mobile computing environments, and also lists bandwidth conservation and other performance criteria as advantages, as does Picco [108]. Molina [51] adds a further advantage, that of simplified maintenance. As Ubiquitous / Pervasive Computing requires easily maintainable applications, due to the vast scale and minimal human interaction, there is further evidence to support mobile agents as a Ubiquitous / Pervasive Computing architecture.

2.4.3 Problems with Mobile Agents

There are some perceived problems with mobile agent approaches. Chess [109] has questioned the supposed advantages of efficiency and flexibility, and has also raised concerns of the security of mobile agent platforms. Considering the requirements of a mobile architecture, which promotes both component and connection mobility, Silva [63] has raised the question on what happens when two connected agents move simultaneously. This problem can be related to the lack of consideration of

connection mobility within mobile agent approaches, and code mobility applications in general.

Picco [108] believes that many of the problems within mobile agent applications stem from the usage of Java and similar object-oriented platforms. In particular, the reliance on the threading mechanisms in Java leads to problems with scalability, and communication between components has been raised as an issue. Communication in this respect can be considered as the connections between other agents and resources. Picco notes that this problem is generally overcome by only allowing co-located elements the ability to communicate with one another. This does negate some possible applications of mobile agents however. Picco also argues that there is too much focus on how mobile agents can be developed, and not on why they should be. Java is seen as a blessing and curse in this respect, and there is generally no willingness to develop applications that interact with existing applications.

The communication and coordination between agents problem has been examined by a number of authors. Cabri [107] believes coordination between agents is fundamental, and utilises Linda like coordination to overcome the problem. Fortino [112] defines an event based architecture built upon existing communication middleware, including Linda and also RPC.

2.4.4 Mobile Agent Platforms

Most mobile agent platforms have been applied within Java. Lange [57] describes the Aglet API for Java, and believes that Java provides a number of characteristics that make it advantageous to use. These include platform independence, secure execution, dynamic class loading, multithreaded programming, object serialization and reflection. Many of these advantages focus on allowing simple mobility of agents and code (dynamic class loading, object serialization, reflection), and execution of agents (platform independence, multithreaded programming). However, from a Ubiquitous / Pervasive Computing point of view, platform independence is advantageous. Molina [51] also repeats these advantages when considering Ubiquitous Computing.

However, Lange does point out some limitations within Java when considering mobile agent platforms. These include inadequate support for resource control, no protection of referenced objects, no sense of object ownership, and no strong mobility. Many of these have been discussed already, although the resource control does raise questions on scalability when considering Ubiquitous Computing.

Izatt [113] describes the Ajents platform, which also utilises Java. Although adding little in comparison to Aglets, Izatt does argue against the suitability of Java Remote Method Invocation (RMI) for mobile agent communication, due partially to the ownership problems imposed by object orientation. Ajents also tries to overcome the strong mobility problem of Java by imposing asynchronous communication and allowing an agent to be called and a reply to be waited upon by the caller. This does not solve the two simultaneously migrating agent problem highlighted by Silva.

The JADE Agent Platform [114] focuses on communication and coordination as opposed to mobility [115], although it has been shown that various design common mobile agent design patterns can be implemented using JADE [116]. JADE provides an almost strong approach to mobility, in that execution state is captured without extra developer code, but not at any point during execution. JADE is also implemented in Java, and provides its own communication mechanism that utilises serialization.

Gray [111] describes the D'Agents system, which allows agents to be developed using multiple languages. D'Agents appears to be the only attempt at doing this so far, although there are limitations when considering communication and migration of agents between platforms. Different language platforms also do not allow the same capabilities as others. For example, Java D'Agents does not attempt to overcome the strong mobility problem caused by Java.

2.4.5 Summary

Although mobile agents are a promising approach to Ubiquitous Computing, there are still a number of disadvantages. Interoperability has been raised as an issue, and there seems to be little regard for promoting connection mobility. Considering

the π-Calculus as a model for Ubiquitous Computing, the mobility of connections is seen as more prominent. Java does appear to be the platform of choice, although there are limitations. However, considering how prominent Java is as a platform, it does provide a suitable starting point to investigate Ubiquitous Computing middleware.

In the following section, an approach that incorporates both component and connection mobility is discussed, which can incorporate many of the ideas from agents, and thus provide a possible platform to support Ubiquitous Computing.

2.5 Communicating Process Architectures

Communicating Sequential Processes (CSP) [15, 16] is a formalism that describes a set of processes (components) communicating with one another via a set of events (connections). This is similar to the π-Calculus, where processes also communicate via events. There are currently a number of implementations of CSP behaviour, and in particular Java has the Communicating Sequential Processes for Java (JCSP) library [117], which provides the necessary constructs to build CSP like applications within Java. JCSP also has a package that enables these constructs to operate across a communication mechanism [14], thus providing a base mechanism to support distributed systems. At the heart of mobile agents and mobility architectures, there is the notion of a distributed architecture, and likewise for Ubiquitous / Pervasive Computing.

2.5.1 Similarities between CSP and Agent Orientation

Agents have their roots in the actor model, which are self contained, interactive, concurrently executing objects, with internal state and respond to messages from other agents [62]. 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).

The concurrent behaviour of agents and processes also brings a number of similarities, and Gonzalez [74] has utilised CSP techniques to describe agents. Petitpierre [118, 119] has argued on the similarities of active objects and CSP, and

considering the similarities discussed between agents and active objects, there does appear to be commonalities between a CSP process and an agent.

2.5.2 Mobile Processes and Mobile Channels

Recently, mobility has been of interest within the field of Communicating Process Architectures (CPA). Languages such as **occam**-π [12] have added channel and process mobility, and the distributed framework for **occam**-π, pony [120], also added mobility of networked channels. Connection mobility is seen as missing from mobile agent platforms, so applying these ideas may overcome this problem. Previous work on JCSP mobility [1, 17] has also attempted to incorporate both channel and process mobility.

There are advantages when considering mobility in this form and examining Ubiquitous Computing ideas. One of the required properties of Ubiquitous Computing is scale, and Milner [11] believes the π-Calculus will enable understanding of this scale. Ritson [121] has shown some of this capability, by implementing a system with millions of interacting mobile process components in a manner that can be considered simple to comprehend.

Considering the ideas of mobility presented thus far, it is possible to define how practically a mobile process can be defined. As location is integral to the notion of mobility, a mobile process can be considered as a process that has the ability to change location. This description is basically the same as a mobile agent.

2.5.3 Examining the Capabilities of JCSP Networking

As mobile processes can be considered similar to mobile agents, and as mobile channels enable the connection migration which is considered missing from mobile agent definition and application, JCSP Networking can be considered a possible architecture for Ubiquitous Computing applications. JCSP Networking brings together a number of strengths from the different fields that have been seen as applicable for Ubiquitous Computing, such as distribution and concurrency, and work on pony has shown that distributed channel mobility is possible. However, there are still properties that require examination.

Firstly, the defined requirements for Ubiquitous Computing (Section 2.1.2) must be analysed within the JCSP Networking architecture. This will require extensive analysis of the JCSP Networking architecture and the performance characteristics thereof. Due to the distributed nature of JCSP Networking, the common approaches to analyse networking performance of throughput and latency are required. Considering Ubiquitous Computing, these properties require examination in a suitably constrained environment, utilising wireless networking and mobile devices.

Java utilises serialization to enable transfer of objects between remote machines, and thus serialization requires examination (a discussion on Java's serialization mechanism is provided in Appendix A). JCSP utilises serialization, but the commonly utilised communication architecture for distributed Java applications is Java Remote Method Invocation (RMI). Serialization has been examined by a number of authors [122, 123] with the focus being on analysing performance based on the complexity of the objects. Analysis of RMI [123] has also shown that examining the individual parts of the communication mechanism allows greater insight into the underlying architecture. Applying these concepts to JCSP Networking is therefore worth considering.

The next consideration is the implementation of the distributed mobile channel structure. Although the pony architecture [120] has proposed a method to enable distributed channel mobility, a more in depth analysis of the suitability of this model within the context of Ubiquitous Computing is required. pony itself has some significant overheads associated with its channel mobility model.

Finally, it has been noted that strong mobility is a requirement of logical mobility, and that Java has problems in permitting this form of mobility. Therefore, the development of a technique to enable the strong mobility of processes is required. Due to the restrictions of Java, the actual capabilities of any technique must also be brought into question. Currently, within JCSP, distributed process mobility is only allowed at either the start state of the process, or when the process is in a stopped

state. The aim is to permit the same level of process mobility as local JCSP processes.

2.6 Summary

Within this chapter, an analysis of the requirements of Ubiquitous Computing has been presented, and has primarily focused on the underlying software architecture requirements for Ubiquitous Computing. By examining the potential models for software to support Ubiquitous Computing, it has been shown that mobility is seen as a key feature, and likewise the capabilities of distributed systems due to the distributed nature of the applications under consideration. Although several platforms provide some of the properties of interest, there are still limitations when considering such approaches as mobile agents when considering Ubiquitous Computing, therefore another approach has been proposed as requiring examination, utilising JCSP Networking as a test case.

In the following chapter, the current implementation of JCSP Networking is presented. The existing architecture and functionality are described, and some initial observations are made. These observations are required for further analysis of JCSP Networking against Ubiquitous Computing requirements, which is presented in Chapter 4.

Chapter 3 JCSP Networking

In this chapter, a description is presented of the current implementation of JCSP Networking. From this description, an initial examination of the structure and individual components required for the network architecture to operate and some initial observations are made prior to a more thorough evaluation of the implementation in Chapter 4. Section 3.1 presents the aim of JCSP Networking, and Section 3.2 presents the current architecture. Section 3.3 examines the functionality and Section 3.4 provides a brief analysis before initial observations are made in Section 3.5.

3.1 Aim of JCSP Networking

The core implementation of JCSP is aimed at providing constructs necessary for a CSP based concurrency model in Java. The network architecture expands JCSP by providing channels that operate over a communication mechanism. Two statements of the aim of JCSP Networking have been made. The first [14] alludes to the creation of process networks over a communication medium by interpreting the T9000 virtual channel model [124]. The second stated aim [125] is "*to build efficient, richly functional, scalable, distributed and dynamic evolving systems*". The second interpretation of the aim of JCSP Networking comes from a discussion on cluster computing, which is the main application area of JCSP Networking. The main aim of JCSP Networking can therefore be interpreted as the exploitation of parallelism in distributed system applications. This aim does not fit within the sphere of Ubiquitous Computing per se, but the scalability and dynamic architectures are requirements. Therefore, it can be claimed that JCSP Networking may be a suitable framework for Ubiquitous Computing.

3.2 JCSP Network Architecture

There are a number of components required to achieve the functionality within JCSP Networking, and in this section a description of these components shall be presented. Diagrams illustrating component interactions shall also be given.

The diagrams presented do not reflect previous reporting of JCSP Networking [14] as a number of modifications have been made. The original implementation of JCSP Networking utilised service processes for output channels, and the `EventProcess` described in the next section was also not present. The `LoopbackLink` was also a later addition to allow local channel ends to connect.

3.2.1 High Level View

Figure 1 illustrates the high level view of the current JCSP Networking architecture, presenting the key components and how they interact. Solid lines with arrow heads represent channel connections, and dashed lines represent object references. Ovals represent active components (processes), rounded rectangles represent a collection of active components and rectangles represent passive components (objects). Channels with an infinity sign are provided with an infinite buffer.

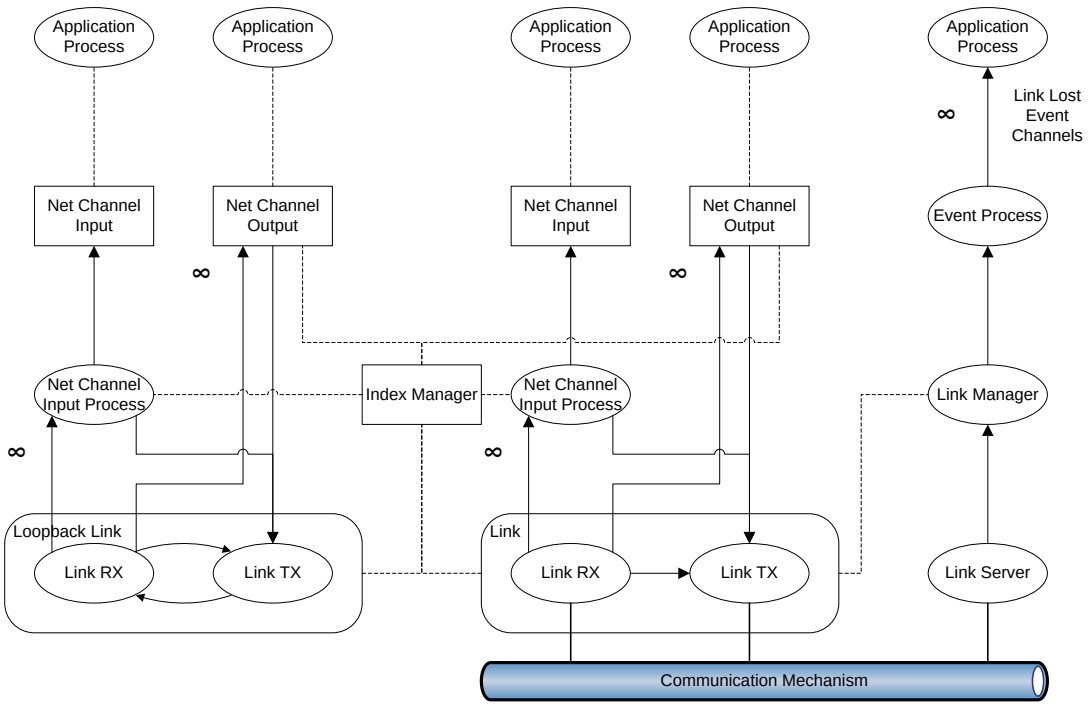


Figure 1: Current JCSP Networking Architecture

- `Link` – the `Link` component is responsible for connecting a JCSP Node (a single JVM) to another JCSP Node. The `Link` and its relevant sub-components are designed to allow operation upon any communication mechanism if the necessary addressing and connection functionality is developed. At present only TCP/IP mechanisms are provided within the JCSP Networking package. The `Link` component has two sub-components which provide input and output operations between Nodes:
  - `LinkTX` – the `LinkTX` process is responsible for transmitting messages to the remote JCSP Node. `LinkTX` has little responsibility except serialization of the sent message onto the communication output stream.
  - `LinkRX` – the `LinkRX` process is responsible for receiving messages from the remote JCSP Node. `LinkRx` interprets incoming messages and acts on the message type, accessing the destination channel if required.
- `LoopbackLink` – the `LoopbackLink` operates as a normal `Link` and provides a virtual connection within the local Node. If an output end of a channel is connected to an input end within the same JCSP Node, the message will travel through this component.
- `LinkServer` – the `LinkServer` process is responsible for receiving incoming connection requests for the Node, creating the required `Link` component to service the connection, and interacting with the `LinkManager` to control, store and mange the `Links` within the Node.
- `LinkManager` – the `LinkManager` process is responsible for managing the `Links` operating within the Node. This process ensures that only one `Link` to a given Node is active at any time, and retrieves an existing `Link` to a given Node when requested.
- `EventProcess` – the `EventProcess` is spawned by the `LinkManager` and broadcasts `LinkLost` messages to any interested process. Whenever a `Link` fails, the `Link` informs the `LinkManager` which sends a message to the `EventProcess`. The `EventProcess` writes this message to all

registered `LinkLostEventChannels`. These channels are infinitely buffered to avoid deadlock.

- `NetChannelOutput` – the `NetChannelOutput` component provides the interface to the writing end of a networked channel, and hides the underlying interactions with the `Link`. The channel receiving messages from the `Link` is infinitely buffered.
- `NetChannelInput` – the `NetChannelInput` component provides the interface to the reading end of a networked channel, although the interaction with the `Link` is handled by a separate process.
- `NetChannelInputProcess` – the `NetChannelInputProcess` services communication between the `NetChannelInput` and the `Link`. It receives messages from the `Link`, and either forwards the message to the `NetChannelInput`, or responds to the message directly. The incoming channel to the `NetChannelInputProcess` from the `Link` is infinitely buffered.
- `IndexManager` – the `IndexManager` is a shared data object which manages the networked channel ends within the Node. This component allocates index numbers (Virtual Channel Numbers) [124] to channels and allows retrieval of channel objects based on these indices.

The described components provide the application level channel functionality. Some components may have numerous instances in operation. For example, each `NetChannelInput` created has a front end and a `NetChannelInputProcess`, and each connection to a remote Node requires a `Link`. There may be multiple `LinkServer` processes if multiple interfaces or protocols are used.

The channels connecting the `Links` with the networked channel components are shared at the writing end (they are Any-2-One). This permits multiple channel ends to write to a `LinkTX`, and any `LinkRX` to receive incoming messages for any channel. A virtual channel operation can be defined as a number of component interactions, as described in the following section.

3.3 JCSP Networking Functionality

Figure 2 presents the component interactions that occur during a normal networked channel read/write operation.

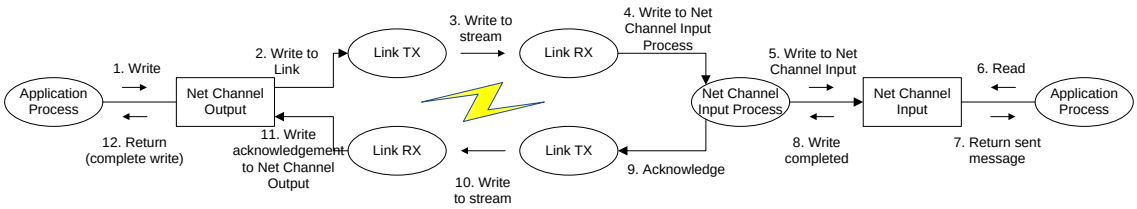


Figure 2: Networked Channel

1. An Application Process calls the `write` method on the output end of a networked channel, passing the data to be sent within the method call.
2. The `NetChannelOutput` wraps the data within a `ChannelMessage`. The `ChannelMessage` contains the destination index, source index, a flag indicating if the message should be acknowledged, and possibly the name of the channel on the remote Node. The `NetChannelOutput` contains the specific channel connected directly to the `LinkTX` and can write the `ChannelMessage` onto this channel directly.
3. The `LinkTX` reads the outgoing `ChannelMessage` from its input channel and streams it to the other Node via the communication stream. This involves serialization of the `ChannelMessage` via an `ObjectOutputStream`.
4. The receiving Node's `LinkRX` deserializes the incoming `ChannelMessage` from the connection stream, and examines the object to determine its type. For an incoming send message, the destination index is extracted and used to retrieve the channel to the `NetChannelInputProcess`. The `LinkRX` adds the channel connecting to its partner `LinkTX` process to the `ChannelMessage` to allow the `NetChannelInputProcess` to send the acknowledgement. The `ChannelMessage` is then sent to the `NetChannelInputProcess`.
5. The `NetChannelInputProcess` reads the incoming `ChannelMessage` from the `LinkRX` and sends the sent data to the `NetChannelInput`. This

is a blocking operation, and until the Application Process calls `read` on the `NetChannelInput`, the `NetChannelInputProcess` will wait.

6. The Application Process calls `read` on the `NetChannelInput`. This call may have occurred at any stage prior to this step, causing the receiving Application Process to block until now.
7. The `NetChannelInput` returns the sent data.
8. The send is completed between the `NetChannelInput` and `NetChannelInputProcess`, allowing the later to resume.
9. The `NetChannelInputProcess` creates an acknowledgement message. The destination index of the message is the source index of the original send message. The acknowledgement message is communicated to the `LinkTX` using the channel attached to the incoming `ChannelMessage` in step 4.
10. The `LinkTX` process receives the outgoing acknowledgement and serializes it over the connection stream.
11. The `LinkRX` process of the original sending Node deserializes the incoming `ChannelMessage`. As an acknowledgement message has been received, the `LinkRX` retrieves the channel to the `NetChannelOutput` from the `IndexManager` using the destination index from the `ChannelMessage`. The `ChannelMessage` is then sent to the NetChannelOutput.
12. The `NetChannelOutput` reads the acknowledgement and completes the `write` method call, allowing the writing Application Process to proceed.

This illustrates the basic read/write operation. There are a number of different message types within JCSP Networking. A brief description of these is provided next.

3.3.1 JCSP Network Message Hierarchy

The hierarchy of network messages is presented in Figure 3. The message types for networked channel operations are on the left – `ChannelMessage` and its children. The other messages are of no concern for the rest of this work. `BounceMessage` was used by `MigratableChannels` (an original implementation of mobile channels), while `PingMessage` and `PingReplyMessage` are used during initial

`Link` interactions. The `ConnectionMessages` are used by `NetConnections`, and although a consideration for the future, are not examined in detail here.

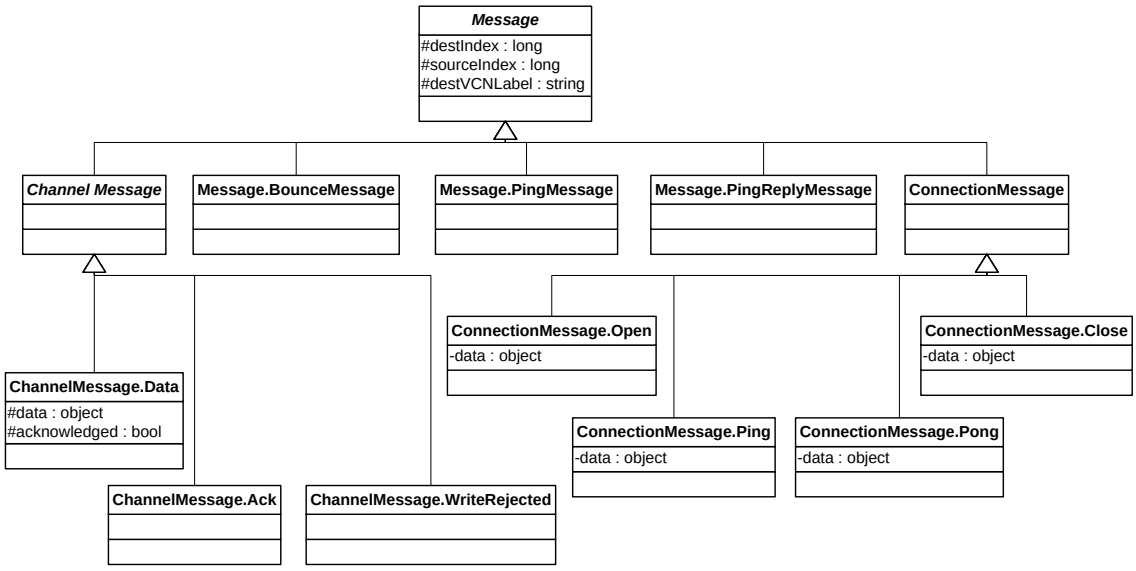


Figure 3: JCSP Network Message Hierarchy

Figure 3 helps to illustrate the amount of data sent in a `ChannelMessage`. This will be examined more fully in Chapter 4. The brief analysis in the following section focuses on previous analysis performed on JCSP Networking and similar frameworks, as well as presenting some issues based on the architecture presented thus far.

3.4 Brief Analysis of the Current Architecture

Previous research into networked architectures based on CSP has primarily focused on the performance gained from task parallelisation. In this work, the main focus is the performance of the communication mechanism, and the overheads associated with the architecture. Categorisation of some of the different CSP inspired frameworks has been previously presented [126] when considering localised systems, indicating the suitability of these frameworks within different contexts.

3.4.1 Previous Analysis on JCSP Networking

Little performance analysis of the communication mechanisms of JCSP Networking has been made. Schaller [127] examined performance of Java parallel computing libraries undertaking tasks across multiple networked Nodes. Vinter [125]

examined similar properties with other Java libraries and different tasks, but analysis was again based on parallel performance and not the communication mechanism. Kumar [128] examined JCSP performance in the context of multiplayer games, and although providing interesting results on the scalability of JCSP, little performance of the communication mechanism is provided.

3.4.2 Previous Analysis on Other Process Oriented Network Architectures

Greater analysis of communication performance has been carried out within other CSP based architectures. Brown [129] has examined latency and performance overheads in C++CSP, but no extensive testing of the communication mechanism was made. A work allocation method was used, with different packet sizes sent to remote machines for processing. Although this did lead to some information on communication performance, it does not go into enough detail to analyse the variance between standard communication and C++CSP Networked. Brown also conducted experiments for ping time, but this does not give a good indication to communication time on its own. The problem is message flow, where the acknowledgment for the original send is sent and immediately followed by the resulting ping reply. From the point of view of the pinging process, the time taken would vary little from the standard send-acknowledge cycle.

Schweigler [130] has performed extensive analysis on CPU overhead and throughput in pony, and provides comparisons to JCSP Networking. Little analysis is made of the communication mechanism in comparison to standard approaches, although effort has been made to analyse the performance of the pony networked channel. Comparisons with JCSP are made in a case study, although the main conclusions gathered are interpreted from throughput and comparison when parallelising a task.

Analysis of CSP.NET [131] provides only simplistic results thus far, without any comparison to other communication approaches. A brief comparison to JCSP Networking has been made however. The authors themselves note that the tests performed are by no means thorough enough to constitute a benchmark.

3.4.3 Resource Usage

If Figure 1 (page 34) is examined, there are a number of processes required for the network functionality. Each connection to another Node involves two processes – the `LinkTX` and `LinkRX`. Each `NetChannelInput` requires a service process (the `NetChannelInputProcess`) and the `NetChannelOutput` is lightweight in comparison. `Link` creation and management requires at least three processes: the `LinkServer`, the `LinkManager`, and the `EventProcess`, but, as previously stated, there may be multiple `LinkServer` processes in operation within a Node. The `LoopbackLink` is created when a Node is initialised, requiring two processes. Finally, although not illustrated in Figure 1, there is a process spawned with the first `NetChannelOutput`. This process is meant to inform `NetChannelOutputs` of `Link` failure, although this does not always operate as expected.

A number of temporary processes are also created during Node initialisation and subsequent `Link` connection. These processes are used to set up resources and perform connection handshaking and are subsequently killed when they have completed their task.

Therefore, there are a number of processes utilised by JCSP Networking prior to application processes being considered. As each process requires a thread to operate, it can be seen that an unconnected Node requires six threads – two for the `LoopbackLink`; a `LinkManager`; a `LinkServer`; an `EventProcess`; and the main thread. A Node connected to a Channel Name Server (used as a channel name broker) requires 11 processes – two for the new `Link`; one service process to the CNS, the service having an input and output channel; one for a `NetChannelInputProcess`; and the `NetChannelOutput` `Link` failure process.

The required processes increase as the number of `Links` and `NetChannelInputs` increases. As an example, a Node connected to five other Nodes, with ten networked input ends and an initial CNS connection will require a total of 31 processes. On resource constrained devices such as those required in Ubiquitous Computing it can be seen that JCSP Networking does not scale well.

Unfortunately, these processes are spawned without usage of the JCSP `Parallel` construct. The `Parallel` acts as a pool for created threads, and attempts to reuse threads whenever possible. As most of the processes used within JCSP Networking are created without the `Parallel`, the reclamation of resources may be slow. Many of these spawned processes are also created outside the application level, and therefore cannot easily be stopped. Methods are in place to destroy the `NetChannelInputProcess` components, but if a reference to the `NetChannelInput` is lost, then the process cannot be reclaimed and is lost.

The main reason for the heavy resource usage stems from the CSP / **occam** philosophy of using a process whenever possible. This is an ill advised approach when considering Java, particularly within resource constrained devices. A major problem is the use of a process to service a `NetChannelInput`, as this reduces the number of possible input channels into a Node.

3.4.4 Complexity

JCSP Networking is a complex architecture. One of the properties of JCSP Networking is that the architecture is removed from the underlying communication mechanism, meaning it can be implemented upon any guaranteed packet delivery protocol. The argument is that if the correct addressing and `Link` creation mechanism is provided, JCSP Networking can utilise the communication mechanism. Although this statement is true, it is difficult to achieve, requiring a great deal of knowledge of the internal architecture of JCSP Networking. Without the source code it would be difficult for a custom communication mechanism to be used.

3.4.5 Objects Only

JCSP Networking only permits serializable objects to be transferred between Nodes. In principle this is not a problem if JCSP is considered within the context of standard Java, but adds difficultly when trying to communicate with other frameworks. It would be useful to send raw data between Nodes as required, which can be done in principle as a byte array is an object in Java, but there is an overhead in the serialization. It is also a problem that not all Java platforms support serialization,

leading to difficulties when trying to implement JCSP Networking on reduced Java platforms [132].

Depending on serialization means that primitive data must be wrapped in an object prior to sending. This brings an overhead, and limits networked channels to object types only. The core JCSP package for example implements a primitive integer channel for increased performance.

3.5 Initial Observations

This chapter presented a very high level analysis of the current JCSP Networking implementation, and from this some initial observations can be made. Firstly, there has been little in depth analysis of the communication mechanism in JCSP Networking, although this is a key indicator of the overall performance of JCSP Networking. Although other frameworks have been examined in greater depth, little comparison with standard communication mechanisms has been made. Secondly, the resource requirements for JCSP Networking are high, and thus reduce scalability. Thirdly, the complexity of JCSP leads to difficulties when porting the architecture to different platforms and communication mechanisms. Finally, allowing only objects for network interactions prevents the interaction with other platforms unless they implement Java serialization.

In the following chapter, a deeper analysis of JCSP Networking is presented. For Ubiquitous Computing, there is a need to understand the properties of JCSP Networking to determine how suitable the implementation is for Ubiquitous Computing applications. Performance of networked channel communications is the main focus, with other properties examined that are relevant to JCSP Networking within the context of Ubiquitous Computing.

Chapter 4 Analysis of Current JCSP Networking

In this chapter, experimental data is presented that allows examination of the current implementation of JCSP Networking in the context of Ubiquitous Computing. The data is gathered from experiments within an environment that is restrictive enough to determine the outcome of using JCSP Networking in a relatively resource constrained manner similar to the possible scenarios envisioned for Ubiquitous Computing. The aim of the experiments is to produce metrics that allow a close approximation of the separate interactions of the individual JCSP Networking components that form the basic network channel. By doing this, it is possible to determine where any overheads occur which can be resolved. Section 4.1 describes the test framework in which the experiments are conducted, and Section 4.2 examines the two Java Virtual Machines in use. Section 4.3 provides experimental results that allow analysis of the network which allows analysis of JCSP Networking in Section 4.4. Section 4.5 examines serialization within the test framework, and Section 4.6 illustrates the overhead of JCSP Networking. Finally conclusions are drawn in Section 4.7.

4.1 Test Framework

The data presented is gathered from the interactions between a small factor device (a PDA) and a desktop PC acting as a server. Communications occur over a wireless network. Various interaction properties are examined that incorporate both raw data and objects of different sizes and complexities. This promotes insight into how well JCSP Networking compares to standard communication within the test framework. First, a description of the framework is provided.

4.1.1 PDA Specifications

The mobile device is an HP iPaq 2210, running Windows Mobile 4.2. It has 64 MBytes of memory, shared between storage and applications. The processor is Intel XScale based and operates at a maximum frequency of 400 MHz. The PDA only has Bluetooth capabilities to provide wireless communication, and therefore a SDIO wireless card has been added to provide 802.11b wireless capabilities. The wireless card is a SafeCom Technologies SDW11B and provides a 100 metre range at 11 Mbits/s bandwidth.

4.1.2 PC Specifications

The PC has a Pentium IV 3 GHz processor and 512 megabytes of memory. It is connected to the network using a standard Ethernet card to a wireless router. This provides the PC with a potential bandwidth of 100 Mbits/s. The operating system installed is Ubuntu Linux 7.10.

4.1.3 Network Specifications

The network is controlled via a wireless router – a NetGear WGR614. The wireless interface is 802.11g compatible, and potentially supports 54 Mbits/s bandwidth. The PDA restricts bandwidth to 11 Mbits/s due to its wireless interface. The wireless network does not utilise any form of security. The Ethernet interface allows 100 Mbits/s bandwidth for the PC.

As two separate interfaces are used, there are differing maximum packet sizes (Maximum Transmission Unit) in operation. The Ethernet interface has an MTU of 1500 bytes and the wireless interface 2272 bytes. The larger packets are fragmented by the router for sending on the Ethernet interface, and are then reconstructed by the PC.

4.1.4 Test Classes

A collection of classes have been developed to analyse the performance of a networked JCSP channel in comparison to Java object streams when considering object serialization. These objects vary in complexity and size to allow examination of these properties to determine if they have an effect on communication time.

Complexity relates to the number of object references sent against the number of unique objects sent. As Java serialization recreates the sent object graph, there is the possibility that the sent object contains multiple references to the same object. As this requires the usage of a lookup table, incorporating complexity allows the lookup time characteristic to be taken into account.

The definitions of the test classes are provided in Appendix B. A brief summary of the different classes is presented here:

- *Integer array* – an array of `Integer` objects. The length of the array ranges from 0 to 100.
- `TestObject` – an object that contains both an `Integer` object array and a `Double` object array. The lengths of these arrays are equal and range from 0 to 100.
- `TestObject2` – extends `TestObject`, and thus contains the `Integer` and `Double` arrays. `TestObject2` declares its own `Integer` and `Double` array. All four arrays have equal length.
- `TestObject3` – extends `TestObject` and contains its own `Integer` and `Double` arrays. However, each individual element of the `Integer` array is referenced in the partner `Integer` array, thus leading to only 100 unique `Integer` objects instead of 200. Likewise for the `Double` arrays.
- `TestObject4` – extends `TestObject`, and has the same array definitions as `TestObject3`. `TestObject4` also contains a reference to another `TestObject4` which has its own unique arrays and array elements. The second `TestObject4` references the original `TestObject4`, creating a pair of objects bound together.
- `TestObject5` – extends `TestObject`, and is similar to `TestObject4`. However, the other `TestObject5` referenced within this object has arrays which contain the same elements as this `TestObject5`. Thus there are only 100 unique `Integer` objects and 100 unique `Double` objects.

A description of Java serialization is presented in Appendix A. For clarity, the amount of data sent for each object relative to *n* (the length of the internal array(s)) is provided in Table 1. All values are in bytes.

Table 1: Test Object Sizes

Object Type	n = 0	n > 0
Integer array	41	118 + (n – 1)·10
TestObject	167	297 + (n – 1)·24
TestObject2	247	401 + (n – 1)·48
TestObject3	247	387 + (n – 1)·34
TestObject4	326	500 + (n – 1)·68
TestObject5	326	486 + (n – 1)·54

The number of references against unique objects is presented in Table 2. This information alludes to the different object complexities, and helps to determine if there is an effect on communication performance because of this complexity. Java object streams hold references to all sent/received objects (see Appendix A). Thus lookup tables are kept of all serialized objects and classes. For serialized object graphs with more unique objects, these tables will grow larger than object graphs with fewer unique objects. Serialization time should therefore increase for object graphs containing more unique objects.

Table 2: Test Object Reference Count against Unique Object Count

	n = 0		n > 0	
Object Type	**Obj Ref**	**Unique Obj**	**Obj Ref**	**Unique Obj**
Integer array	1	1	n + 1	n + 1
TestObject	3	3	3 + 2·n	3 + 2·n
TestObject2	5	5	5 + 4·n	5 + 4·n
TestObject3	5	5	5 + 4·n	5 + 2·n
TestObject4	10	10	10 + 8·n	10 + 4·n
TestObject5	10	10	10 + 8·n	10 + 2·n

These objects have been chosen as it allows examination of the serialization process itself. The largest object size will fit within the buffer JCSP has within its Link connection streams (8192 bytes). Larger data sizes are tested by sending raw data without the serialization process.

In the following section, the versions of Java used on the different devices are examined to allow a better understanding of the results presented later in this chapter. Full results can be found in Appendix D. Unless otherwise required, only the results for `TestObject4` are presented within this chapter as the size and complexity of `TestObject4` allows analysis for the majority of cases.

4.2 Examining the Java Virtual Machines

In this section, the two different JVMs are examined. The specifications of the different versions of Java are presented, and benchmarks provided to allow a closer comparison.

4.2.1 Java Versions

The two Java Virtual Machines in operation are quite different. The PC has a standard Sun Java Development Kit version 1.6 JVM. The PDA has a reduced IBM J9 JVM that conforms to the Java 2 Micro Edition (J2ME) Connected Device Configuration (CDC) Personal Profile. This provides a JVM that is approximately Java 1.3.1 compatible.

To benchmark the JVMs two methods have been used. The Java Grande Benchmark Suite [133], although designed to benchmark JVMs in the context of high performance computing, provides a number of tests that allow comparison of the two JVMs. The second method is aimed at the JCSP implementation specifically by performing standard benchmarks used to evaluate performance of CSP based frameworks. A comparison of the two JVMs is provided in Appendix C. In general, the PDA operates between 1 and 2.5 orders of magnitude slower than the PC in these tests. The variance between the different result sets will be largely due to the PC having faster I/O and having specialised machine instructions for some operations.

There are some benchmarks that are relevant to the discussion of the performance of JCSP Networking in the context of the experiments that have been performed. The Java Grande Suite provides object creation and serialization benchmarks, and

the CommsTime benchmark in JCSP provides an approximation of the channel communication time between two processes.

4.2.2 Java Grande Object Creation Benchmarks

The Java Grande Suite object creation benchmarks are performed on small objects with certain properties, such as internal fields and sub-classing. The results are based on allocation time, which does not allow enough insight into the time taken to create the various object types being examined in this chapter. Therefore, the operation of the benchmark is replicated using the test classes. The results for the PC are presented in Figure 4 and the results for the PDA in Figure 5. The values represented are the average time taken to create a single object of the given size (along the x-axis) and type in milliseconds.

The creation time for the PC is almost negligible, the largest object taking approximately 11 microseconds to create. The PDA performs approximately two orders of magnitude slower than the PC. The Java Grande benchmark for object creation (see Figure 80 in Appendix C) shows the PDA performing one order of magnitude poorer than the PC with more complicated objects increasing this variance. The increase from 1 to 2 orders of magnitude variance between the PC and PDA can be attributed to memory allocation as the test objects are large in comparison to the small objects tested using the normal Java Grande benchmark.

These results indicate that memory allocation time has the greatest impact on object creation. `TestObject2` and `TestObject5` are approximately equal in size for n < 100 (see Table 1, page 47), and take approximately the same time to create. `TestObject4`, which is less complex than `TestObject5`, takes a greater time to create, and `TestObject3`, which is more complex than `TestObject2`, takes less time to create. Object complexity due to the number of references appears to have little effect on object creation time.

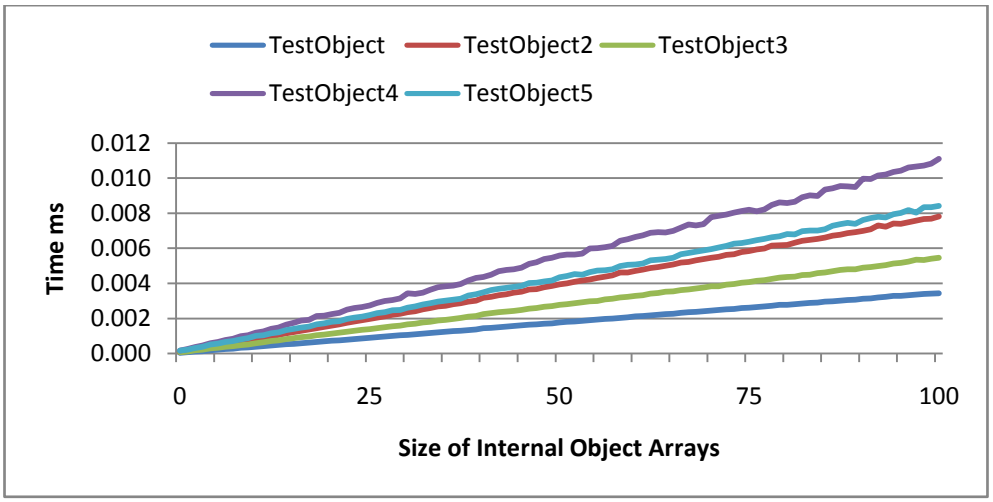


Figure 4: PC Test Object Creation Times

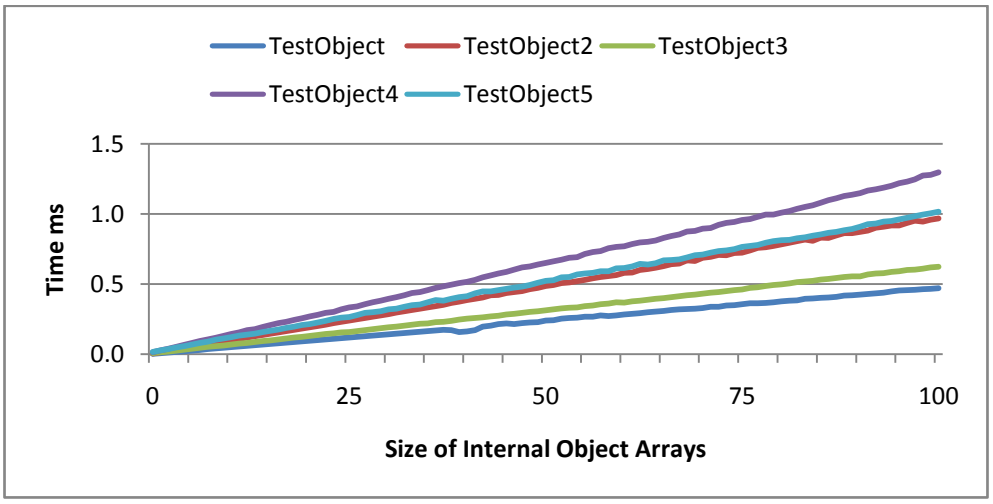


Figure 5: PDA Test Object Creation Times

4.2.3 Java Grande Object Serialization Benchmarks

The Java Grande Suite provides object serialization benchmarks based on data structures of various sizes and complexities. This test can be modified to operate on the test classes. The Java Grande serialization benchmark does not reset its streams after every object write operation, so the test is modified to reset the streams after every communication. Within JCSP Networking, this is done to avoid aliasing problems upon the object stream (see Appendix A). Therefore, this is replicated within the serialization benchmarks. The standard Java Grande serialization benchmark writes the serialized object to file.

Figure 6 presents the results for the PC performing the Java Grande serialization benchmark with the test objects. The results represent the average time in

milliseconds taken to serialize a single object of the given type, with the x-axis ranging over the length of the object's array(s).

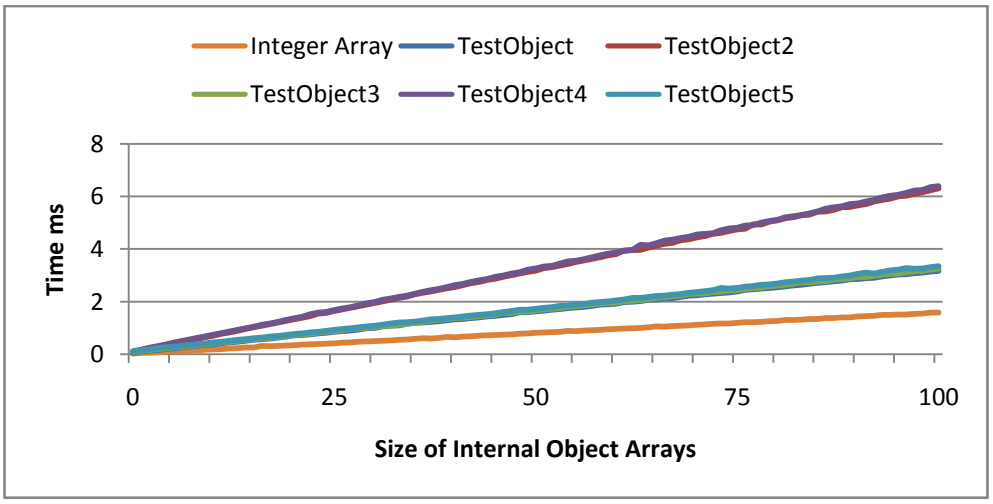


Figure 6: PC Java Grande Test Object Serialization

The interesting phenomenon where the lines for `TestObject2` and `TestObject4` increase in unison and likewise for `TestObject`, `TestObject3` and `TestObject5` can be attributed to the object complexities defined in Table 2 (page 47). `TestObject2` and `TestObject4` increase the number of unique objects by a factor of 4 relevant to *n*. The other object types (aside from the `Integer` array) increase by a factor of 2. Therefore, for the Java Grande serialization benchmark, the lookup table increasing in size does have an effect on performance.

Figure 7 presents the results for the PDA performing serialization on the test objects. The lines are not grouped based on the number of unique objects and the interesting phenomenon is that `TestObject2` and `TestObject5` are closely grouped. These objects are approximately equal in size for $n < 100$. Therefore, for the PDA the I/O time associated with writing the serialized objects to file has a significant effect.

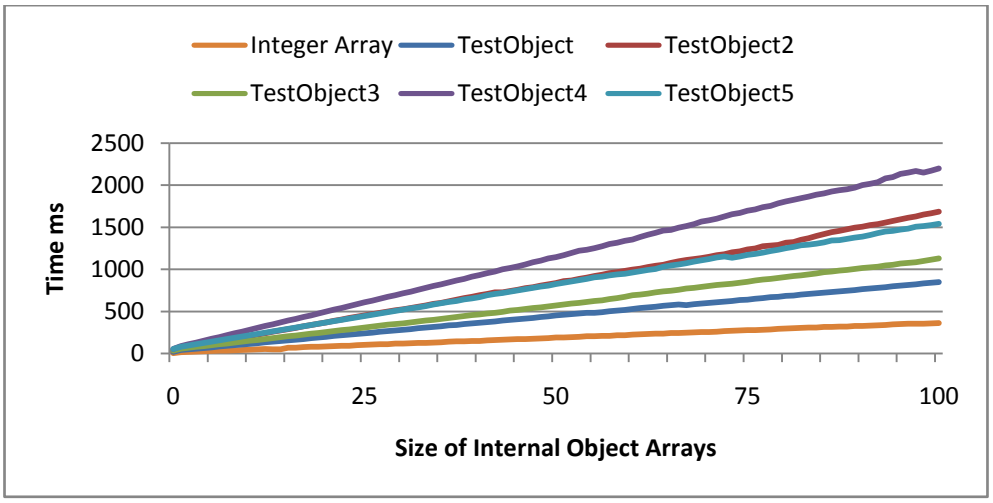


Figure 7: PDA Java Grande Test Object Serialization

Figure 8 compares the performance of the PDA and PC for (de)serialization of `TestObject4`. The results present the average time in milliseconds to (de)serialize a single `TestObject4`. The x-axis represents the object size in bytes as generated from Table 1. These results show the PC performs approximately 2.5 orders of magnitude faster than the PDA, which concurs with the general performance difference of the two devices. It is also of note that deserialization is faster than serialization on the PDA, while the converse is initially true for the PC. This may be due to the extra lookup required for each serialization of an object prior to it being written, whereas the deserialization process only performs a lookup when prompted to by a reference signal appearing on the stream. The PDA may also have slower file output performance that input performance.

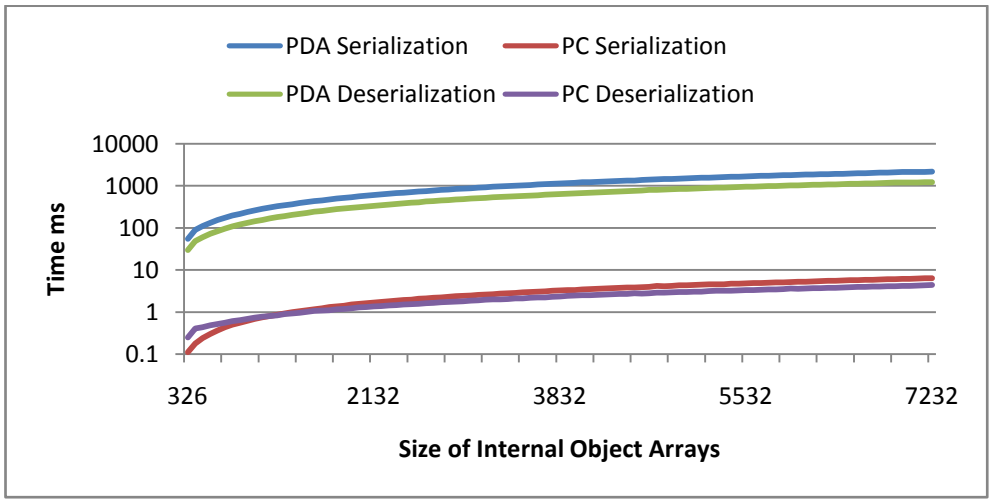


Figure 8: PC against PDA TestObject4 Java Grande (De)Serialization Benchmark

The results presented give some indication to the performance of the (de)serialization process on the different devices, but they also incorporate file I/O time. For JCSP network communications, objects are (de)serialized within a memory buffer, which incurs a lower I/O overhead. Therefore, experiments involving (de)serialization within memory are performed.

4.2.4 Serialization within Memory

The Java Grande serialization benchmark can be modified to use memory streams instead of a file stream. This operation is generally fast, and would require a greater number of operations within a timed cycle of operations to avoid noise within the results. The available memory restricts this possibility, as the memory stream must be declared prior to any timed operations. To avoid the buffer within the stream requiring expansion during the timed cycle, a 10 million byte allocation within the PC and a 1 million byte allocation within the PDA is used. This restricts the maximum number of operations in a timed cycle to 1000 and 100 for the PC and PDA respectively.

Figure 9 presents the results from the PC performing serialization into memory. Unlike the file based serialization, the lines are more separated. `TestObject2` and `TestObject4` are still rising close to uniformly. Both of these object types increase in object size at different rates and therefore the amount of data is not the major factor in their close proximity at these data ranges.

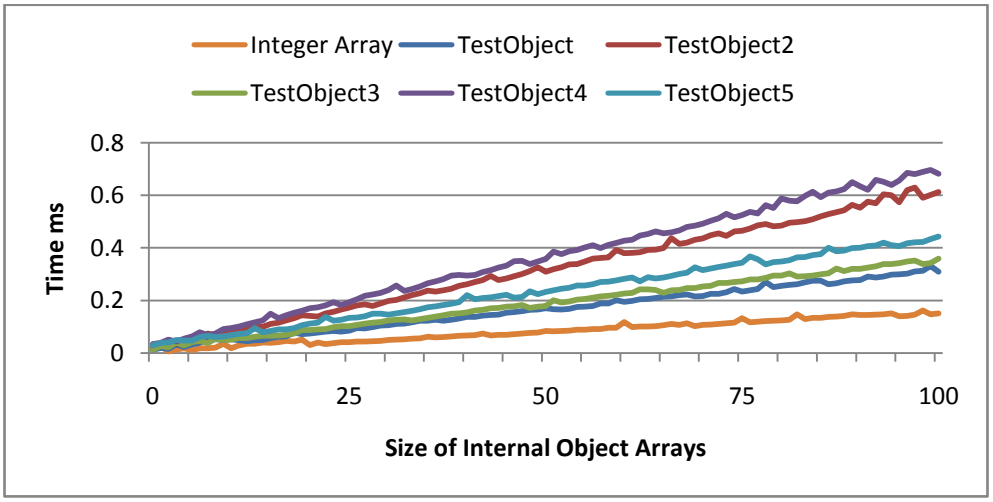


Figure 9: PC Memory Test Object Serialization

Figure 10 presents the results for the PDA serializing the various test classes into memory. Unlike file based serialization, the PDA has `TestObject2` and `TestObject4` grouped together, and `TestObject`, `TestObject3` and `TestObject5` grouped together. These results show the same object complexity and lookup attributes as the PC, with time increase based on the number of unique objects as opposed to I/O throughput.

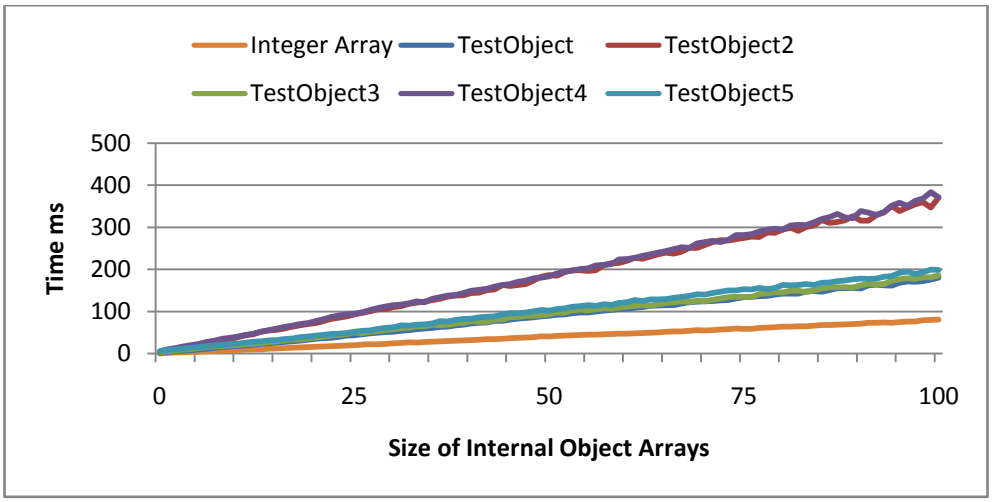


Figure 10: PDA Memory Test Object Serialization

As JCSP Networking initially serializes objects directly into memory, the number of unique objects has a greater effect on serialization time than object size when considering these test classes. JCSP Networking resets its object streams after every `Link` communication, which leads to more data being sent within a single transaction due to class information requiring transmission every time. However, it would appear that the reduction in lookup time does provide increased performance, and this may overcome the increase in transferred data.

Figure 11 presents the PC against the PDA for (de)serialization of `TestObject4` within memory. Figure 11 is similar to Figure 8, and an approximate 2.5 order of magnitude difference between the two devices is still evident. This is despite the different trends seen for the PDA for memory based serialization. It is also of note that the difference between serialization and deserialization is smaller than the file based serialization operations for the PDA, and that serialization is now faster than deserialization on the PDA at larger sizes of `TestObject4`.

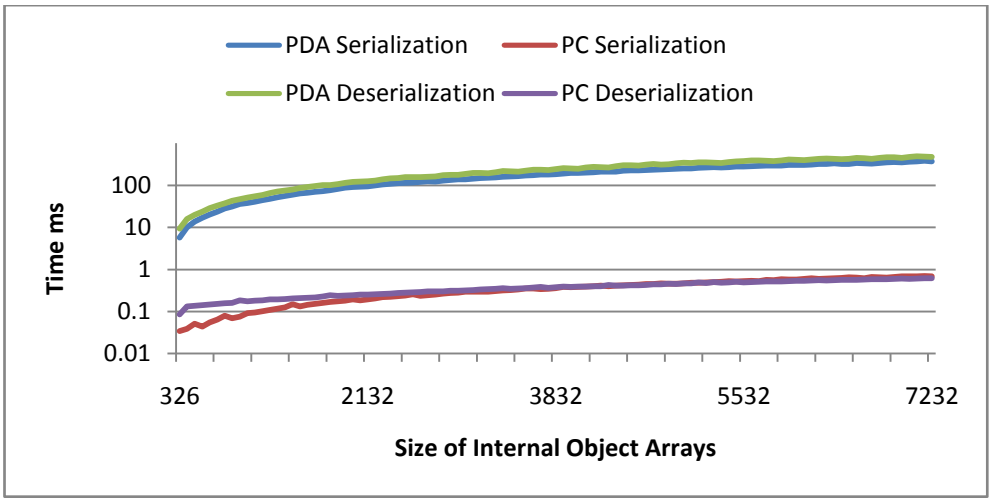


Figure 11: PC against PDA TestObject4 Memory (De)Serialization Benchmark

4.2.5 CommsTime

The CommsTime benchmark is a common mechanism for measuring channel communication performance in a CSP based framework [134]. The PDA has a CommsTime figure of approximately 720 microseconds and the PC approximately 62 microseconds for object based channels (see C.6.1). As CommsTime incorporates a system with four channel communications, the channel communication time for the PDA can be approximated at 180 microseconds, and the PC at 15 microseconds. The PC has approximately a one order of magnitude faster channel communication time.

4.3 Examining the Network Performance

In this section, the network infrastructure used in the test framework is examined. There are a number of different properties of concern. The effect of the JVM on the PDA networking performance is a factor, and therefore the underlying native network libraries (Winsock) are tested. A comparison between normal Java network streams and Java object streams on the network is required to establish the overhead from using object based streams. Finally, there is the buffered stream that has been placed within the JCSP `Link` and thus object streams require buffering within these experiments. The buffer is set at 8192 bytes, the size of the buffer internal to JCSP Networking.

There are two standard properties to measure network performance – latency and bandwidth. A ping test allows analysis of latency, which is the overhead of the communication compared to expected results, and bandwidth allows an approximation of throughput. Various sizes of byte array are passed between the two devices to evaluate the network properties. The smallest possible data size to send via a network stream is a single byte. The smallest data size for an object stream is a null value, which also takes up a single byte.

4.3.1 Simple Ping

To determine the basic send-acknowledge operation, a simple ping test is used. A single byte or null object is sent from one device to the other and back. Each operation is carried out 10,000 times within a timed cycle, and ten timed cycles are performed. The ten times are trimmed to six by removing the top two and bottom two values, and the mean of the six median values calculated. Times are gathered from both devices for when the PC pings the PDA (PC to PDA), and when the PDA pings the PC (PDA to PC), using both network streams and object streams. Tests are repeated 2 to 3 times to ensure consistency, and one set of the results chosen for representation. All individual results of the median six are within twenty percent variance of the trimmed mean.

JCSP networking has the Nagle algorithm switched off for underlying TCP/IP network connections. The Nagle algorithm increases performance by buffering outgoing messages until either an entire packet of data is ready for transfer, or the previous packet is acknowledged. By default, Nagle is turned on, and turning it off is ill advised. Therefore, for the simple ping test, sockets with Nagle on and off are examined. The usage of Nagle highlights the reason to use 10,000 operations in a timed cycle. For 1,000 operations, the amount of data would fit into a single packet on both interfaces, and would therefore give the Nagle based results significant improvement. Therefore, the number of operations is increased by an order of magnitude. Figure 12 presents the results of the simple ping test. The values are the average times in milliseconds to perform a single ping-pong operation using the various communication mechanisms.

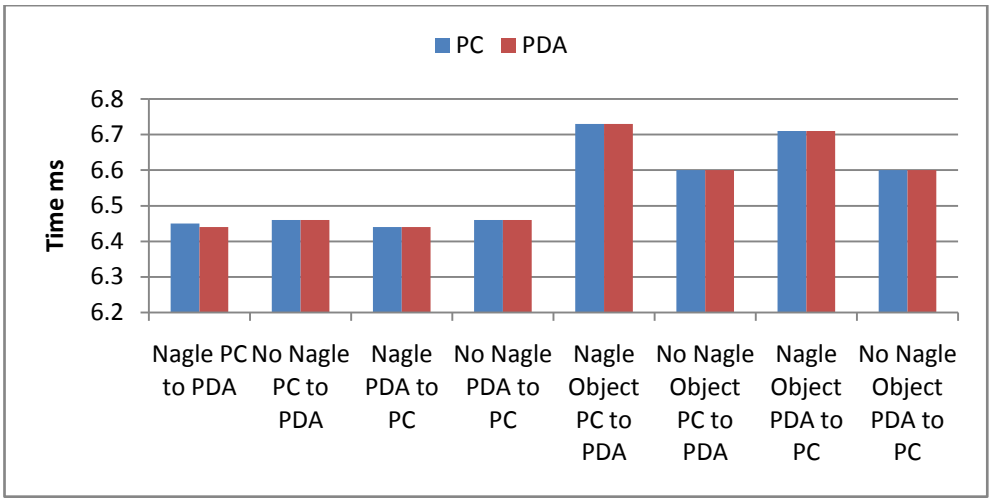


Figure 12: Simple Ping Test

For standard networked streams, it can be seen that performance is slightly better with Nagle turned on, and the time for PC to PDA has no large variance from the PDA to PC. For object streams, having Nagle turned off appears to improve performance by 100 microseconds per message. For an object stream, there is a 150 to 250 microsecond overhead, which is probably due to the encoding and decoding of the null value on the stream, and the examination process required to determine the object type during sending.

4.3.2 Bandwidth

Small packet send time can be determined by a device sending a single byte to the other and gathering an average time for this operation from both the sending and receiving device. These results are presented in Figure 13. As for the ping test, the time taken to perform 10,000 operations is gathered ten times, and the median six used to calculate the mean time to send a packet.

Figure 13 indicates that network streams have no significant improvement for having Nagle on or off. Of interest is the time the PC takes to send a small message; approximately 1.5 microseconds. The PDA takes 5 milliseconds, which is 3.5 orders of magnitude greater. The PDA records a time of 240 microseconds to receive from the PC, 2.5 orders of magnitude greater than the time for the PC to send the message. Thus, it can be determined that the greatest bottleneck for small packet sizes in the test framework is the PDA sending data to the PC. All JCSP network

communications require acknowledgement, which is a relatively small message size of 207 bytes. It can be estimated that that the PDA takes at least 5 milliseconds to send an acknowledgement to the PC. The PC acknowledgement time is close to insignificant in comparison.

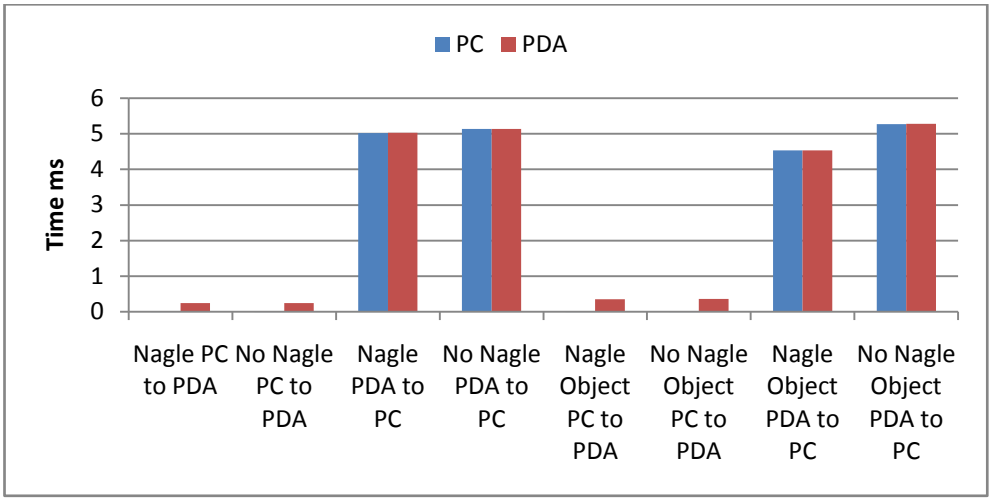


Figure 13: Send and Receive Benchmark

Object streams show a similar performance difference, although the results are better with Nagle than without. This contradicts the ping time test. The nature of the Nagle algorithm means that ping tests are not well suited as no buffering will occur for sent messages. JCSP network send messages are acknowledged, and therefore Nagle may not cause an increase in performance. Performance would therefore be determined by how many channels are serviced by a specific `Link`.

To determine actual bandwidth, different byte array sizes are transferred between the PDA and PC. Sizes range from 10^3 to 10^6 bytes. Each array is sent ten times in a timed cycle, and ten timed cycles performed. The six median values are taken, and the mean time calculated from them. Figure 14 presents the PDA sending data via Java network streams, native network streams, and networked object streams. The results represent throughput in bytes per millisecond achieved with the different data sizes.

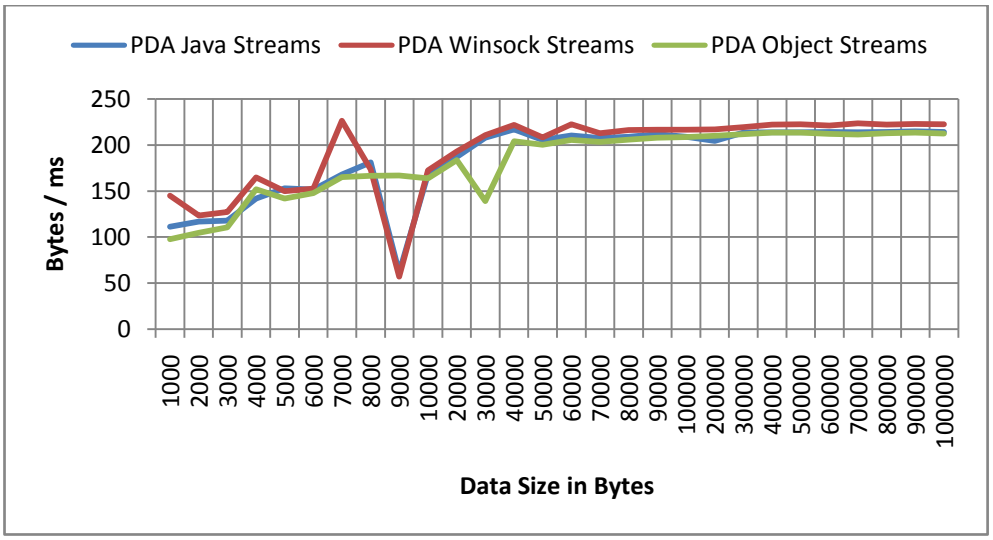


Figure 14: PDA Bandwidth

From Figure 14, it can be determined that the PDA can transmit data between 100 and 225 bytes per millisecond. There are three interesting points. At data size 7,000, the native streams appear to perform better than Java network and object streams – sending the data packet 10 ms faster than the other two mechanisms. The reason for this apparent performance increase is unclear.

At 9,000 bytes, both native and Java streams dip in performance, whereas the object streams do not. The reason for this occurring has not been fully determined, although repetition within native and Java results points to an issue with the PC, the PDA hardware, or the network infrastructure. Further analysis of this phenomenon is presented in Section 4.5.1. The reason the object stream does not exhibit this property is due to the extra data sent due to serialization. A primitive array object has 23 bytes of serialization information, and this is enough to negate the performance drop. The difference in the actual data packet size causes a similar drop in performance for the object streams at 30,000 bytes.

Figure 15 presents the results for the PC sending to the PDA. These results show the PC performs better for small packet sizes, which is the converse to the PDA. The result for data size 1,000 is not shown as this gives bandwidth in excess of 60,000 bytes/ms, and would not permit the detail present in Figure 15. From these results, it can be determined that the PC can output data onto the network between 800 and 300 bytes/ms.

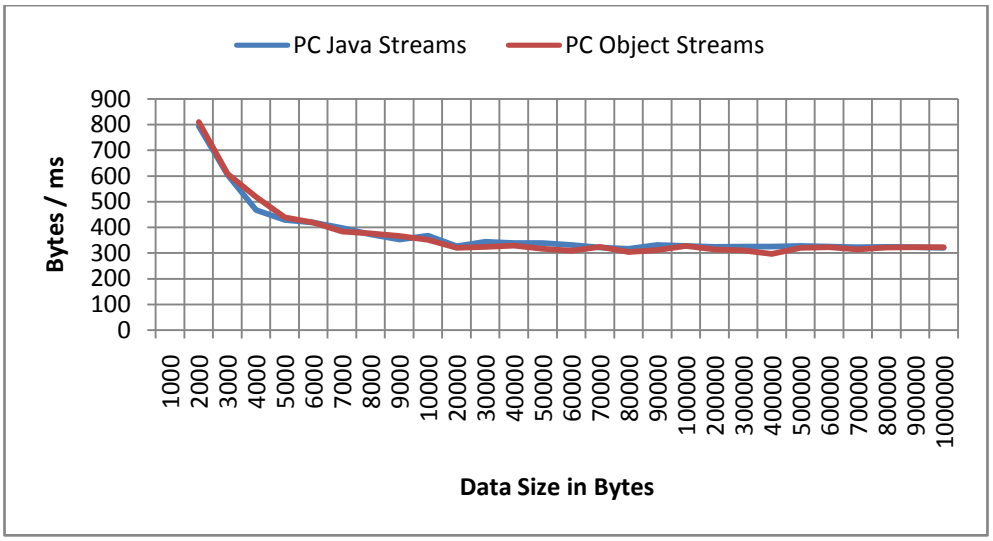


Figure 15: PC Bandwidth

4.3.3 Latency

Latency refers to the overhead time taken for a message to travel from one device to another beyond expected time, and includes any encoding and decoding of messages. A simple method to determine latency is to perform a roundtrip (ping-pong) message between the two devices and remove the time it should take for the two devices to send data to one another. From the send and receive benchmark (Figure 13 – page 58) and ping test (Figure 12 – page 57), it is possible to determine latency of approximately 1.5 ms for a roundtrip message on a network stream, and approximately 1.7 ms on an object stream for small message sizes. For a more thorough examination, the bandwidth benchmark is repeated using roundtrip operations.

Figure 16 presents the results for the PDA sending to and then receiving from the PC. The results presented are the time in milliseconds taken to perform a single roundtrip operation. The expected times are calculated by adding the time taken for the send from the PDA to the PC and the send from the PC to the PDA. Native Winsock streams perform approximately as the standard Java streams in this experiment and are not presented. These results are given in Appendix D.

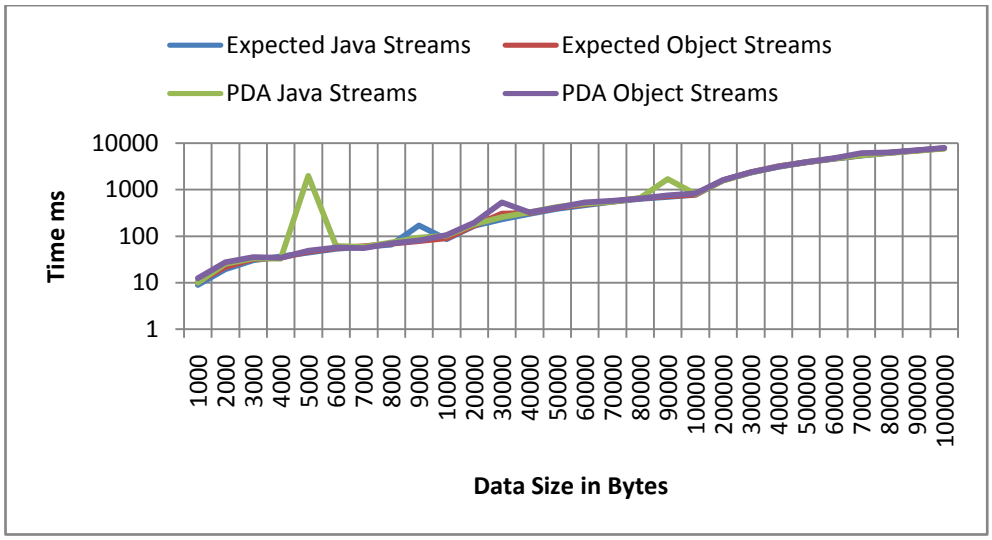


Figure 16: Roundtrip PDA to PC

At certain data sizes within Figure 16 performance drops during the roundtrip operation. The data sizes of these drops (5,000 and 90,000) are different from the performance drop for only sending data (9,000 bytes). The performance drop at 9,000 bytes has actually disappeared. This leads to further evidence that a network centric or device centric issue is causing the drop in performance.

Object streams have the same performance drop at 30,000 bytes, and the actual result has a larger peak than expected. It is unlikely that a serialization issue is causing this drop, as after the initial header information for a serialized array, performance is based on I/O throughput on the number of bytes.

Figure 17 presents the roundtrip time for the PC sending to the PDA. A comparison of Figure 16 and Figure 17 show that the results for PDA to PC and PC to PDA to be approximately equal, with the same points of poor performance.

From the roundtrip results, actual roundtrip time and estimated roundtrip time are approximately equal, particularly for large data sizes. Excluding the peaks, only smaller packet sizes have latency times noticeable in relation to time taken.

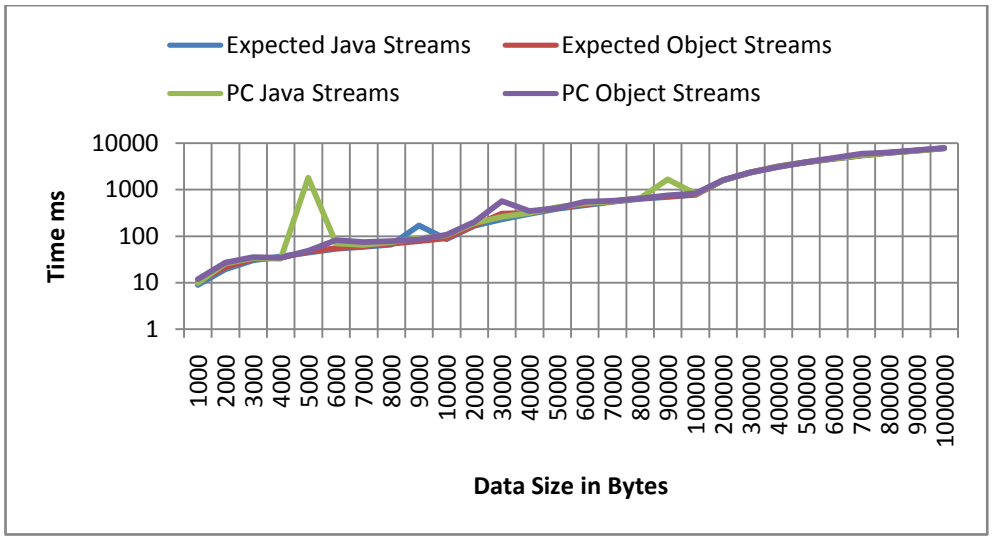


Figure 17: Roundtrip PC to PDA

The network experiments presented within this section show that there is little performance overhead incurred by the PDA JVM with respect to network bandwidth, and that latency is generally low. There are some unexplained performance issues at certain data sizes, but these are probably due to device or network problems as opposed to anything from the JVM on the PDA. The deactivating of the Nagle algorithm in JCSP may be an issue for performance, but considering the send-acknowledge communication of network channels, this may not strictly be true.

4.4 Examining JCSP Performance

With the information from sections 4.2 and 4.3 it is possible to estimate the expected performance of JCSP Networking for sending the specified test objects. If Figure 2 (page 37) is examined, there are eleven operations that have measured values:

1. `NetChannelOutput` writes to the `Link` (channel communication)
2. `Link` serializes sent object message (data plus 249 byte message header overhead – see Appendix A).
3. `Link` transmits the data to the remote `Link`
4. Remote `Link` deserializes the object message
5. `Link` writes the object message to the `NetChannelInputProcess` (channel communication)

6. `NetChannelInputProcess` writes the object to the `NetChannelInput` (channel communication)
7. `NetChannelInputProcess` writes the acknowledgement to the `Link` (channel communication)
8. `Link` serializes the acknowledgement message
9. `Link` transmits the acknowledgement message to the remote `Link`
10. Remote `Link` deserializes the acknowledgement message
11. `Link` writes acknowledgment to the `NetChannelOutput` (channel communication)

A virtual networked channel will have one end on the PDA and one end on the PC, and therefore the two channel times will be different. Separating the above interactions into output operations and input operations allows values to be entered into a performance calculation. Thus, there is a formula for channel output time and a formula for channel input time. *chan* represents channel communication time on the device, and the size in bytes of the ack message is 204 bytes, and a send message incurs a 249 byte overhead for the header:

$$C_{out} = 2 \cdot chan + serialize(sizeof(message) + 249) + deserialize(ack)$$

$$C_{in} = 3 \cdot chan + deserialize(sizeof(message) + 249) + serialize(ack)$$

The total time to communicate from across the channel is:

$$NetChan = C_{out} + C_{in} + transmit(sizeof(message) + 249) + transmit(ack)$$

Transmission time is separated as it relies on sender throughput and receiver throughput independently, and the network infrastructure. From Sections 4.2 and 4.3 approximate values for the properties of interest can be given. These values are presented in Table 3.

Table 3: Communication Properties

	Channel (ms)	Serial Small (bytes/ms)	Serial (bytes/ms)	Deserial Small (bytes/ms)	Deserial (bytes/ms)	Min Transfer Time (ms)	Transfer Throughput (bytes/ms)
PC	0.015	20000	10000	2000	10000	0.001	320
PDA	0.18	80	20	40	15	5	215

Channel is the channel communication time gained from the CommsTime benchmark. Serial Small is the approximate bytes/ms serialization performance for small objects, and likewise Deserial Small for deserialization. Serial is the approximate throughput for serialization, and is calculated from the approximate value for byte throughput when serializing the test objects at large sizes, gathering the mean of these six values and rounding up to one significant digit. Deserial is likewise calculated, but the PDA deserialization value has two significant digits due to the closer proximity to 15 and the relatively small value. Min Transfer Time is the send time for small messages gathered from Figure 13. Finally, throughput is the approximate bandwidth values presented in Figure 14 and Figure 15 (pages 59and 60 respectively).

JCSP networking has two network channel types: acknowledged synchronous and asynchronous without acknowledgement. The latter channel type is used to implement server type connections, where a channel requests a message from a server and the server responds. The unacknowledged channel would appear to be an attempt to circumvent the poor exception handling in JCSP networking, which could cause deadlock on the server if a connection failed [135]. If a server replies asynchronously, there is no issue. If used for standard communication, the infinite buffering in the underlying channel can cause a problem when no synchronization occurs. Unacknowledged channels are tested to ascertain whether they can lead to further insight into the overhead of network channel communication caused by the acknowledgement signal.

To evaluate the network performance overhead of JCSP Networking, the ping, bandwidth and latency tests are repeated. The results from these experiments are presented in the following subsections.

4.4.1 Simple Ping

It is possible to estimate the ping time for a JCSP networked channel based on the communication formula *NetChan*. As object streams are in operation within the JCSP Networking architecture, the smallest data value to send over a networked channel is null. With the 249 byte message header taken into consideration, this

provides a 250 byte sent data object, which is considered a small message. Likewise the 204 byte acknowledgement message is considered small. Applying known values to the *NetChan* formula provides the estimated overhead for JCSP network channels within the test framework. These values are presented in Table 4. The *NetChan* values are those for the specified device outputting to the other. All times are in milliseconds.

Table 4: Net Channel Overhead

	C_{out}	C_{in}	NetChan
PC Sync	0.1445	0.1802	14.4855
PC Async	0.0275	0.155	6.6385
PDA Sync	8.585	9.34	13.7662
PDA Async	3.305	6.61	8.46

As asynchronous channels are examined, these values are also calculated. Removing a channel communication and the (de)serialization and transfer time for the acknowledgement provides these values. With the calculated approximate channel communication values, it is possible to evaluate roundtrip time on small messages. These results are presented in Figure 18. The Object Streams values are taken from the No Nagle results in Figure 12.

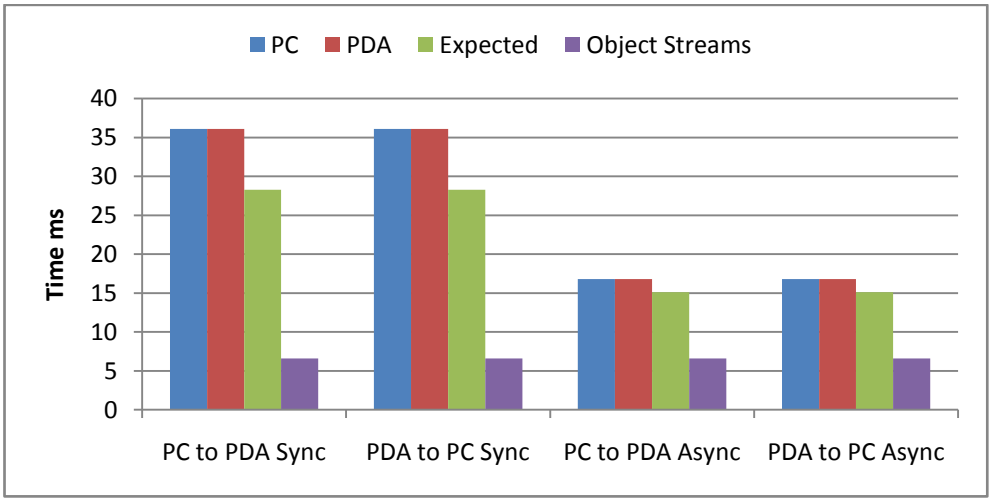


Figure 18: JCSP Network Channel Ping Test

For null messages JCSP Networking is six times slower for a ping operation than Object Streams when using synchronous channels. This is a significant overhead, attributed to the extra information required for channel messages and the

synchronisation during the send. Asynchronous channels are under three times slower than the Object Stream results, and this will be largely due to the message header overhead.

Actual values are greater than the expected values – by 8 ms for synchronous channels and 2 ms for asynchronous channels. There is the extra overhead for lookup time with the `IndexManager` when an incoming message is received to consider. There is also the underlying latency of the network architecture which is 1.5 ms for a roundtrip message. This will be doubled for synchronous channels as two send and acknowledge interactions are occurring for each roundtrip. Another consideration is actual (de)serialization time, which may be greater than the estimated value due to the number of properties within a JCSP Networking channel message.

The asynchronous results are below half the time for synchronous channels. This does point to good performance benefits for having asynchronous message passing within JCSP Networking, but the inherent danger due to the infinite buffering within the underlying channel requires care.

4.4.2 Bandwidth

As Section 4.3.2, both the time to send the smallest possible message (null) and the time to send byte arrays of various sizes are gathered within JCSP Networking. Small message passing results, with Object Stream and Expected results for comparison, are presented in Figure 19.

In this case, Expected results are approximately 2 ms better than actual results. When latency is considered, Expected and actual values are approximately equal. From the synchronised channel results, the estimated ping time should be 31.5 ms. The actual result is 36 ms in Figure 18, and therefore an approximate latency of 4.5 ms is present for a JCSP Networking channel roundtrip communication in the test framework.

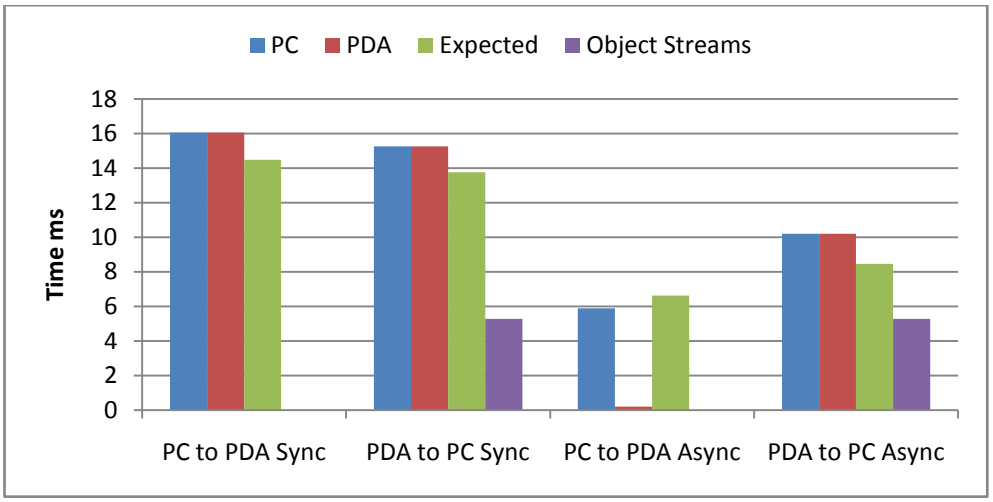


Figure 19: JCSP Network Channel Send and Receive Benchmark

The Asynchronous results for the PC to PDA help illustrate a problem with the underlying `Link` processes in JCSP Networking, which will be analysed further in Section 4.6. The `Link` processes are given maximum priority within JCSP Networking to enable communication to start quickly and data transfer to be serviced quickly. The usage of high priority is due to the aim of JCSP Networking for cluster computing scenarios where the computation time to communication time ratio is high. However, this usage of high priority can lead to a problem when a slow device is flooded by large data packets sent from a faster device. Therefore the PDA appears to take no time to receive messages from the PC asynchronously as the lower priority application process cannot start the timer while the PC effectively floods the device with data. However, unacknowledged channels should not really be used in this manner due to the infinite buffering issue, and their existence in JCSP is questionable.

The Expected time to send a message asynchronously from the PC to PDA is greater than the actual time, and is due to the channel being able to continuously write to the `Link` to send a message without blocking. The `Link` is responsible for I/O and therefore the PC application does not register this time fully within its asynchronous results.

To gather throughput information for JCSP Networking, the bandwidth experiments are repeated with the synchronous and asynchronous channels. The expected results are calculated using the *NetChan* formula with the properties in Table 3. As

the bytes within the array are not serialized, the (de)serialization overhead is calculated as the message header plus the 23 byte array description. Figure 20 presents the bandwidth results for the PDA for synchronous and asynchronous channels, with the expected results also given.

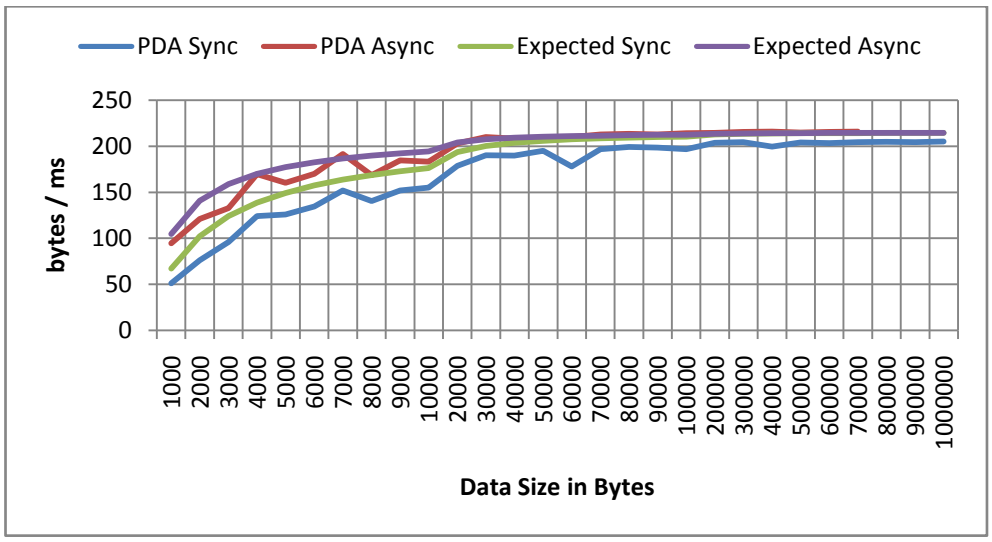


Figure 20: PDA Channel Bandwidth

Synchronous bandwidth is lower than expected, and levels at 205 bytes/ms. This value is 10 bytes/ms lower than expected. For asynchronous channels, bandwidth is as expected. This suggests that synchronisation inflicts an approximate 10 bytes/ms overhead within the test framework at large data sizes. The asynchronous results only reach $7x10^5$ bytes, as after this point the PDA cannot handle the amount of data being pushed towards it and fails with a memory exception. This is the result of the high priority `Link` problem, as all the sent information requires buffering which is obviously limited on the PDA.

The channel bandwidth for the PC is presented in Figure 21. Expected results are provided based on the known properties and the *NetChan* formula.

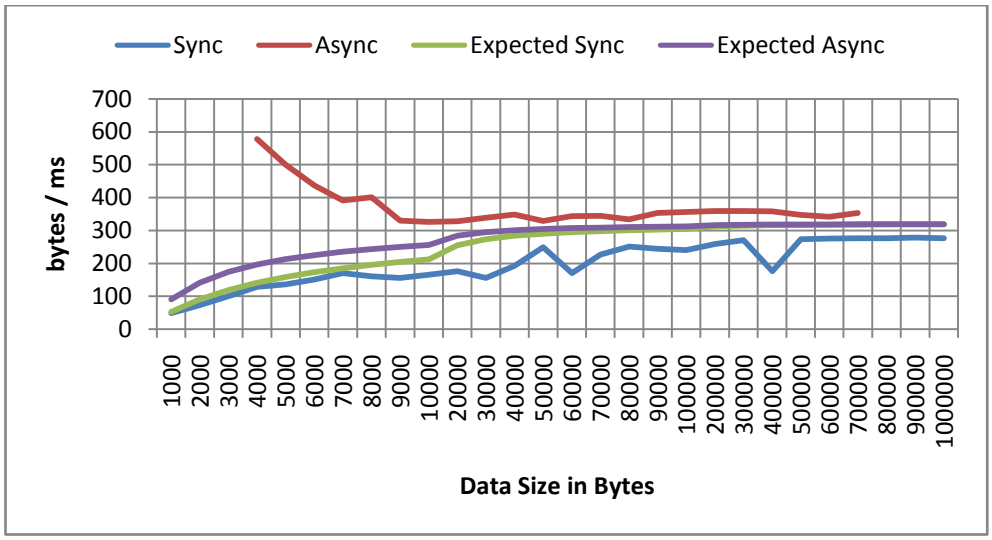


Figure 21: PC Channel Bandwidth

For the PC, the throughput for sending small (1,000 to 3,000) byte arrays asynchronously is not presented on the chart due to the significant large value when compared to the other communication results. These values give throughput of up to 60,000 bytes/ms. The asynchronous results have greater throughput than expected, due to the application level channel object outputting to the `Link` and not waiting for the actual I/O to occur. As the PDA cannot accept the amount of data pushed at it by the PC, the PC results also only reach $7x10^5$ bytes.

Synchronous channels initially provide close to expected performance but drop below expected results for data sizes above 8,000 bytes. Performance levels at 275 bytes/ms, which is 45 bytes/ms lower than expected. This will be largely due to the expected calculations not considering the time for the PDA to input data, which is greater than the time taken for the PC to output data. Receive time data is provided in Appendix D.

4.4.3 Latency

The final property to examine within JCSP networking is latency. The roundtrip experiments conducted on network and object streams are repeated with both synchronous and asynchronous channels. The expected results are calculated from the properties in Table 3 and the *NetChan* formula. As the results presented are similar for both PDA to PC and PC to PDA, only the former results are presented.

Other data can be found in Appendix D for comparison. Figure 22 presents the channel roundtrip results.

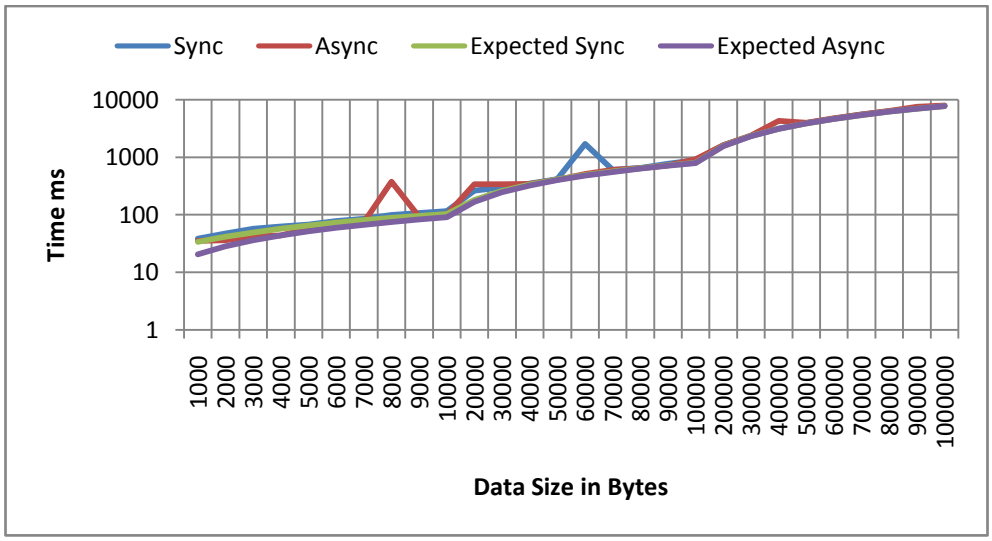


Figure 22: Channel Roundtrip PDA to PC

There is some variance between expected results and actual results. Excluding the peaks within the Sync results, the mean latency is approximately 25 ms when actual results are compared to expected results. For asynchronous channels, performance is initially better than synchronous channels, but does reduce at higher values. Figure 23 illustrates the variance between actual and expected results for the roundtrip time in milliseconds, with the significant peaks removed, and subsequent adjoining points connected.

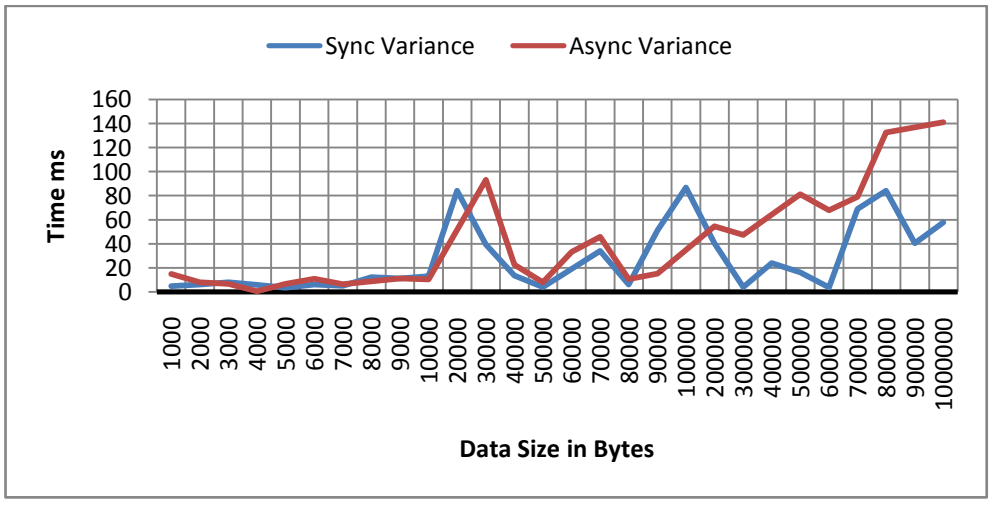


Figure 23: Variance between Actual and Expected Channel Roundtrip Results

As Figure 23 illustrates, Asynchronous channels degrade in performance over time. As the same underlying channel mechanism is used for both acknowledged and unacknowledged channels, these results do seem to indicate that any initial benefit for small message sizes sent asynchronously is balanced by poorer performance for large data sizes. From the expected values, asynchronous channels should have a 14 ms lower roundtrip time. Therefore, subtracting 14 ms from the values presented in Figure 23 indicates that asynchronous channels have what could be considered a severe roundtrip overhead in the test architecture for large packet sizes. As the results presented are similar for both PDA to PC and PC to PDA, and the PDA starts its timer before initiating the roundtrip operation, the variance cannot be due to the application process being unable to start its timer before the PC sends data. After each timed cycle, a handshake is also performed to ensure that the PDA is not flooded with the next cycle of data from the PC. The variance is therefore not the fault of the high priority `Link` processes.

From the results presented in this section, it can be ascertained that JCSP Networking does have some communication overhead, particularly for small message sizes. For large message sizes, channel bandwidth is not far removed from that of Java object streams. Most of the overhead thus far can be attributed to the message header that requires serialization, and the acknowledgement message. In the following section, serialization is examined in greater detail by comparing JCSP and object streams for sending the various test objects.

4.5 Test Object Messages

To examine serialization and the effect serialization has on JCSP Networking, the various test classes are subjected to the sending and roundtrip experiments that raw data messages were subjected to. These experiments operate upon the various test objects with sizes ranging from 0 to 100, and examine the different communication mechanisms presented thus far. The mean is gathered from the six median values from ten timed cycles.

4.5.1 Sending via Object Streams

The first results present the time taken for the PC to send the test classes using networked object streams. These values are presented in Figure 24. The x-axis represents the length of the `Integer` and `Double` object arrays within the specified object.

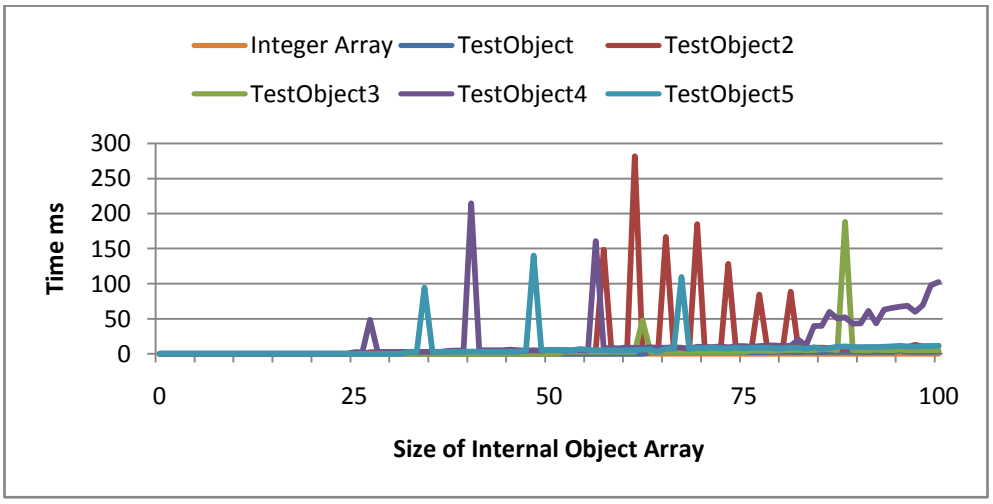


Figure 24: PC Sending Test Objects via Object Streams

Unfortunately, this does not allow close examination due to significant peaks. However, the results for `TestObject4` do increase significantly after size 80. At this point, `TestObject4` is larger than 6000 bytes, and no other test object reaches this size.

The peaks, unlike those in previous results, are for smaller data sizes and allow closer examination. The seven peaks within the `TestObject2` results, for example, occur at regular intervals, the size interval between each peak being 4. From the data size calculated using the equation for `TestObject2` in Table 1, the interval between each peak of these seven peaks represents 192 bytes, which is a multiple of 16. This indicates a probable reason internal to the test framework. Table 5 presents the data size at all the peaks present in Figure 24, calculated with the equations in Table 1 and sorted. The Interval values are the variance between a Peak Value and the previous Peak Value. The Rounded values are the Interval values rounded by 1 to a suitable number.

Table 5: Object Sizes at Peaks

Peak Value	Interval	Rounded
1768	-	-
2268	500	500
2461	193	192
3024	563	564
3137	113	112
3152	15	16
3281	129	128
3345	64	64
3473	128	128
3665	192	192
3857	192	192
4049	192	192
4050	1	0
4240	190	192
4241	1	0

Table 5 indicates a pattern within the peaks, as most variances are multiples of 16 when rounded, except the large intervals of 500 and 564. However, the difference between these two values is also a multiple of 16, and thus the observation of an underlying pattern is strengthened. Two pairs of values have only a single byte variance. The first pair (4049 and 4050) is from `TestObject2` and `TestObject5` respectively, and the second pair (4240 and 4241) is from `TestObject4` and `TestObject2` respectively. This indicates a data packet size problem and not an object complexity problem.

As the peaks can likely be dismissed, any subsequent presented data will have the peaks removed, and the two adjoining value points connected. Actual data results are given in Appendix D. Figure 25 presents the results from Figure 24 with the peaks removed. The results for `TestObject4` are shortened to allow closer examination.

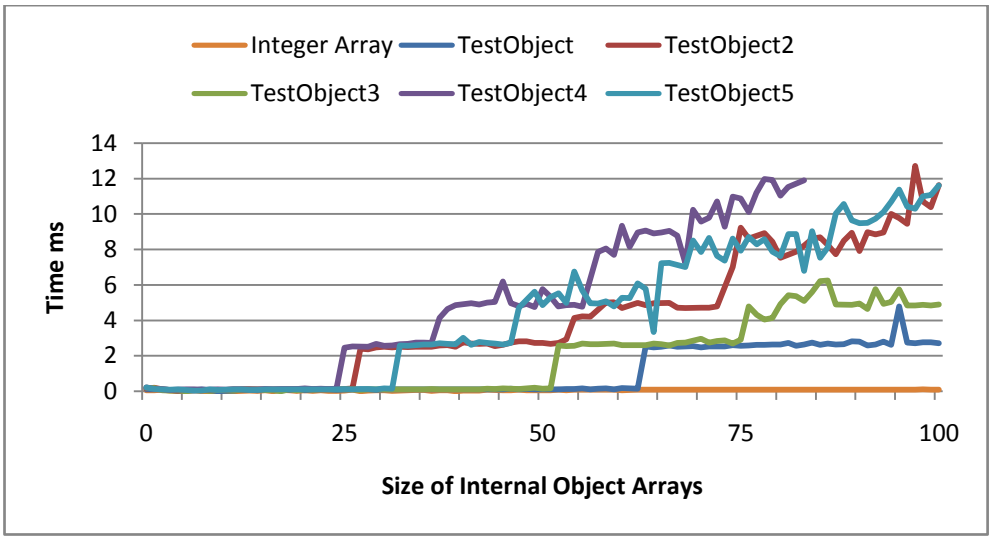


Figure 25: Cleaned PC Sending Test Objects via Object Streams

Figure 25 indicates a number of steps within the results which occur when extra packet send operations are required. Table 6 presents the data size when the steps occur. The MTU for the Ethernet interface is 1500 and for the wireless interface 2272.

Table 6: Object Sizes at Steps

TestObject	TestObject2	TestObject3	TestObject4	TestObject5
1785	1649	2121	2132	2160
	2945	2937	2948	2970
	3905		4240	3942
			5124	5130
			5736	
			6144	

Three test object types have an initial step at approximately the same size, and these values are close to the wireless interface packet size. `TestObject` and `TestObject2` have their initial step appearing earlier however. Four of the test classes show a step at approximately 2950 bytes, which is approximately two Ethernet packets in size. `TestObject2` and `TestObject5` have a step at approximately 3900 bytes, which is approximately an Ethernet packet plus a wireless packet in size. `TestObject4` has a third step value at 4240, which does not conform to a packet size ratio. Both `TestObject4` and `TestObject5` have a step at approximately 5130 bytes, which is approximately two Ethernet packets plus

a wireless packet in size. `TestObject4` has two further steps. At 5736 bytes, there is no conformity to a packet combination. However, 5736 is approximately one Ethernet packet larger than the other `TestObject4` value (4240) that did not conform to a packet combination. The final step in the `TestObject4` results comes at 6144 bytes, approximately one Ethernet and two wireless packets in size.

From these results, I/O time is the key factor for the PC sending objects to the PDA in the test framework, due to the extra packet requirement. Thus, smaller objects will be more efficient for the PC. Large data objects require the PC to send extra network packets, and in the test framework each packet takes approximately 2 to 2.5 ms to send.

Figure 26 presents the results gathered from the PDA sending the test classes to the PC. No peaks have been removed from this result set.

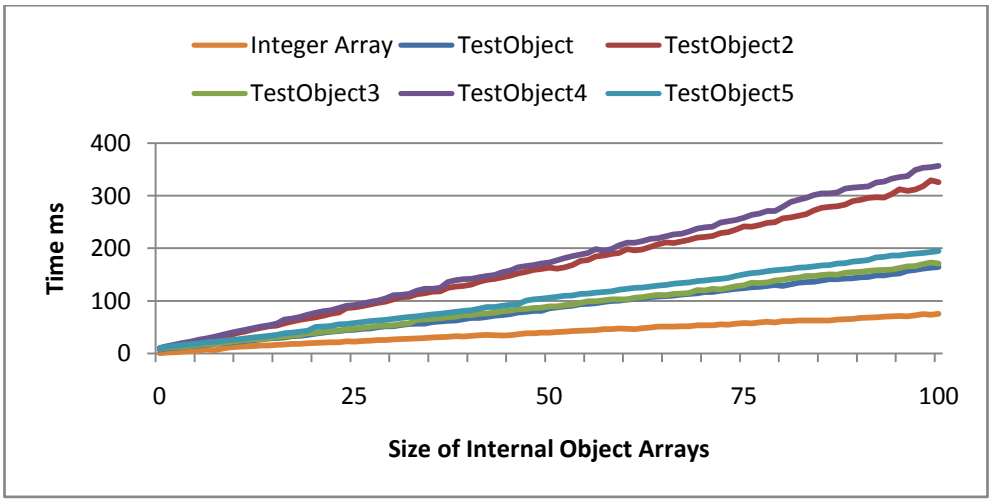


Figure 26: PDA Sending Test Objects via Object Stream

Unlike the PC results, the PDA results has grouping based on object complexity. In fact, this chart is almost exactly as Figure 10 (page 54), which indicates that serialization is the main contributing factor for the PDA sending the test classes. This is of course in line with the lower (de)serialization throughput of the PDA in comparison to I/O throughput on the network, as shown in Table 3.

A possible cause for the PDA being serialization bound is the conversion of numeric values into bytes for transmission. For the test classes, most unique objects wrap a

primitive data type. Analysing primitive data sending illustrates why this is not the case.

Figure 27 presents results for the PDA sending primitive integers using different conversion mechanisms. Int Array is the primitive `int[]` type sent via an object stream. Ints takes every element in the array and transfers it with the `writeInt` method on a Java `DataOutputStream`, which is the stream which underlies the Java object streams. Converted Ints has each integer element converted into four bytes with bitwise operations, with the subsequent bytes stored in a byte array and the byte array transferred as raw data on the network stream. Only Int Array has a result at size 0 as the other methods send nothing at this point. All streams are buffered as with the other experiments, except Converted Ints which sends the generated byte array directly on the network stream.

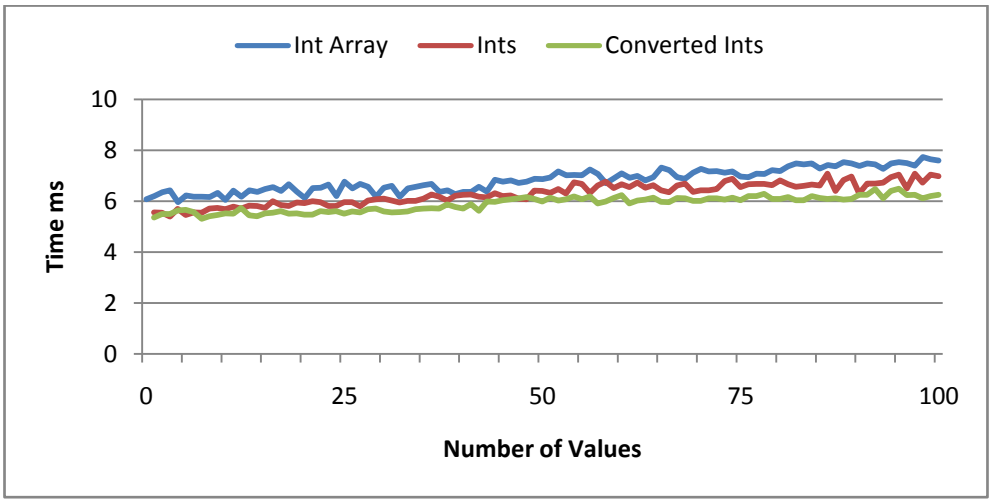


Figure 27: PDA Sending Ints

There is no significant increase in overhead as more numbers are converted. If this was a major factor in the serialization process, it would be expected that performance would change as the Integer array results presented in Figure 26.

There are other interesting points in Figure 27. The bitwise conversion of integers into bytes appears to give a marginal performance increase when compared to sending the integers directly. This may be because of the flush operation required in the buffered stream results. The other interesting point is the slight overhead for sending the `int[]` object, which is approximately 0.55 ms. This will partially be

from the 23 byte serialization header sent with the object, and may in fact be the serialization operation time. If this is true, then it is likely to increase as more objects are serialized, as the 0.55 ms per object overhead alone cannot attribute the difference between the Integer Array results presented in Figure 26 and the primitive `int[]` results presented in Figure 27.

Figure 28 compares send times and receive times for `TestObject4` taken from the PC and PDA with peaks removed. The x-axis provides the calculated size of the object in bytes. The PDA sending time and the time recorded for the PC receiving are equal and the variance between the two results sets never increases above 1 ms. Thus it appears as if only three lines are present in Figure 28, as the PC Receiving result is imposed upon the PDA Sending results.

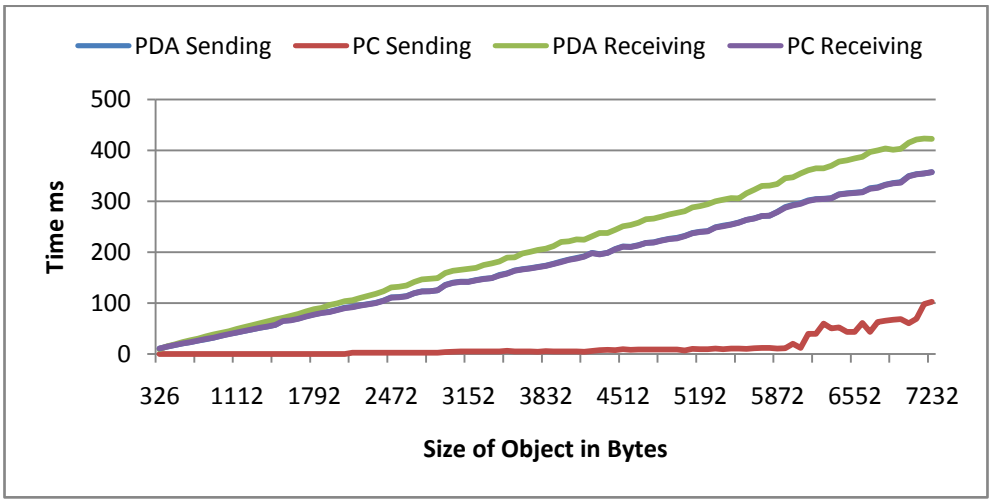


Figure 28: Sending and Receiving TestObject4 via Object Streams

4.5.2 Sending via Channels

To compare the JCSP Networking against object streams for sending the test classes, the results from sending `TestObject4` via JCSP networked channels are presented. Other channel result sets are provided in Appendix D. Expected times are generated using the *NetChan* equation and the performance characteristics provided in Table 3 (page 63) and object sizes calculated from Table 1 (page 47). Only expected synchronised channel values are calculated as synchronisation has little effect on communication time for objects of this complexity. Figure 29

presents these results for the PC. The x-axis represents the size in bytes of the serialized object.

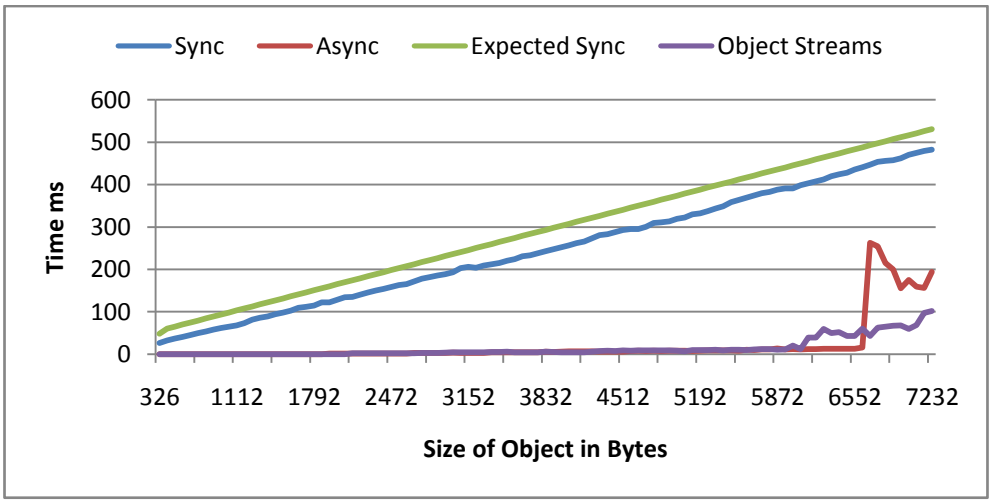


Figure 29: PC Sending TestObject4 via Networked Channels

Synchronised channels perform better than expected, the variance being between 30 and 50 ms. However, not all test classes show improved performance over expected results (see Appendix D) and therefore it is deemed that there are no adverse performance differences for large serialized objects sent via JCSP Networking channels. Async results are relatively flat until spiking at the end similar to Object Streams. The difference in object sizes at the two spikes is greater than the JCSP message overhead, although this will have an effect on the observed results.

Figure 30 presents expected and actual results for the PDA sending `TestObject4`, and no peaks were removed from this data. As with the PC results, the PDA initially shows better than expected results, but as object size increases the variance between the two result sets reduces to zero. As with the PC results, the different test classes exhibit either better or worse results than expected based on their type.

Async channels perform initially as well as Sync channels, but degrade as object size increases. The variance between the Sync results and Object Stream results also increases with object size, although initially channels have performance that is comparable to object streams.

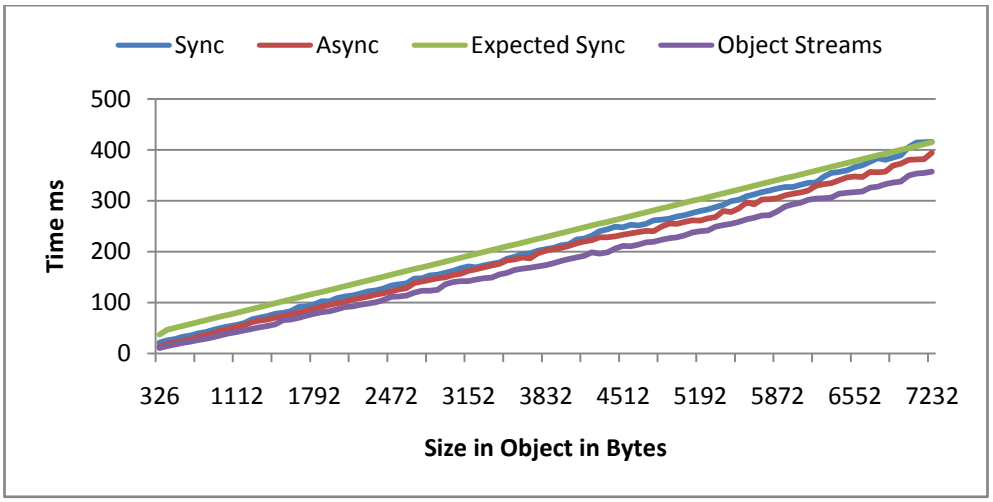


Figure 30: PDA Sending TestObject4 via Networked Channels

From the `TestObject4` sending results presented it is possible to approximate the throughput of the networked channel for a complex object. This value takes into consideration the serialization and deserialization time of both devices. The PC can transfer `TestObject4` messages at approximately 15 bytes/ms, and the PDA can transfer `TestObject4` at approximately 17 bytes/ms. The variance in performance is due to the PDA having lower deserialization performance than serialization performance, and the PDA (de)serialization process being the significant bottleneck.

The throughput reduction is concerning, and is attributed to (de)serialization performance of the PDA. If PC channel bandwidth results (Figure 21 – page 69) and PDA channel bandwidth results (Figure 20 – page 68) are examined, the serialization process for `TestObject4` reduces channel performance by 260 bytes/ms and 188 bytes/ms for the PC and PDA respectively. However, JCSP performance is better than expected, and the PDA results indicate little overhead in comparison to object streams.

Comparing transfer time for large serialized objects gives some indication to the overhead associated with the PDA, but PC results are inconclusive due to the lack of acknowledgement on the object streams causing significantly better results when compared to JCSP Networking. Therefore, roundtrip results are also presented to help compare the object stream and JCSP Networking results further.

4.5.3 Roundtrip

Figure 31 presents the results for the PC performing a roundtrip operation with `TestObject4` using Sync and Async channels and Object Streams. The expected results are also calculated using the properties from Table 3 and the *NetChan* formula. The PDA results are not shown as they are similar, and are available in Appendix D.

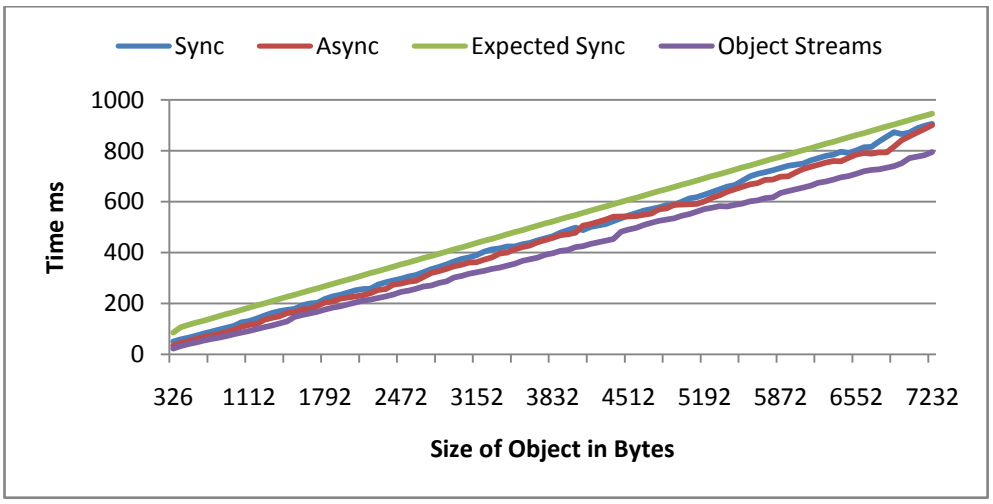


Figure 31: PC to PDA Roundtrip TestObject4

Figure 31 illustrates that actual channel communication time is better than expected, and the expected and actual results increase in unison. Async results are also comparable to Sync results. Object Streams perform better than networked channel communications, and over time the performance gap increases. This highlights a possible problem with complex objects sent over channels, the variance between the two result sets reaching approximately 100 ms at $\texttt{TestObject4}_{100}$[1]. If results from the PDA sending `TestObject4` (Figure 30) are examined, there is an approximate 50 ms variance between channels and Object Streams at $\texttt{TestObject4}_{100}$. The variance between receive times (Figure 32) on the PDA is approximately 50 ms. Therefore, the PC has no significant overhead observed when using JCSP Networking to send complex objects within the test framework, and any overhead can be attributed to PDA performance.

[1] The notation TestObject_n is used to signify the length of the internal arrays within the object in question. e.g. If n = 100, the length of the internal arrays of TestObject is 100.

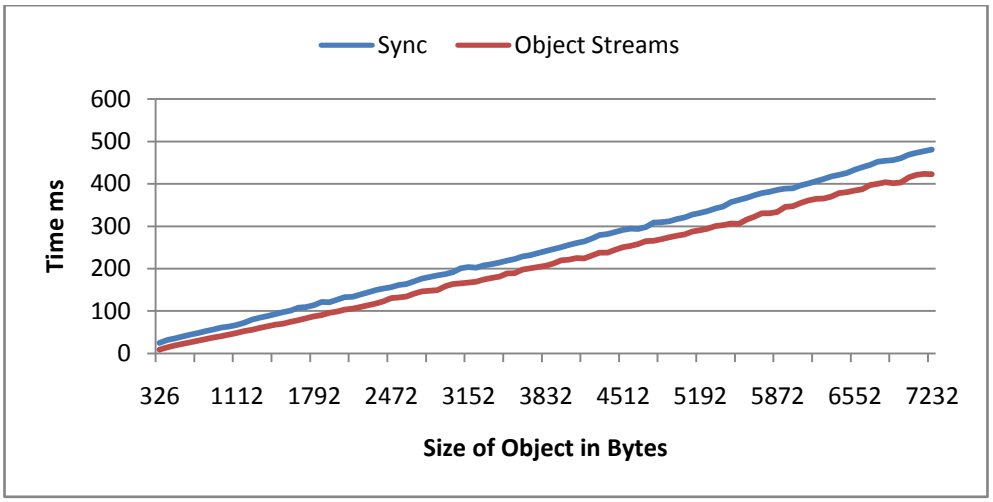


Figure 32: PDA Receiving TestObject4

4.6 Examining JCSP Networking Overhead

Experimental data presented thus far has focused on communication overhead within JCSP Networking. Another concern is resource overhead of the JCSP Networking architecture. Section 3.4 highlighted some initial observations regarding scalability, and issues with `Link` priority were highlighted in Section 4.4. In this section the latter problem is examined in more depth.

To investigate the priority problem, a CommsTime benchmark, utilising fast integer based channels, is performed on the PDA and PC in conjunction with the roundtrip experiment for large data sizes. The latter experiment involves data being sent and received in large blocks, and thus it is possible to examine the computational overhead for I/O. There is a warm up and cool down period when the bandwidth experiment is not operating, allowing the base CommsTime result to be determined. The PDA results for the CommsTime benchmark in this scenario are presented in Figure 33. Experiment time increases with the x-axis, and the broken horizontal line across the figure is the recorded CommsTime figure at various times during the experiment. Vertical lines indicate a packet size time being recorded at that point in time during the experiment, packet sizes increasing as the roundtrip experiment (0, 10^3, $2x10^3$ ... 10^4, $2x10^4$ ... 10^5, $2x10^5$ ... 10^6). Gaps in the CommsTime blue line result indicates that no time was gathered during the packet sizes represented by the red lines. For example, between the times gathered for packet size 0 and 10,000 no CommsTime figure is gathered as the relevant packet

sizes each record a time while the CommsTime benchmark does not, leading to a gap in the blue line representing the CommsTime result. This is due to the device being consumed by I/O operations and is unable to perform computation for the CommsTime benchmark.

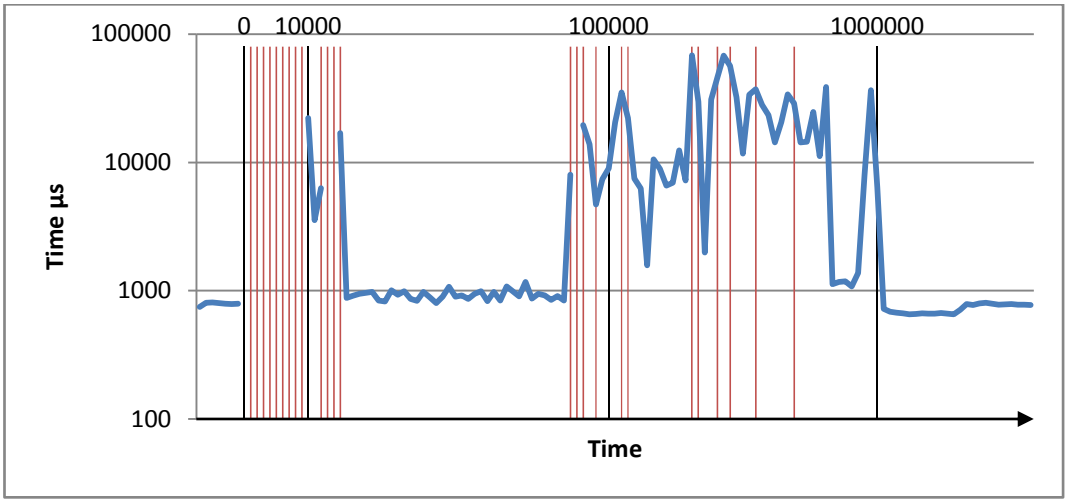


Figure 33: PDA CommsTime Stressed Network

Figure 33 illustrates the high priority `Link` problem as the CommsTime on the PDA increases from approximately 680 µs to approximately 70,000 µs during large packet size transfer. The PDA is essentially flooded and has reduced computation performance within this period, particularly during the larger packet sizes.

There is an interesting phenomenon where CommsTime reduces to approximately normal levels, and no bandwidth results are recorded. This valley occurs during data size 60,000, where channel roundtrip performance also drops in Figure 22 (page 70). It can therefore be judged that the PDA is not performing any operation that should be significantly affecting network performance at this stage, which leads to the probable cause of the network infrastructure causing a performance drop.

Roundtrip results recorded during this experiment, and the original recorded results for roundtrip operations, are presented in Figure 34. As can be seen, the results are similar, indicating that I/O has not suffered during the CommsTime experiment, and I/O has effectively caused the application level CommsTime benchmark to be allocated less computation resource.

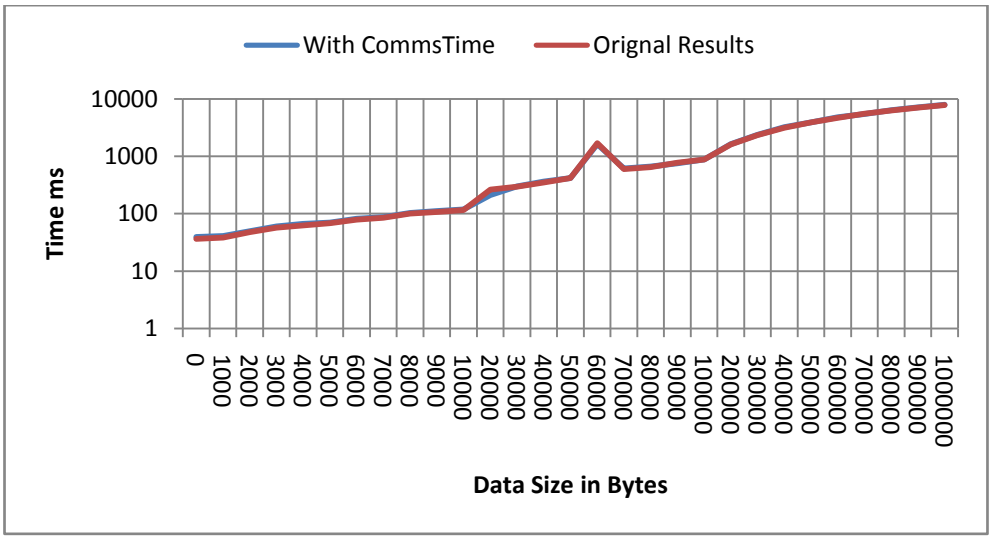


Figure 34: Networked Channel Roundtrip with CommsTime

The PC results are not presented as CommsTime presents no impact on performance and channel roundtrip time results are similar to PDA results.

4.7 Summary

A number of performance experiments have been performed that allow an analysis of the current JCSP Networking implementation. Reflecting these characteristics and the initial observations in Section 3.5 upon the usage of JCSP Networking in a Ubiquitous Computing context raises a number of concerns. Considering the required properties for Ubiquitous Computing highlighted within Chapter 2, the following subsections discuss the problems.

4.7.1 Interoperability

JCSP Networking relies on Java serialization to the point where it is used internally to transfer even non-data messages. From a Ubiquitous Computing point of view, this is a weakness, as not all versions of Java implement serialization, particularly versions aimed at small factor devices [132], which will be in operation in a Ubiquitous Computing environment. As reflection is not available in reduced versions of Java, it is not possible to implement a custom serialization mechanism to overcome the lack of serialization. JCSP Networking requires modification to permit non-serialization interactions, with methods implemented by classes and used to convert an object into a byte array, and the serialization header information

transferred with the byte array. The channel message header is a problem, but as a structure is present (see Appendix A) it is possible to overcome.

A separate `Link` could be developed to allow communication without serialization, and a different message header mechanism could also be developed. Channel messages are constructed in the networked channel object however, and thus the channel would require modification to communicate with the `Link`, or the `Link` would require further modification to extract the information to send.

If either approach is taken, JCSP Networking will require modification to promote interoperability between Java versions. However, there is still consideration for interoperability between different frameworks. Not all computational elements within a Ubiquitous Computing environment will be capable of operating a Java Virtual Machine. When this further restriction is placed on requirements, any notion of Java serialization becomes a problem. In particular, the difficulty interpreting the sent Java object is a problem, as not all platforms provide reference based data structures and object graphs. Data structures are often interpreted differently on different platforms, and thus object based serialization should be avoided. Although work by Ripke [136] has shown that the underlying serialization headers can be accommodated for in languages such as **occam**, the actual implementation of data structures within Java does cause problems.

4.7.2 Performance

JCSP Networking provides performance in the test framework close to optimal performance between the two devices. When using JCSP networked channels, the PC drops to 275 bytes/ms bandwidth from 320 bytes/ms, a 45 bytes/ms reduction. The PDA drops to 205 bytes/ms bandwidth from 215 bytes/ms when using JCSP network channels. The low variance in performance for the PDA indicates that no significant throughput overhead is observable. The PC has a greater variance between the JCSP Networking channel and networked streams, although this can be attributed to the deserialization time for the sent object on the PDA, and subsequent serialization of the acknowledgement packet. From an initial analysis,

JCSP Networking has no significant communication overhead to argue against its usage in various distributed computing contexts, not just Ubiquitous Computing.

Bandwidth does drop when complex object serialization is considered; performance dropping to 15 bytes/ms. This highlights another problem with the reliance on serialization for message encoding. Therefore it is argued that reliance on serialization is a bad choice for high communication ratio applications, which could be prevalent in a Ubiquitous Computing environment.

Latency is a problem however. Within the test framework, a ping takes approximately 36 ms on a synchronized channel and approximately 16.75 ms on an asynchronous channel. Object streams record ping at approximately 6.75 ms, and network streams approximately 6.5 ms. The comparison to a synchronous channel is possibly unfair due to the synchronisation between sender and receiver. However, the asynchronous channel indicates a ping overhead of 10 ms for JCSP networked channels within the test framework above the object streams. The majority of the overhead can be attributed to serialization of the message header on the PDA. High latency can be a problem in high communication applications, and therefore JCSP Networking may not be suitable for such applications.

Asynchronous channels do provide an initial performance increase, but over time the benefit reduces. Eventually asynchronous channels perform poorer than synchronous channels. The infinite buffering mechanism within the JCSP Networking architecture and garbage collection may have an effect. Therefore asynchronous channels do not appear to be a good solution for high latency in all scenarios.

Serialization performance on the PDA is disappointing, and is the greatest bottleneck within the test framework. The PC can serialize objects at approximately 10,000 bytes/ms. The PDA can only achieve approximately 20 bytes/ms in comparison. Considering the network throughput recorded on the PC and PDA (320 bytes/ms and 215 bytes/ms respectively), the PC is I/O bound and the PDA serialization bound. Performance cannot be attributed to conversion of individual values within the transferred object (Figure 27 – page 76) and appears to relate to

the lookup table internal to the serialization process increasing in size. Memory allocation and I/O time do not indicate a relation to the size of the sent object. On small factor devices the usage of serialization can be considered a severe limitation. As Ubiquitous Computing scenarios involve computational elements ranging from large to small, there is a further argument against object serialization.

4.7.3 Resource Usage

As discussed in Section 3.5, process usage within JCSP Networking increases as the number of networked input channels and inter-node connections increases. Temporary processes are created and destroyed during operation, and thus problems can arise in resource constrained devices. The PDA has an approximate 400 thread limit, and although possibly a large number, smaller devices will have fewer threads available. On smaller devices however, a single connection to a server and a single input channel may be all that is required, and the excessive usage of processes may not be a factor.

JCSP Networking relies on a JVM capable of object serialization, which some of the reduced Java configurations do not accommodate. As discussed in Section 4.7.1, this problem can be overcome, but even a reduced JVM may be too resource heavy to operate on some devices. A reliance on Java in Ubiquitous Computing scenarios is therefore a limitation.

4.7.4 System Overhead

JCSP Networking was designed operate in cluster computing type scenarios, which leads to conflicts when considering other usages. `Link` processes are given maximum priority, and therefore during intense I/O operations the application and device will be allocated less computational resource to accommodate I/O. For applications with high computation to low communication ratios, such as cluster computing, high priority I/O enables fast service of communications. For high communication to low computation ratios, the application must wait for I/O to complete, and overtime a small device can be flooded. Small factor devices and high communication ratios are possible in Ubiquitous Computing, thus the high priority `Link` can cause a problem.

Other overheads in JCSP Networking are attributed to using objects as message headers. The required information in a message packet (type, source and destination) is small and therefore 249 and 204 byte headers are excessive. This is another problem with the reliance on serialization for communications.

4.7.5 Scalability

Linked to resource usage and system overhead is scalability. Ubiquitous Computing demands large scaled environments, with multitudes of devices interacting. Scalability is one of the main arguments for using a formalised mobility model. JCSP Networking does not scale well within these architectures. Considering the capabilities for creating dynamic topologies of interacting components possible with JCSP Networking, scalability can be seen as one of the major problems to overcome.

Java is also a problem for scalability. A JVM is not available on every device, thus reliance on Java and serialization is a limitation. Thread limitations allow the PDA approximately 400 processes and the PC 7,000 processes. Applications involving thousands of agent processes moving through devices become difficult if not impossible to achieve. Reliance on Java to accommodate such scale is therefore a limitation.

4.7.6 Stability

No evidence of erroneous behaviour within JCSP Networking is presented, but usage of the framework highlights a number of problems. The underlying architecture does not accommodate exception handling that is accessible to application layer developers. A process may block while communicating to a remote process if the connection between the two Nodes fails. Ubiquitous Computing requires management of failure to enable an environment to stay active in the presence of erroneous behaviour. JCSP Networking does not indicate erroneous behaviour reasonably and cannot be considered a suitable framework from this perspective.

4.7.7 Accessibility and Extensibility

Properties internal to JCSP Networking are hidden. The Nagle algorithm being turned off improves performance in cluster computing scenarios, but does not for scenarios where sending as much data as possible in a packet is more efficient. High priority I/O also causes problems for usage of JCSP Networking in domains outside cluster computing. Buffering the underlying network stream increases performance, but the size of the buffer cannot be modified to suit individual purposes. Finally, reliance on serialization for communications limits inter-framework interaction.

Exposing the underlying mechanisms and attributes would allow modification. Unfortunately, many of these properties are hidden and cannot be modified outside the source code. Numerous scenarios are possible in Ubiquitous Computing due to differing communication, device and architecture configurations. Thus, the underlying properties should be exposed to allow modification.

A final consideration is extensibility. In principle, JCSP Networking can utilise different communication mechanisms, and functionality can be extended using the networked channels. However, the existing architecture requires numerous resources to allow a networked channel, and therefore resource usage for other communication scenarios negates scalability. The implementation is also complex [135], requiring a level of understanding of the internal mechanisms of JCSP Networking to allow extensions to be created.

4.7.8 Conclusion

In this chapter, weaknesses have been identified within JCSP Networking that highlights issues when considering a Ubiquitous Computing scenario. Many of these problems can be linked to Java and serialization, although some are related to implementation decisions within JCSP Networking. Therefore, it is necessary to address these problems and modify JCSP Networking to accommodate Ubiquitous Computing ideas. In the following chapter, a new implementation of JCSP Networking is presented which aims to rectify many of the highlighted problems.

Although performance outside object serialization is not considered a major problem, any improvement of performance is also desirable.

Chapter 5 A New Architecture and General Protocol for JCSP Networking

Chapters 3 and 4 highlighted limitations of JCSP Networking when considering Ubiquitous Computing requirements. In this chapter, a description of a new implementation of JCSP Networking is presented and a definition of a protocol to allow communication between various implementations of distributed communicating process architectures is provided. Section 5.1 presents the new architecture, and Section 5.2 the underlying protocol. Section 5.3 discusses the operation of the new implementation, and Section 5.4 illustrates why it promotes data independence. Finally, Section 5.5 provides a summary of the new JCSP Networking implementation.

5.1 New Architecture for JCSP Networking

Two architectural views of the new implementation are presented. The first view provides a layered examination of JCSP Networking, allowing separation of functionality into different layers. The second view examines the internal components of the layers, discussing how they interact together to support the underlying distributed channel mechanism.

5.1.1 Layered Model

A basic layered view of the architecture is presented in Figure 35. It consists of four layers:

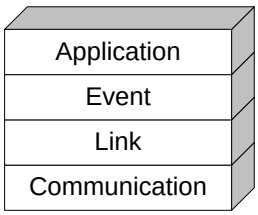


Figure 35: Basic Layered Architecture

- *Application Layer* – user level processes and applications.
- *Event Layer* – networked channel ends and other synchronization primitives. Interfaces are provided to the application level processes, and the communication functionality of the components encapsulated.
- *Link Layer* – connections to other nodes within the system, including receive (RX), transmit (TX), server, and manager processes.
- *Communication Layer* – the underlying communication mechanism that a JCSP Networking system is implemented upon.

The original implementation of JCSP Networking also has some layered attributes, but the new implementation places more restrictions on cross layer communication. Messages travel up and down the layers as far as necessary, and this will be discussed further in Section 5.3.

The layered diagram can be expanded to illustrate how each layer communicates with others, and how addressing within each layer is handled. This diagram is presented in Figure 36.

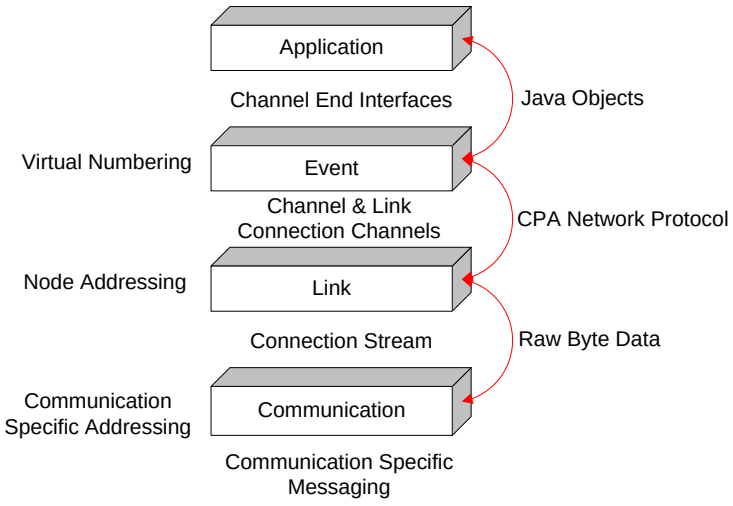


Figure 36: Detailed Layered Architecture

On the left the addressing mechanism is given, on the right the message types are given, and down the centre the interfaces between the layers is given. The interfaces are:

- *Channel End Interfaces* – interfaces defined by the core, non-networked, JCSP channels with networked functionality added.
- *Channel & Link Connection Channels* – the crossbar allowing multiple `Links` to communicate to multiple channel ends and multiple channel ends to communicate to multiple `Links`. The crossbar is implemented using Any-2-One channels in both directions.
- *Connection Stream* – `Links` communicate with the communication mechanism using streams. These streams are communication specific.
- *Communication Specific Messaging* – the communication mechanism's specific messaging protocol (e.g. TCP/IP). This is of no concern to the new implementation or the rest of this research.

Addressing mechanisms between each layer are:

- *Virtual Numbering* – number allocated for addressing and lookup purposes. These are 4 byte signed integers for an addressing range of -2^{31} to $2^{31}-1$.
- *Node Addressing* – each Node is uniquely identifiable to allow inter-Node connections. `Link` management relies on addressing to ensure that only one `Link` to a remote Node exists. An address takes the form *<Protocol>**<Address>*. *Protocol* identifies the underlying communication mechanism (e.g. tcpip) and *Address* is the unique address of the Node based on the addressing mechanism of the communication mechanism.
- *Communication Specific Addressing* – the addressing mechanism enforced by the communication mechanism, for example *<IP Address>:<Port>.*

Most interface and addressing concepts are inherited from the original JCSP Networking implementation. Addressing has been modified to allow addresses to be easily constructed and deconstructed into strings to promote inter-framework interoperability. For example, a `NetChannelLocation` (address of a specific channel end) of a channel with virtual number 74 on a TCP/IP connected Node takes the form *tcpip**192.168.1.100:5000/74*.

Each layer only understands certain message types. These are:

- *Java Objects* – the Application Layer of a Java system operates using Java objects. Therefore this is the type of data it will communicate via the networked channel ends.
- *CPA Network Protocol* – the responsibility of the Event Layer is to convert outgoing messages into Network Protocol messages for communication via the Link Layer, and conversion of incoming Network Protocol messages from the Link Layer to communicate with the Application Layer. This protocol will be discussed further in Section 5.2.
- *Raw Byte Data* – data leaving a Node is sent as bytes, which aids other platforms to interpret the incoming message. In particular protocol messages are transmitted as primitive data written directly onto the stream.

To avoid a reliance on Java objects and serialization, the Application Layer can operate using whatever data type it understands. The Event Layer converts data into raw bytes for transmission and subsequent reconstruction on reception. To do this, the Event Layer utilises data encoders and decoders to perform the conversion. This will be explained further in Sections 5.3 and 5.4.

5.1.2 High Level Model

The individual components and how they are connected is presented in Figure 37, which closely resembles Figure 1 (page 34) with changes to the implemented components.

- `LoopbackLink` has been removed. This component was unnecessary and locally connected `NetChannelOutputs` now send directly to the corresponding `NetChannelInput` end. This will be explained further in Section 5.3.
- `NetChannelInputProcess` has been removed. The required functionality has been folded into the `NetChannelInput`, and the `NetChannelInput` is now as lightweight as the `NetChannelOutput`.
- `IndexManager` has been renamed `ChannelManager`. Each communication primitive requires its own management component within the Event Layer, and the renaming reflects this change.

- `LinkManager` is now a shared data object instead of a process.
- The `EventProcess` has been removed. When the `LinkManager` is informed of `Link` failure, the event is immediately written to the Link Lost Event Channels. These channels are infinitely buffered to avoid deadlock.

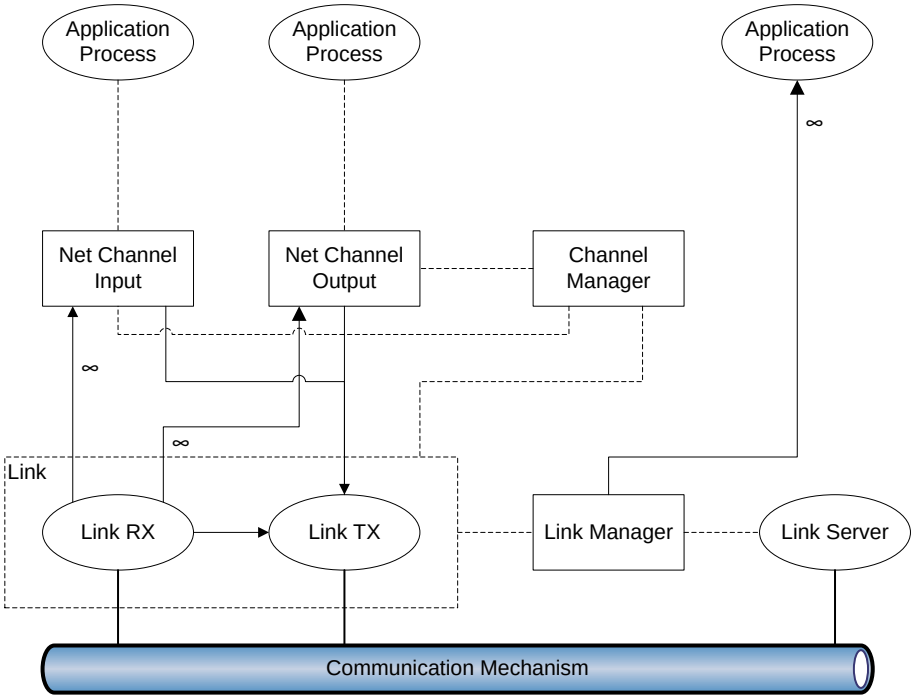


Figure 37: High Level Architectural Model

The key feature of the new architecture is the reduction in resource usage by reducing the number of processes internal to JCSP Networking. The removal of the `NetChannelInputProcesses` ensures that channels are more lightweight, and other unnecessary processes have also been removed. Although not shown, the connection service to the Channel Name Server has also been modified to a shared passive object instead of a process, although this could have trivially been accomplished in the existing architecture.

5.2 General Protocol for Communicating Process Architectures

Permitting inter-framework communication is difficult without well defined protocols. In this section, a description of the communication protocol for JCSP Networking is presented. The primary goal is that messages should be platform independent, and using simple data primitives helps to achieve this goal.

5.2.1 Protocol Definition

The protocol is based primarily on an examination of the original JCSP Networking implementation, the pony Framework [130] and C++CSP Networked [129]. By performing this examination, a great deal of common functionality and messaging can be deduced. The key shared feature is a virtual channel across a communication mechanism. Sent messages have a destination, and to permit synchronisation the message must be acknowledged, and therefore messages also have a source. Thus there are two attributes for a basic send message. The type of the message must also be included, providing a message triple. All required messages can be defined with a triple. There is also the optional data segment for data messages, providing the following message signature:

(<message type>, <attribute 1>, <attribute 2>, [<data>])

Each value in the message header is represented by a primitive data type. The message header signature is:

(byte, 32 bit signed integer, 32 bit signed integer)

Inclusion of the data segment depends on *<message type>*. A message receiver acts accordingly to read data from the communication stream based on the incoming message type. If the size of the data is sent as a header, then the receiver will know how many bytes to read. Therefore, *data* has the following signature:

(<size>, <bytes>)

or

(*32 bit signed integer, [1..size] bytes*)

A signed integer is used as Java provides no unsigned value types, although this could be changed to avoid negatively sized data messages. It is not envisioned however that a data message will be as large as 2^{31}-1 bytes (2 Gbytes), which is beyond the limits of allocated data sizes. The next standard data size is a 16 bit unsigned integer, and this would only provide a maximum message size of 64

kilobytes. Although larger data packets could be avoided, they are not unimaginable.

The basic channel message types are:

- *SEND* – basic send message, requiring a source, destination, and data segment. *(SEND, <destination>, <source>, <data>)*
- *ACK* - SEND acknowledgement which notifies the sender that the message has been read. Only the destination of the acknowledgement is required. *(ACK, <destination>, null)*
- *REJECT_CHANNEL* – when a message is sent to a non-existent or destroyed channel, the sending channel is informed with a REJECT_CHANNEL message. The term reject is taken from rejectable channels that were used to pass I/O exceptions to application processes in the original JCSP Networking implementation. Only a destination is required. *(REJECT_CHANNEL, <destination>, null)*
- *POISON* – poisoning of channels is a new addition to JCSP [137], based on work originally by Welch [138] and then Sputh [139]. Poisoning will be briefly discussed in Chapter 8 in relation to process mobility. A POISON message requires a destination and a poison strength. *(POISON, <destination>, <strength>)*
- *LINK_LOST* – when a `Link` fails, `NetChannelOutputs` connected via the `Link` to their corresponding `NetChannelInputs` must be informed. LINK_LOST messages are sent to each `NetChannelOutput` by the `Link`. This message is not channel specific as other components in the Event Layer will also be informed of this occurrence. LINK_LOST messages will also never be transmitted between Nodes, but by a `Link` to local components. No extra information is required within this message. *(LINK_LOST, null, null)*
- *ASYNC_SEND* – an unacknowledged SEND message. Usage of this message permits the unacknowledged channel functionality from the existing JCSP Networking architecture. Asynchronous messages are used by the Channel Name Server to avoid blocking when servicing name registration or request. The requirement for asynchronous messaging was to avoid deadlock caused

by connection failure prior to acknowledgement, but this problem has now been resolved (Section 5.3). The ASYNC_SEND message has the same form as a SEND. *(ASYNC_SEND, <destination>, <source>, <data>)*

Beyond the networked channels, there are other components within the Event Layer. These include networked `Barriers` and `Connections`, but currently not networked `AltingBarriers`. For completeness, the required message types are provided in Appendix E.

The protocol is not complete, and further work is needed to discover other required message types. For example, **occam** uses a claiming technique to control access to shared channel ends [140]. JCSP has no such technique, but it should be possible to enforce on networked channels without modifying the channel interfaces. However, claiming of channels may not be a requirement as future versions of **occam** may not utilise explicit claiming. `AltingBarrier` [137] is a further consideration for networked systems.

5.2.2 General Nature of the Protocol

The protocol promotes inter-framework coordination due to how the messages are defined. Message type is represented by a single byte – providing 255 message types – and thus a lookup enumeration can be used on the message type on reception. If all frameworks agree on message values, each framework can focus on how the architecture can be implemented. If correct behaviour is emitted by an implementation (e.g. each SEND must be given an ACK) then the individual implementations are separated. Addressing of individual synchronisation primitives in the Event Layer utilises 32 bit signed integers, allowing interpretation on the majority of other frameworks. There is a concern related to the usage of big-endian or little-endian to represent values on a framework [129]. Network byte-order (big-endian) should therefore be conformed to. Data within a message has been separated from the header, and thus only conversion of data across platforms' is a concern. This will be discussed in Section 5.4.

5.3 Operation

In this section, basic operations of the new JCSP Networking architecture are outlined. The methods to capture `Link` failure and data conversion are also covered. First a brief description of the new virtual channel is presented.

5.3.1 Virtual Channel

The new implementation of the virtual channel is functionally similar to the original implementation with `NetChannelInputProcess` operations folded into the `NetChannelInput`. Thus the reading process performs the read operation explicitly. Therefore it is the reading process' task to recreate the sent data into an object (or otherwise).

Figure 38 illustrates how a virtual channel crosses the layers of one system to another, the arrow being the virtual channel.

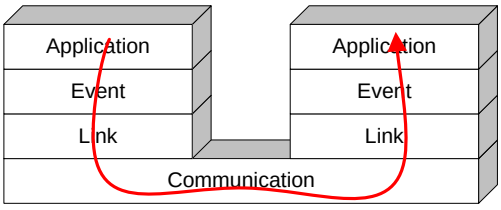


Figure 38: Layered Virtual Channel

Figure 39 illustrates how components interact in the new architecture to form a networked channel. Figure 39 is similar to Figure 2 (page 37) with `NetChannelInputProcess` removed. Messages between components illustrate the data that is being communicated, with SEND being represented by 1 in the protocol and ACK being represented by 2. Numbers in parenthesis within the `NetChannelInput` and `NetChannelOutput` are the virtual channel numbers in use.

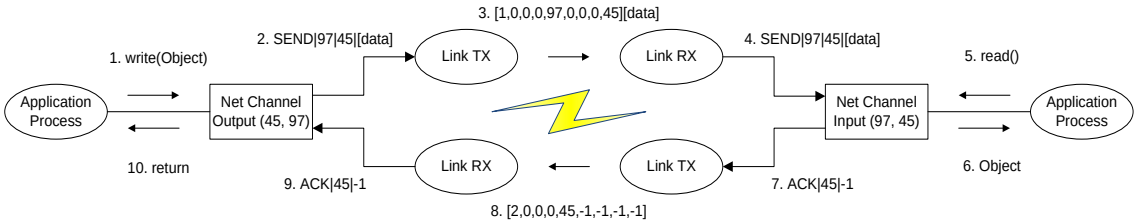


Figure 39: New Networked Channel

5.3.2 Basic SEND / ACK Operation

Figure 39 illustrates a normal read-write operation in the new architecture. A description of the operation within the existing architecture was provided in Section 3.3. Here the same description is given for the new implementation of JCSP Networking, discussing the messages being sent between the components.

1. An Application Process calls `write` on a `NetChannelOutput`, passing an Object to send as a parameter.
2. The `NetChannelOutput` constructs a network message, setting the type as SEND, attribute 1 as the destination value (97) and attribute 2 as the source value (45). The `NetChannelOutput` then must convert the Object into bytes. This is the only point at which data is copied, and if actual bytes are sent then no copying may happen at all. On creation of the `NetChannelOutput`, an encoding filter was provided to accomplish this, and once passed through the filter, an array of bytes is returned. The `NetChannelOutput` attaches this to the network message and sends the message to the `LinkTX`, and awaits acknowledgement.
3. The `LinkTX` reads in the network message and writes the type (1) and two attributes (97 and 45) to the stream. The stream of bytes sent is therefore <1, 0, 0, 0, 97, 0, 0, 0, 45>. The `LinkTX` examines the type of message, and as it is SEND there is a data portion. The `LinkTX` writes the size of the byte array to the stream, and then the bytes that make up the object.
4. The receiving Node's `LinkRX` reads in the type and the two attributes, creating a network message from them. The `LinkRX` process then examines the message type, which is SEND and thus contains data. The size is read from the stream and used to read the required number of bytes from the

stream. The `LinkRX` then retrieves the destination channel end from the `ChannelManager` and checks its state. If the channel is in an OK_INPUT state the channel connecting to the partner `LinkTX` is added to the message, and the message sent to the `NetChannelInput`.

5. The Application Process calls `read` to receive the incoming message.
6. The `NetChannelInput` reads in the network message and checks the message type. As the type is SEND the message is to be delivered. The `NetChannelInput` has a decoding filter to convert a sequence of bytes back into an object, and retrieves the bytes from the message, passes them through the filter to recreate the sent Object and returns this to the Application Process.
7. During step 6, a network message is created with the type ACK. Attribute 1 is set to attribute 2 of the incoming message (the original source) and attribute 2 is not used and set to -1. This message is written on the channel contained in the original message; the channel to the `LinkTX` process connected to the sending Node.
8. The `LinkTX` process reads the network message and writes the type (2) and the two attributes (45 and -1) to the stream. The stream of bytes sent is <2, 0, 0, 0, 45, -1, -1, -1, -1> or <2, 0, 0, 0, 45, 255, 255, 255, 255> if byte is considered unsigned. The `LinkTX` examines the type of the message, and as the type is ACK there is no data.
9. The original sending Node's `LinkRX` reads in the type and two attributes creating a network message from them. The `LinkRX` then examines the message type, and as it is a type that contains no data there is no need to read data from the stream. The `LinkRX` retrieves the channel from the `ChannelManager` and checks its state. If the channel is in an OK_OUTPUT state the network message is written to the `NetChannelOutput`.
10. The `NetChannelOutput` reads in the network message and checks the message type. As the type is ACK the `write` operation completes normally, freeing the Application Process.

The steps provided describe the operation under normal conditions. If the `NetChannelOutput` is connected locally to a `NetChannelInput`, the same operations occur although at step 2 the message is sent directly to the `NetChannelInput` object with the acknowledge channel of the `NetChannelOutput` attached for direct acknowledgement.

As the architecture utilises I/O there is the possibility that erroneous behaviour can occur. The following sub-sections illustrate how this is handled in the new architecture.

5.3.3 SEND / REJECT operation

As stated in Chapter 3, the existing method for erroneous message delivery was implemented by the now deprecated rejectable channel mechanism. It is obviously still possible that erroneous message delivery can occur due to channel destruction or I/O operations. Therefore message rejection is kept, but implemented within the Link Layer instead of the `NetChannelInputProcess`. Figure 40 illustrates the component interactions that occur. The sequence of operations is:

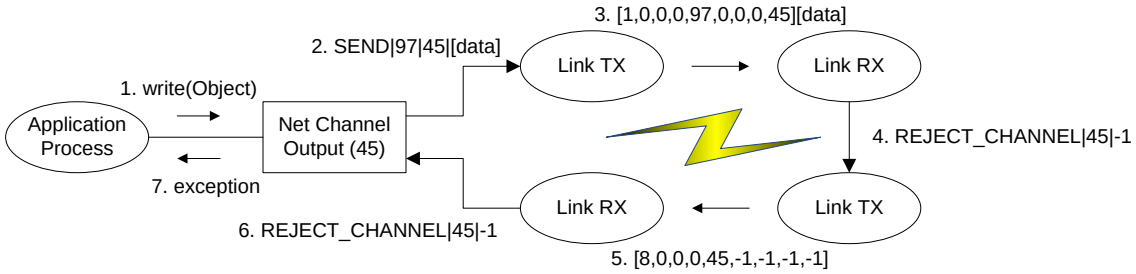


Figure 40: Reject Channel Operation

1. As normal operation
2. As normal operation
3. As normal operation
4. Initially this operation occurs as before. When the `LinkRX` attempts to retrieve channel 97 from the `ChannelManger`, the channel may not exist or its state may not be OK_INPUT. In either case, the `LinkRX` generates a network message and assigns the type REJECT_CHANNEL (8). Attribute 1 is

set to attribute 2 of the original message (45), and attribute 2 is not required. The network message is sent to the partner `LinkTX`.

5. As normal operation step 8, the `LinkTX` writes the message to the stream. There is no data segment.
6. The `LinkRX` reads in the type and two attributes. As the type contains no data segment, no data is read from the stream. The `LinkRX` then retrieves the necessary channel from the `ChannelManager` and checks the channel's state. If the channel is OK_OUTPUT the message is sent to the `NetChannelOutput`.
7. The `NetChannelOutput` reads in the message and checks the message type. As the message type is REJECT_CHANNEL, it is determined that the previous send was rejected. The `NetChannelOutput` changes its state to BROKEN and removes itself from the `ChannelManager`. An exception is raised and causes the Application Process to continue but with an exception.

5.3.4 SEND / LINK_LOST

Another form of erroneous behaviour occurs when the connection to the Node where the `NetChannelInput` resides fails. As stated in Chapter 3, this is not always captured by the original architecture depending on the stage of the read/write operation. To overcome this, a `NetChannelOutput` registers itself with a `Link` when it is created. As a `NetChannelOutput` will only connect to one `NetChannelInput`, a `Link` can retain a set of all connected output channels. If the connection to the remote Node is lost, the `Link` can inform all its registered channels by sending them a LINK_LOST message. `Link` failure may occur at any stage and therefore cannot easily be mapped into operational steps. There are two possibilities however:

- Prior to a `write` operation, the `Link` to the remote Node hosting the `NetChannelInput` fails, causing a LINK_LOST message to be sent to the `NetChannelOutput` on its acknowledgement channel. When `write` is called on the `NetChannelOutput`, the acknowledgement channel is first

checked for pending messages. As LINK_LOST will be present, the `NetChannelOutput` can behave as if a message was rejected.

- After performing a `write`, but prior to receiving the ACK, the `Link` to the `NetChannelInput` fails. The `Link` informs all registered channel ends with a LINK_LOST message on their acknowledgement channels. The `NetChannelOutput` will read in this message, discover it is a LINK_LOST message and act as if the message was rejected.

By having all `NetChannelOutputs` register with `Links`, `Link` failure can be transmitted as required, thus avoiding the deadlock problem described in Section 4.7.6. `NetChannelInputs` do not have this requirement as they may service multiple incoming connections. To avoid deadlock, the `LinkTX` remains active to black hole any outgoing messages. This restriction can be overcome either by converting the `LinkTX` into a passive object which throws an exception when closed, or by poisoning the incoming channel.

5.3.5 Exception Handling

I/O operations can fail for a number of reasons. Passing failures to the Application Layer is the key to allowing recovery by user level applications. Passing exceptions as I/O exceptions is not an option however, as I/O exceptions must be explicitly caught by an application within Java. JCSP Networking utilises the existing JCSP core interfaces for channel ends, and these do not specify I/O exceptions as possible failures. Therefore an exception has been created – `JCSPNetworkException` – which is an unchecked exception and does not have to be explicitly caught by an application, allowing existing processes to operate as if networked channel ends were not in use. If the exception is raised, it will cause the program to terminate if not explicitly caught, thereby allowing erroneous behaviour to be accommodated for if required. Any underlying I/O exceptions within the new JCSP Networking architecture are caught and `JCSPNetworkException` thrown appropriately.

5.3.6 Channel States

Previous operational descriptions have mentioned channel states to determine how the Link Layer and Event Layer should behave. These state objects are shared

between separate processes, and access and modification is protected using standard Java monitors. The individual channel states are as follows:

- *INACTIVE* – initial channel state. This occurs prior to initialisation.
- *OK_INPUT* – a `NetChannelInput` willing to receive incoming messages.
- *OK_OUTPUT* – a `NetChannelOutput` willing to send outgoing messages.
- *DESTROYED* – the channel end has been destroyed by an Application Layer process. This is usually performed to recover resources.
- *BROKEN* – a `NetChannelOutput` end that has become broken due to some form of erroneous behaviour.
- *POISONED* – a channel end that has become poisoned, either by receiving a POISON message or by an Application Layer process invoking `poison`.

Figure 41 illustrates the transitions that occur between states within the channel. This diagram is important when verification of the new architecture is presented in Chapter 6 and when channel mobility is presented in Chapter 7 and Appendix G.

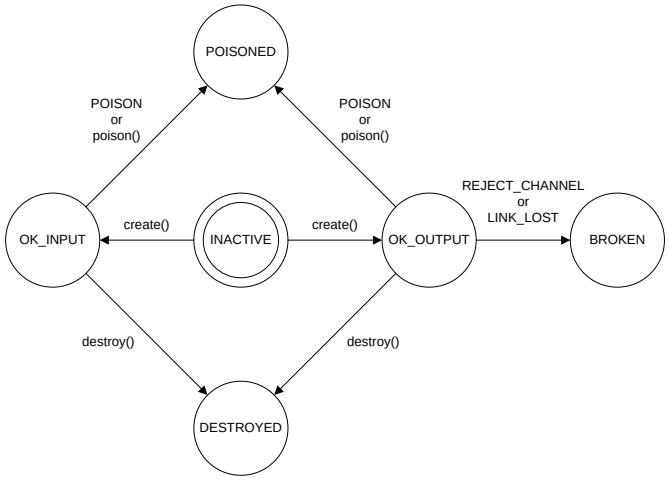


Figure 41: Channel State Transition

The SEND / ACK operation highlighted the usage of filters to encode and decode objects into bytes for transfer. The usage of these filters provides a level of data independence which is discussed in the next section.

5.4 Data Independence

Responsibility for conversion of data is now with the components within the Event Layer. These components have filters placed within them to handle encoding to (output) and decoding from (input) raw bytes. The default filter within a JCSP Networking channel uses serialization as the existing architecture did, except serialization is performed within memory streams instead of buffered communication streams. This is required as `Links` no longer interpret object messages. Different filters will allow conversion using other techniques; the simplest filter sending a byte array and performing no conversion.

Separating data conversion provides the user with some data independence, which is important for cross framework communication. If two frameworks agree on a data transfer mechanism, then inter-framework communication via the communication protocol becomes possible. There are still problems however. Brown [129] illustrates the point when considering C++CSP Networked, in that different platforms may define data structures differently, endianess of bytes being highlighted as a particular problem. Endianess can be overcome by enforcing the network standard byte order, but if other platforms such as pony [120, 130] are considered then some standards must be enforced.

occam has no cyclic data structures as Java, C, and other reference / pointer based languages do. Thus object graphs cannot be faithfully transferred from a JCSP Networking system to pony. The solution is simple although it enforces certain constraints on the Java programmer. If data structures are to be transferred between platforms in a manner that can be interpreted by all available platforms, then the most restrictive structure of data must be adhered too. Schweigler's work on pony [120, 130] permitted communication of **occam** data structures, thus providing insight into possible directions. Providing such a mechanism in Java is left for future work.

As data conversion has been abstracted to the point where the JCSP Networking user can implement their own mechanism, then ubiquitous communication between devices becomes easier. There is no requirement of having Java on the

target platform, thus opening up the possibilities of communication. This is a key feature to permit JCSP Networking to be considered as an architecture suitable for Ubiquitous Computing applications. The only hurdle lies in graph based data structures. Sending such data structures may be a problem for certain frameworks and a question is at what point a cyclic graph becomes a necessary data structure to send between two remote Nodes.

5.5 Summary

In this chapter a presentation of the new JCSP Networking architecture and protocol has been presented. Architectural diagrams and protocol definition were provided, and how separate components communicate. Where necessary, comparison to the original JCSP Networking architecture was provided to illustrate improvements and differences. Exception handling and channel states were also presented. Finally the mechanism for providing data independence was described.

An implementation of this architecture and is currently available via the JCSP repository[2]. The current version is the reference version based on the work presented in this chapter. The reference implementation also includes implementations of the channel mobility and code mobility models presented in previous work [17].

In the following chapter, the new architecture is examined from a performance point of view, repeating the experiments performed on the original JCSP Networking implementation. A verification of the new model and protocol is also presented that illustrates that certain properties are present in the new architecture, and that problems in the original architecture have been overcome.

[2] The JCSP repository is available from http://www.cs.kent.ac.uk/projects/ofa/jcsp/

Chapter 6 Examining the New Architecture

In this chapter, experimental data is presented to compare the new JCSP Networking implementation with the original implementation. Relevant experiments conducted in Chapter 4 are repeated to compare the properties of interest within the test framework described in Section 4.1. To check error handling and other possible architectural implementation issues, a model of the new implementation has been developed using the SPIN model checker [141]. A discussion of the model is presented at the end of this chapter. Section 6.1 presents the expected performance for the new implementation before actual performance is presented in Section 6.2. Section 6.3 examines object serialization and Section 6.4 presents the overhead of the new implementation of JCSP Networking. The verification model is presented in Section 6.5, before conclusions are drawn in Section 6.6.

6.1 Expected Channel Performance

Section 4.4 described interactions between each component within the original JCSP Networking channel and provided formulae to approximate channel performance based on known properties. In this section, new formulae are presented based on the new implementation presented in Chapter 5. There are eight operations with values:

1. `NetChannelOutput` encodes the sent object message
2. `NetChannelOutput` writes the message to the `LinkTX` (channel communication)
3. `Link` transmits the message to the remote `Link`.
4. Remote `Link` sends received message to the `NetChannelInput` (channel communication)

5. `NetChannelInput` decodes the sent object
6. `NetChannelInput` writes the acknowledgement message to the `Link` (channel communication)
7. `Link` transmits the acknowledgement to the original `Link`
8. `Link` sends the acknowledgement to the `NetChannelOutput` (channel communication)

The channel has two ends. Both require a formula to determine approximate communication time. These formulae are:

$$C_{out} = 2 \cdot chan + serialize(sizeof(message))$$

$$C_{in} = 2 \cdot chan + deserialize(sizeof(message))$$

Total communication time from output to input is calculated as:

$$NetChan = C_{out} + C_{in} + transmit(sizeof(message) + 13) + transmit(9)$$

These formulae include object (de)serialization time, but the new implementation allows raw data communication without serialization. In such circumstances, (de)serialization time is omitted. Asynchronous channel operations remove acknowledgement time of a channel communication per networked channel end, and the transmission of the 9 byte ACK message.

From Table 3 (page 63) it is possible to estimate channel performance when sending a null or single byte value. These values are presented in Table 7. All values are in milliseconds.

Table 7: New Net Channel Overhead

	C_{out}	C_{in}	NetChan
PC Sync	0.030	0.031	5.404
PC Async	0.015	0.016	0.221
PDA Sync	0.373	0.385	5.404
PDA Async	0.193	0.205	5.208

From an initial comparison of expected results from the original JCSP Networking implementation presented in Table 4 (page 65) and expected results of the new implementation presented in Table 7, expected communication time has decreased

by approximately 9 ms for a synchronised channel. Asynchronous channels on the PC should perform a send in approximately 0.22 ms and on the PDA performance is expected to be approximately 5.208 ms. The significant decrease for the PC asynchronous channels is due to the PDA not performing deserialziation on the incoming channel message header.

6.2 New JCSP Networking Performance

To analyse the new implementation, the ping, bandwidth and roundtrip experiments are repeated using large data packets. Raw byte data can be sent directly on a channel with no conversion, thus experimental data representing this scenario is also presented. Unlike the experiments conducted on the original JCSP Networking implementation, `Links` are only given normal priority within the test framework.

6.2.1 Simple Ping

Figure 42 presents results for a ping benchmark using the new JCSP Networking channel implementation. Original JCSP Networking and Object Stream results are provided for comparison. Expected results are generated with the new *NetChan* formulae. Times presented are the average time in milliseconds to perform the ping operation.

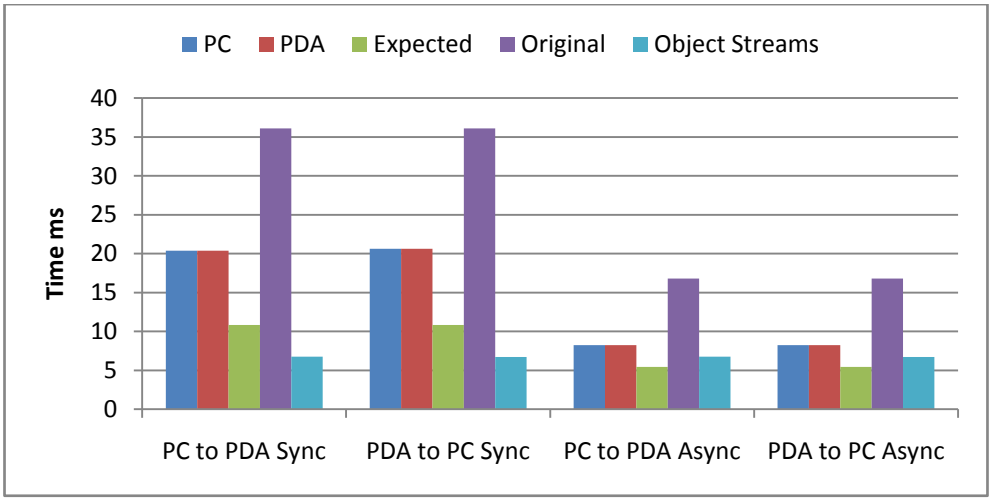


Figure 42: Simple Ping New Network Channel

Figure 42 indicates an increase in performance for small data packets. For synchronous channels, both the PDA and PC results show an approximate 15 ms

improvement. There is an approximate 10 ms variance from expected results. However, taking into account network latency subtracts 3 ms from this figure, providing an approximate latency of 7 ms for a network channel roundtrip operation, or 3.5 ms for a send operation. There is an approximate 14 ms variance between Sync and Object Stream results.

Async results are favourable, actual results being approximately 3 ms greater than expected. Taking into account roundtrip latency of the network at 1.5 ms, the latency for Async roundtrip can be approximated at 1.5 ms. Object Stream results perform approximately 2 ms better than asynchronous channels under these conditions.

6.2.2 Bandwidth

Bandwidth experiments consist of single byte messages and large data sizes. For the former, only null value objects are sent via a networked channel using serialization, and for the latter, both serialization and raw data results are provided. As serialization takes place within the channel, a memory buffer is utilised to serialize the object into. The buffer is allocated 8192 bytes, the same buffer size as the `Link` stream in the new and existing JCSP Networking implementation. Each `NetChannelOutput` is given its own buffer. As the size of the data to be serialized is greater than 8192 bytes within these experiments, the buffer is doubled as required by Java. At the next serialization operation the buffer is reset to 8192 bytes. Increasing the buffer in this manner will have an effect on performance, but it is necessary for large data objects. Giving each channel a large buffer will constrain resources and is therefore not a suitable option. The other approach is to use a single large shared buffer. This could require guarded access which would also reduce performance, although is a possible area of investigation in the future.

Figure 43 presents results for sending null objects via the new JCSP Networking channel. The original network channel and Object Stream results are provided for comparison. Expected values are calculated using the new *NetChan* formulae. Values are the average time in milliseconds to perform a single send or receive operation.

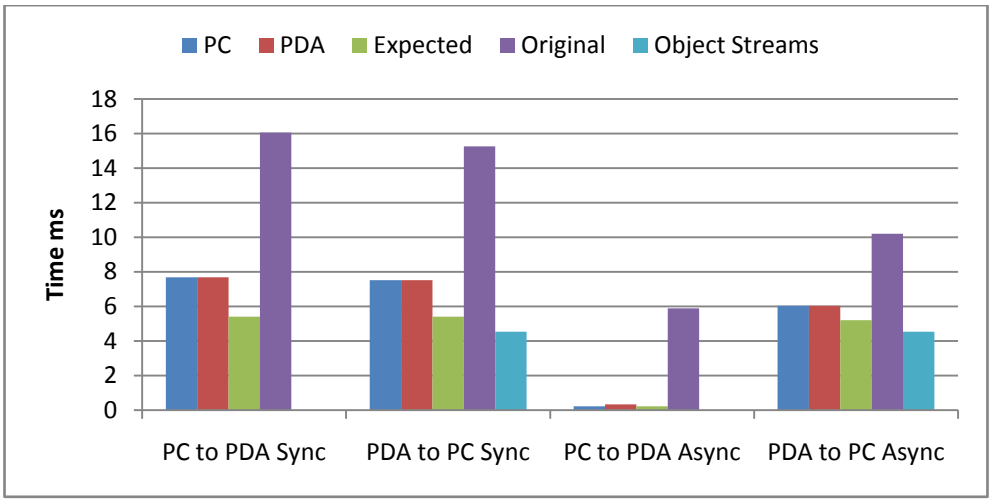


Figure 43: New Network Channel Send and Receive Benchmark

Figure 43 illustrates that the time taken to send a null message using the new JCSP Networking implementation is approximately half the time taken within the original implementation. Synchronous channels perform approximately 2 ms slower than expected, but taking into account the roundtrip latency of 1.5 ms there is an approximate 0.5 ms difference. Discounting PC Object Stream results due to the low value, PDA Object Streams are 3 ms faster at sending a simple packet than a new networked channel. Roundtrip network latency reduces this value to 1.5 ms.

Asynchronous channels for PC to PDA indicate that the high priority `Link` problem has been overcome. The PC and PDA both register low times, the PC being 0.01 ms lower than expected. Comparing this value to the original JCSP Networking implementation where high priority `Links` flooded the PDA, the PC actually gains performance as the PDA is able to service incoming messages quickly. (De)serialization of the message headers in the original implementation will also be a contributing factor to the lower figure however. PDA Async results show an improvement of approximately 4 ms, and are approximately 1.5 ms slower than Object Streams.

For actual bandwidth of the new networked channel, large data packets are sent with and without serialization. Both synchronous and asynchronous channels are examined, and expected results are provided. The performance of the original implementation of JCSP Networking and Object Streams are provided for

comparison. Figure 44 presents PDA synchronous channel results and Figure 45 presents PDA asynchronous channel results.

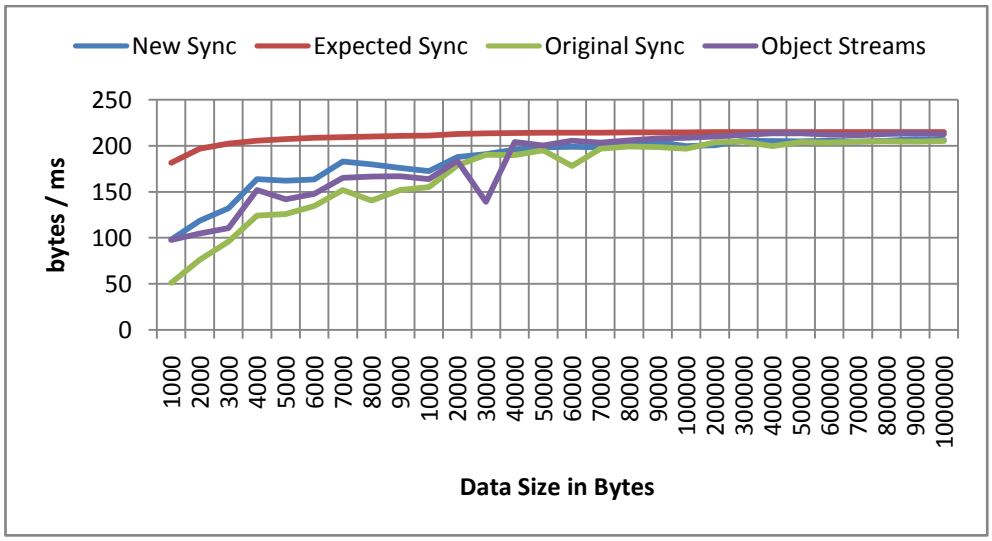


Figure 44: PDA New Synchronous Channel Bandwidth

For synchronous channels there is an initial approximate 50 bytes/ms performance improvement with the new implementation. Performance does converge over time however. Throughput within the new implementation is approximately 2 bytes/ms better than the original at the largest packet sizes, which can probably be attributed to the removal of the object message header. Object Streams have approximately 4 bytes/ms better throughput when compared to the new synchronous channel results.

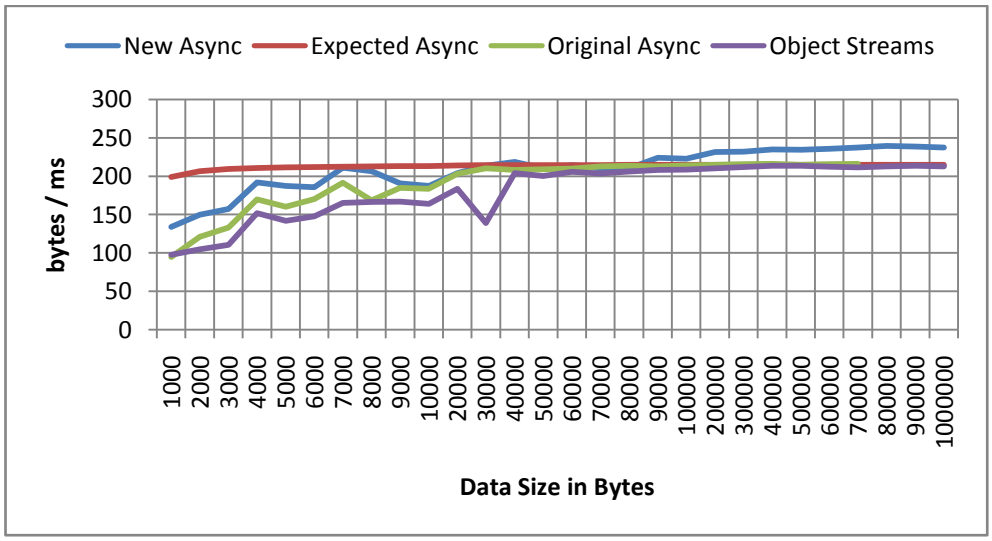


Figure 45: PDA New Asynchronous Channel Bandwidth

Asynchronous channels show improved performance before bandwidth values converge. The new implementation does show improved performance for larger packets, and this will be due to the reduction of `Link` priority. Under these conditions, no memory exception occurs due to large packet sizes flooding the PDA, and thus results continue to the maximum data size, unlike the original implementation. At large packet sizes, performance is approximately 20 bytes/ms better than expected. This will be due to the `Link` performing the actual I/O when the application process has finished. This is also why Object Streams show poorer performance than the new implementation.

Results for sending the data without serialization are provided in Appendix D. There is little performance difference between the serialized and raw data results. Asynchronous channels perform approximately 8 bytes/ms faster for large data sizes, but this will largely be due to the buffering problem described at the start of this section.

Results for PC synchronous channels are presented in Figure 46 and asynchronous channels in Figure 47. Object Stream results are provided for data sizes greater than 3,000 bytes due to the large bandwidth value that small packet sizes provide. For asynchronous channels, all presented results are for data sizes greater than 3,000 bytes due to large bandwidth values. As serialized and non-serialized values are similar, the latter are provided in Appendix D.

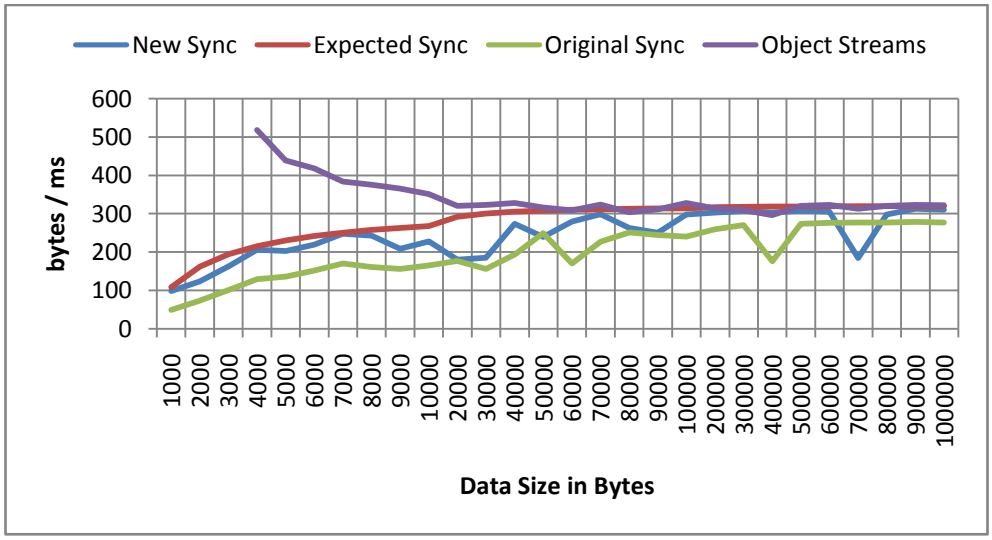


Figure 46: PC New Synchronous Channel Bandwidth

For synchronous channels on the PC there is a performance improvement for all data sizes except for a valley at 800,000 bytes. The new implementation provides an approximate 35 bytes/ms improvement, reaching 310 bytes/ms. This is 10 bytes/ms lower than expected, which is similar to PDA throughput variance between actual and expected results for synchronous channels. The improvement in performance can be attributed to the PDA not having to deserialize the incoming message header.

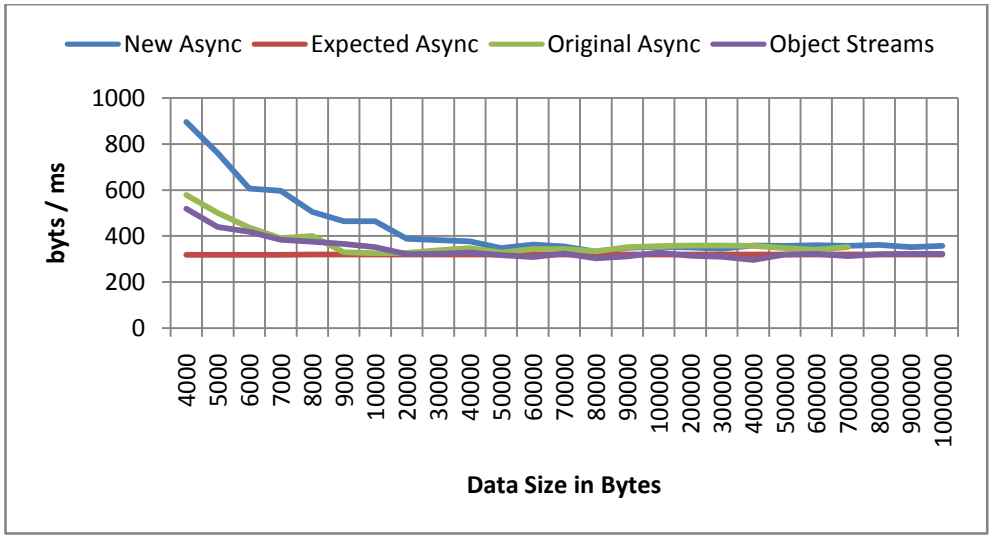


Figure 47: PC New Asynchronous Channel Bandwidth

Asynchronous results show an initial improvement within the new implementation in comparison to the original. Results converge before the original results end due to the memory exception on the PDA. Asynchronous results show a 35 bytes/ms improvement over the expected results, but this will be due to the `Link` performing I/O when the application process has completed.

6.2.3 Latency

From the ping and send experimental results (Figure 42 and Figure 43 – pages 109 and 111), latency can be estimated for null messages within the new implementation. The estimated latency is 10 ms for a roundtrip operation when compared to expected results, and 5 ms when actual ping time is compared to actual send time. Halving these values gives an approximate latency of 5 ms and 2.5 ms respectively within the new implementation of JCSP Networking.

For closer analysis, large data packets are sent via synchronous and asynchronous channels with and without serialization. The serialization scenario utilises an 8192 byte buffer within the channel. Expected and original JCSP Networking results are provided for comparison. As the values recorded for both PC to PDA and PDA to PC are similar, only PDA to PC results are presented. The other results are available in Appendix D.

Figure 48 provides results for a roundtrip operation from the PDA to the PC over the various data sizes, using serializing synchronised channels. Figure 49 presents the results serializing asynchronous channels. Expected times are generated using the *NetChan* formula. Original and Object Stream results are also provided. The values presented are the average time to perform a single operation in milliseconds.

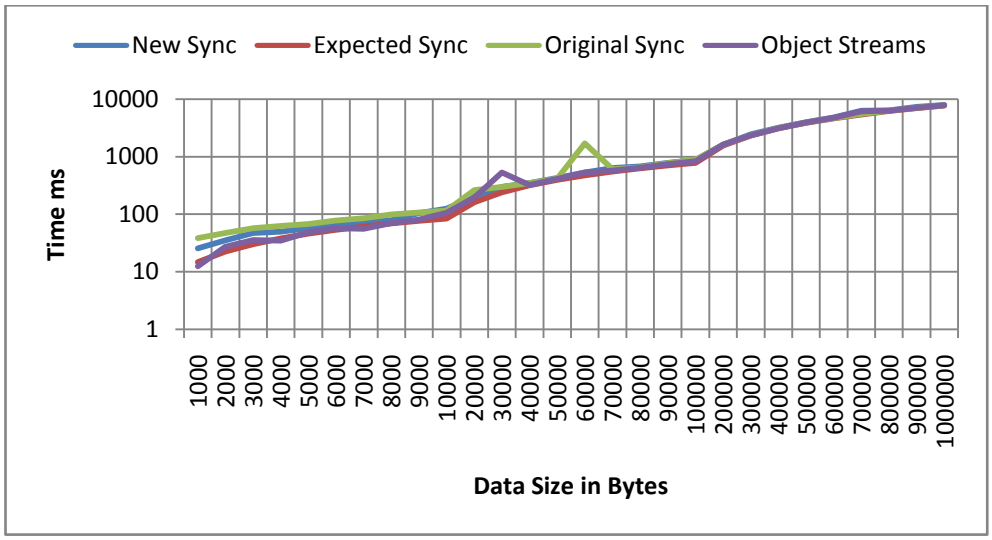


Figure 48: PDA Synchronous Serialization Channel Roundtrip

For synchronous channels, performance within the new implementation is initially better than the old implementation but poorer than Expected and Object Stream results. For larger packet sizes, the original implementation performs better than the new implementation when using serialization within the channels. The new implementation is approximately 150 ms slower than expected, and approximately 70 ms slower than the original implementation.

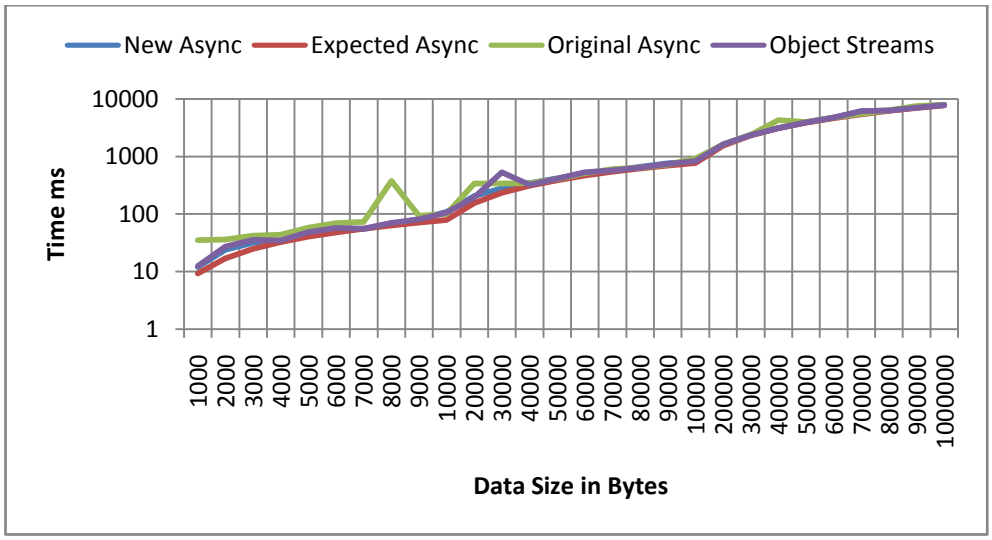


Figure 49: PDA Asynchronous Serialization Channel Roundtrip

Asynchronous channels perform similar to the original implementation and Object Streams. Performance is approximately 150 ms slower than expected. Asynchronous and synchronous roundtrip times are now within a few milliseconds variance for large packet sizes, unlike the original implementation where asynchronous performance deteriorated over time.

To determine the effect of buffer resizing, the experiment is repeated with the data sent with no serialization. Figure 50 presents the synchronous results for a PDA to PC roundtrip and Figure 51 presents the asynchronous results for a PDA to PC roundtrip. Expected times are adjusted to remove the serialization time and the extra data overhead incurred by the byte array class description.

Aside from the peak at data size 2,000, synchronous channels with no serialization perform similarly to serializing channels. There is slight improvement towards large packet sizes. Actual results are approximately 120 ms greater than expected at the largest packet size. There is an approximate 40 ms variance in performance from the original implementation at this packet size.

For asynchronous results, the new implementation performs better than the original implementation over all packet sizes. For the largest packet size, the new implementation performs approximately 70 ms better. Compared to expected results, the new implementation is 85 ms slower at large packet sizes. The

performance of the new asynchronous channel is similar to the performance of the original synchronous channel under these conditions.

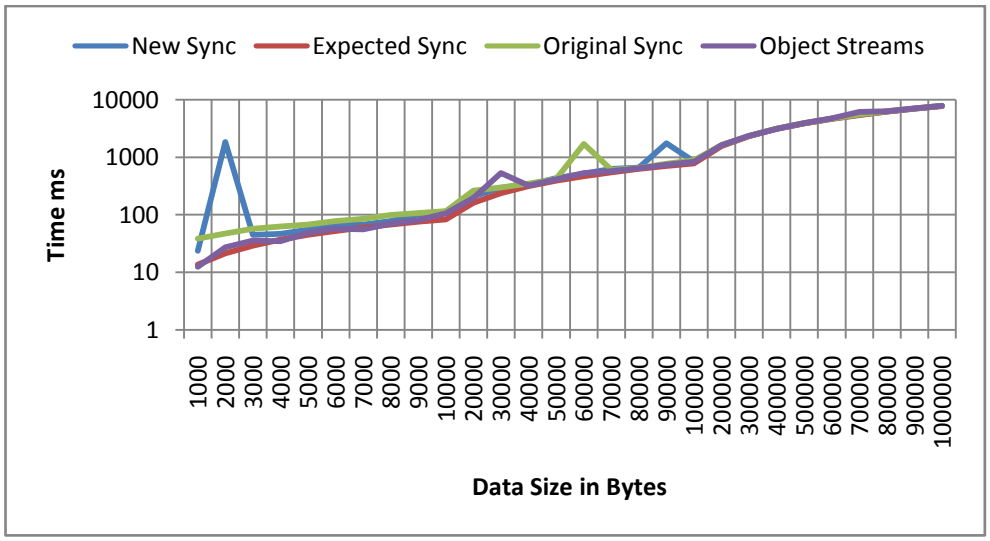


Figure 50: PDA Synchronous Raw Channel Roundtrip

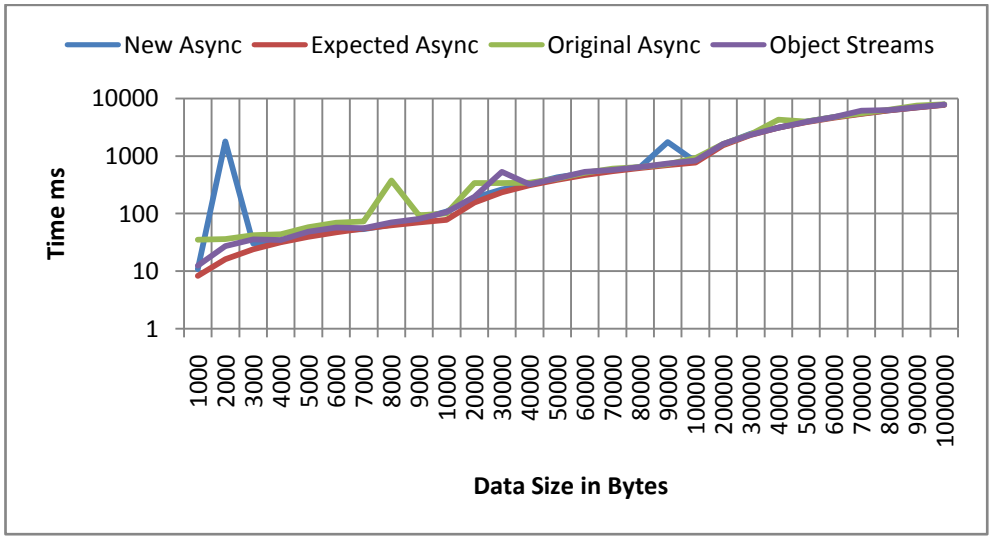


Figure 51: PDA Asynchronous Raw Channel Roundtrip

A possible cause of the performance reduction is the lower priority I/O. As an example, Figure 52 presents the results recorded on the PDA when the PC sends data to the PDA in a roundtrip asynchronous operation without serialization. For the new JCSP Networking implementation, there is an approximate 300 ms performance improvement for the largest packet size. From the bandwidth results for the new JCSP Networking implementation (Figure 44 and Figure 46 – pages 112 and 113), there is an observed increase in throughput. Lower priority I/O does mean I/O is not serviced as quickly in the new implementation compared to the

original implementation, and therefore reducing I/O priority may have increased latency. Reducing I/O priority does permit the application to handle incoming and outgoing messages, overcoming the problem of a fast device flooding a slower one. Thus, exposing the `Link` priority as a property in the new JCSP Networking implementation permits more ubiquitous usage of JCSP Networking within different scenarios.

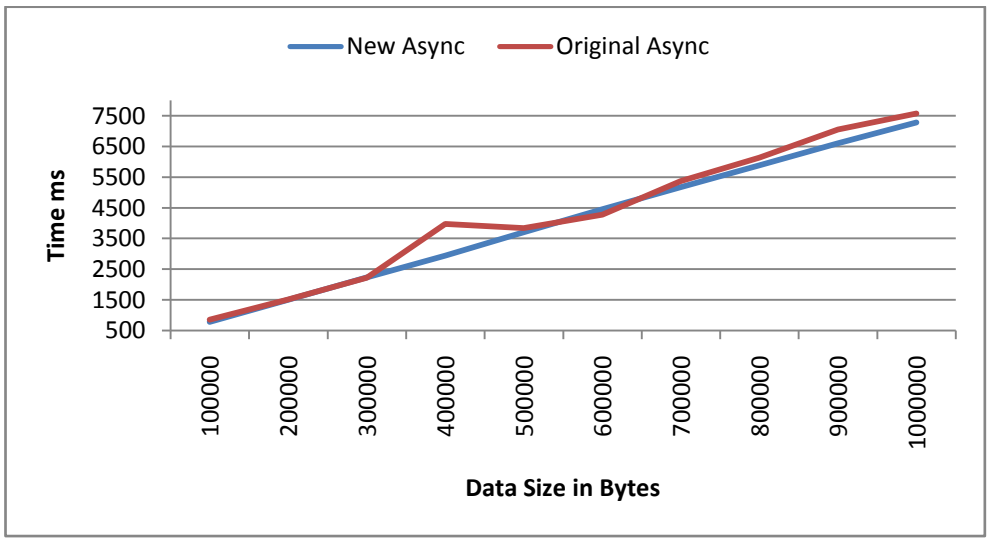


Figure 52: PDA Receiving Asynchronous Raw Channel Roundtrip

To examine the assumption that lower Link priority is effecting latency, `Links` are given maximum priority and the non-serializing synchronous roundtrip experiment repeated. Figure 53 presents these results. Here, high priority I/O results are the same as normal priority I/O. Therefore high priority does not account for the differing performance. A further possible explanation is the removal of the `NetChannelInputProcess`, which serviced input messages prior to actual reading by the application process. This could possibly lead to faster performance.

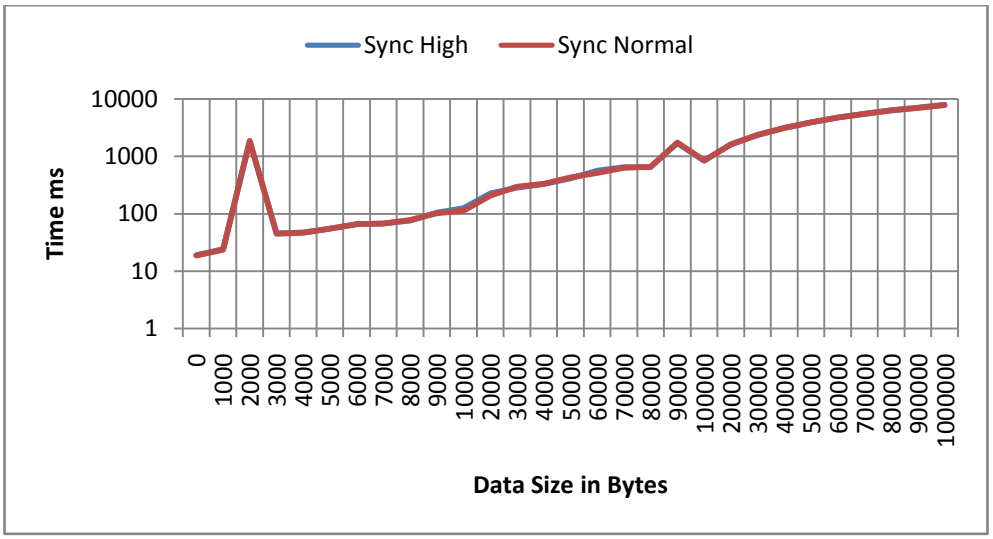


Figure 53: High Priority vs. Normal Priority Link

6.3 Test Object Messages

From experimental data presented in Section 4.5.2, transmission time of the various test objects was shown to be bound by PDA serialization performance except during asynchronous sending by the PC to the PDA. The new implementation should likewise be serialization bound, although improvement should be evident because of the removal of the object channel message header. In this section, the test object experiments are repeated within the new implementation, with original and expected results provided for comparison. Only `TestObject4` results are presented within this section as these provide enough insight into performance. Other results are available in Appendix D.

6.3.1 Sending

Figure 54 presents results for the PC sending `TestObject4` to the PDA via synchronous and asynchronous communication within the new implementation of JCSP Networking. Original Sync and Async results are also provided, as are expected Sync and the underlying Object Streams. The values presented are the average times taken to perform a single send operation in milliseconds. The x-axis represents the size of the sent object in bytes. Any significant peaks have been removed to allow better analysis, with actual results being provided in Appendix D.

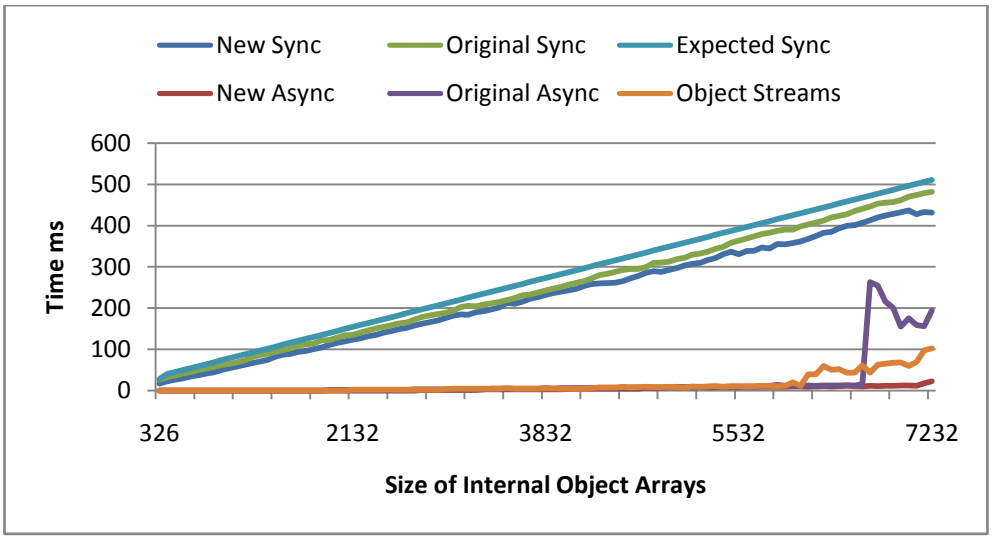


Figure 54: PC Sending TestObject4 via New Networked Channel

For Sync channels in the new JCSP Networking implementation there is a performance increase in comparison to the original implementation. At $\texttt{TestObject4}_{100}$ this improvement is approximately 50 ms. Considering the deserialization time on the PDA for the object message header (approximately 16.5 ms) and the serialization time for the acknowledgement (approximately 10.5 ms) there is an approximate 23 ms variance between the original implementation and new implementation results. Performance of the new implementation appears to level slightly at size greater than 96. $\texttt{TestObject4}_{95}$ has a peak in the new results, so $\texttt{TestObject4}_{94}$ is compared between the new JCSP Networking implementation and the original implementation. There is an approximate 30 ms variance between the two results at $\texttt{TestObject4}_{94}$, which is 3 ms greater than the overhead removed due to (de)serialization of message headers.

New Async results do not show the same jump in time taken as the Original Async results. There is a slight increase in the time taken to perform an operation at large object sizes, but the non-object message header and lower priority `Link` seems to have removed this overhead. Asynchronous channels also perform better than Object Streams, though this will be due to the `LinkTX` process performing the I/O thus reducing application I/O time.

Figure 55 presents the results for the PDA sending `TestObject4` to the PC utilising the new implementation of JCSP Networking. Due to similarities between Sync and

Async results within the PDA only the Sync results are presented here. No peaks have been removed from these results.

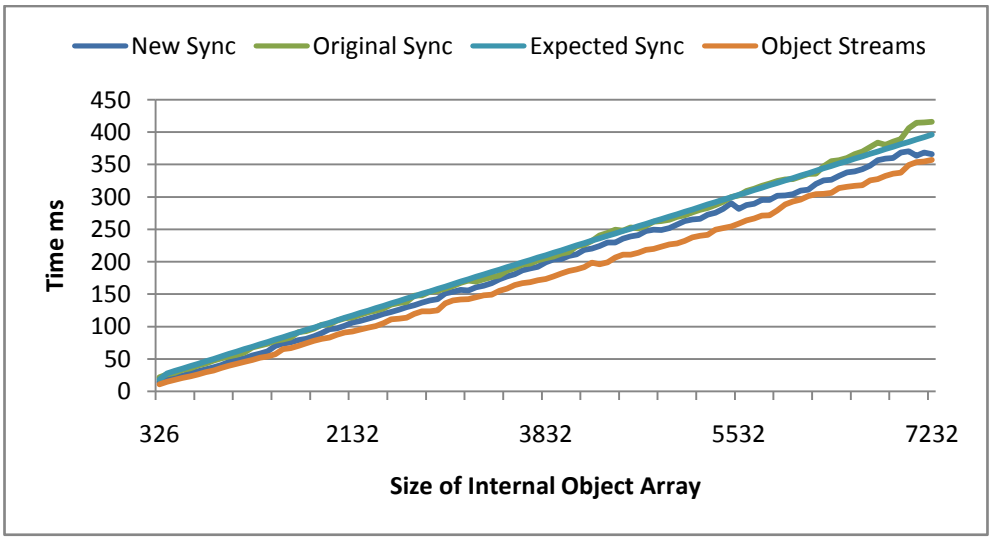


Figure 55: PDA Sending TestObject4 via New Networked Channel

As expected, the new implementation shows similar performance to the original implementation for sending `TestObject4`. There is an improvement in performance for large sized objects, and this will be partially due to the removal of the object message header in the original implementation. Performance of the new channels is only slightly poorer than Object Streams for larger sizes, and would appear to increase more or less in unison with the Object Streams. This is unlike the original implementation where the performance difference between the two results appears to widen.

From Figure 54 and Figure 55 it is possible to estimate throughput of the PC and PDA when performing complex serialization using the new networked channel. For the PC this figure is approximately 16.5 bytes/ms and for the PDA approximately 19.5 bytes/ms.

6.3.2 Roundtrip

Figure 56 presents synchronous roundtrip results from PC to PDA for `TestObject4` within the new JCSP Networking implementation. Expected, Original Sync and Object Stream results are also provided for comparison. PDA to PC results are

similar, and Async results show only slight improvement. Thus these results are provided in Appendix D.

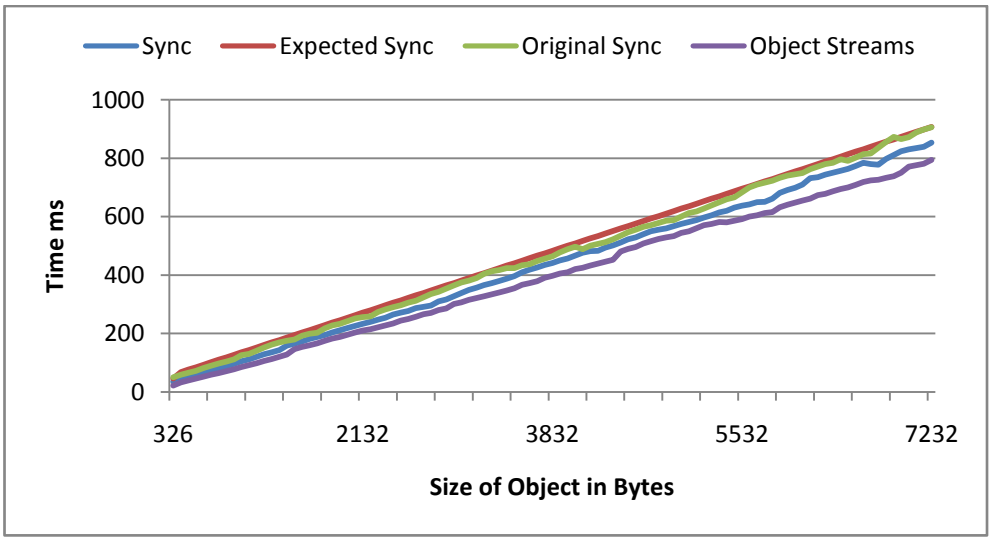


Figure 56: PC to PDA TestObject4 Synchronous Roundtrip via New Networked Channel

For the new implementation, there is some improvement in comparison to the original implementation. This is due to the removal of the object message header. Results are better than expected, but this does not hold for all test object types and is related to the different serialization performance between the objects. The roundtrip operation for `TestObject4` in the new implementation is 60 ms slower at $\texttt{TestObject4}_{100}$ than when performed via an object stream.

6.4 Overhead of the New Implementation

It has been shown that the reduction in `Link` priority has reduced the chance of potential flooding on a resource constrained device. To analyse the reduction in performance overhead that lowering the priority has caused, the CommsTime with roundtrip data experiment is repeated. The lower priority I/O should enable the PDA to record generally lower CommsTime values, while the roundtrip time should increase. Figure 57 presents the CommsTime results for the PDA when performing roundtrip operations of large data sizes with serializing channels. Vertical lines indicate when a time is recorded for one of the data sizes. Gaps in the blue CommsTime line indicate when no CommsTime value was recorded between the packet sizes.

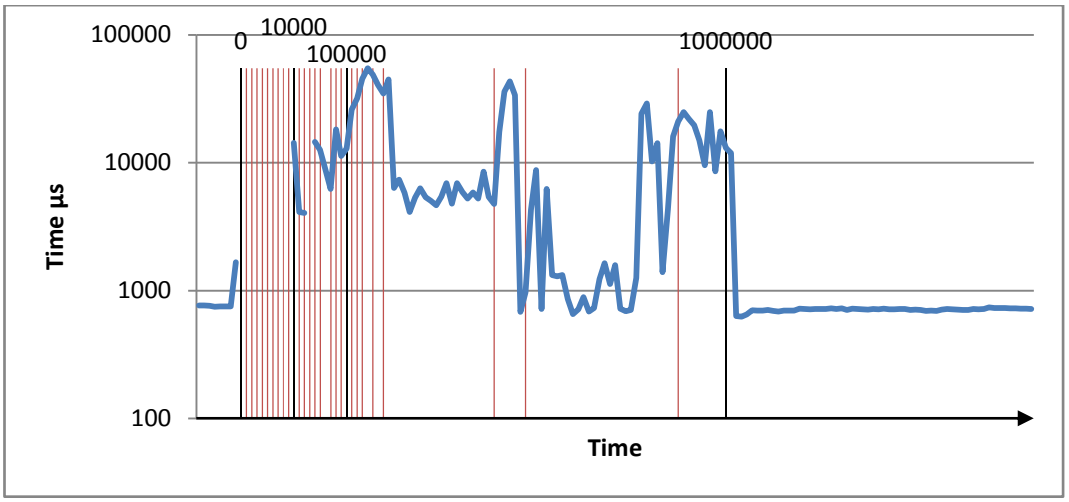


Figure 57: PDA CommsTime New Stressed Network

Comparing Figure 57 against Figure 33 (page 82), the maximum CommsTime value recorded has been reduced from approximately 70 ms to 55 ms. This value is still a significant increase in the CommsTime figure in comparison to the approximate 680 µs time without I/O operations.

Figure 58 presents the recorded roundtrip time on the PDA for large data packets while performing CommsTime. There is little variance between the times recorded with CommsTime and without, except towards the larger packet sizes. Here, performance varies, with the roundtrip time With CommsTime sometimes performing better, and likewise Without CommsTime. The variance between the two result sets can reach approximately 180 ms.

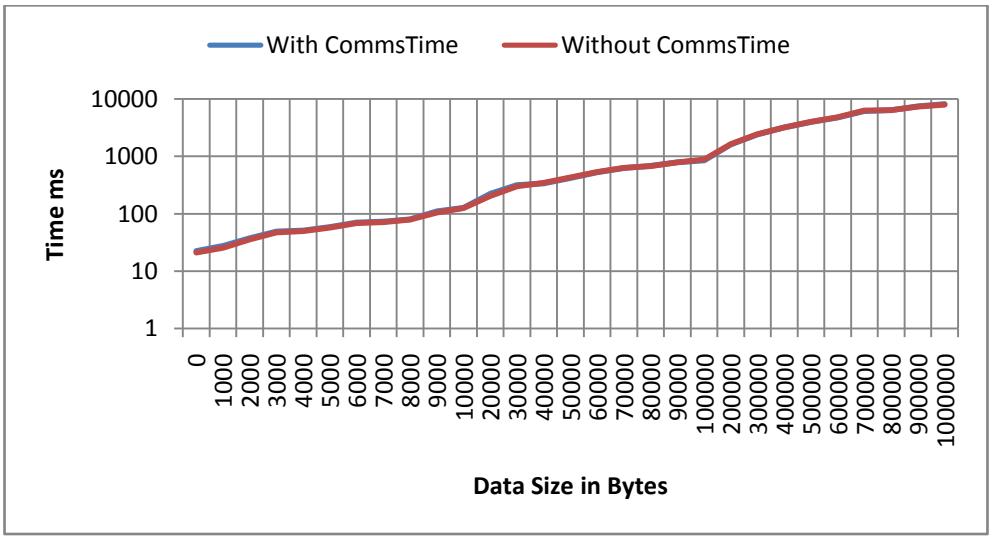


Figure 58: New Networked Channel Roundtrip with CommsTime

It can be judged that the overhead when implementing lower priority I/O has been reduced, but can still be considered as significant. I/O does not appear to suffer due to lower priority. There may be some effect from the removal of the (de)serialization of the message header from the original JCSP Networking implementation, as on the PDA this has been shown to be a slow operation and thus takes some CPU time.

6.5 Verifying the Protocol and Architecture

One of the problems highlighted in the original JCSP Networking architecture was poor error handling. `NetChannelOutputs` can fail due to `Link` failure when the subsequent exception is not passed to the application level. In the new JCSP Networking implementation, `NetChannelOutputs` are registered with the relevant outgoing `Link`. If the connection fails, the `Link` iterates through the list of registered channels and signals each in turn.

To determine whether registering with the `Link` is enough to avoid the output channel hanging, a model of the new architecture and protocol has been developed with the SPIN Model Checker [141]. The development of the model also allows general verification of the architecture to check that it is deadlock free, as well as examination of properties that are of interest.

6.5.1 SPIN

SPIN (Simple Promela INterpreter) is a model checker that allows examination of properties within a derived model by thoroughly checking the state space of the model. SPIN can verify a number of correctness requirements by usage of assertions, checking for deadlock, fairness and liveness of the defined model. The underlying language used to build a SPIN model is Promela (PROcess MEta LAnguage), which has similar semantics to CSP (e.g. channels, processes, choice). SPIN is similar enough that is possible to almost directly compose a JCSP application into a SPIN model for verification.

To verify a model, SPIN converts the Promela code into C code, which is then compiled into an application. The application attempts to verify the model by

creating the entire possible state space of the model, thus visiting every possible state that the model can be in. If at any point it is impossible for the application to move to another state and the model is not in a correct end state, then verification fails. This is a basic deadlock check, and other properties can be checked from the built state model.

Normally, CSP based architectures are verified using the FDR tool [142], which allows verification of a model based on properties such as deadlock and livelock, and also provides refinement checking. Refinement checking allows comparison of a model against expected behaviour. CSP does not at present incorporate channel mobility however, and neither does FDR. Although it is possible to circumvent channel mobility directly by passing values that represent a channel [143], it is not strictly channel mobility.

SPIN does permit channel mobility by passing channels as parameters in a message. The SPIN channel is similar to a channel in JCSP, although SPIN permits guarded operations on shared channel ends. As JCSP does not allow these operations, they are not used within the new JCSP Networking implementation. Thus the SPIN model of the architecture does not utilise such operations either.

The full SPIN model of the verified architecture is provided in Appendix F. Here, a high level description is provided. The model represents only the channel operations and architecture within the new implementation. First a description of the protocol messages is provided.

6.5.2 Protocol Definition

SPIN uses the `mtype` keyword to define message types. From the discussion presented in Section 5.2, six message types within the protocol are relevant to channels. The ASYNC_SEND operation cannot be modelled as it can occur at any point during execution and requires no synchronisation between communicating components. This would increase the state space of the model beyond the capabilities of the model checker. An argument on its verification shall be presented at the end of this section.

Discounting the ASYNC_SEND message, `mtype` is defined as follows:

```
mtype = {SEND, ACK, REJECT_CHANNEL, POISON, LINK_LOST};
```

6.5.3 Channel

A networked channel has a number of required definitions: the possible channel states, the data structure representing a channel, and the processes that represent `NetChannelInput` and `NetChannelOutput`.

6.5.3.1 Channel States

Figure 41 (page 104) presented the possible states and state transitions of the new networked channel. These states are given constant values and added to the model.

6.5.3.2 Channel Data Structure

Each channel is provided with a data structure that contains the Virtual Channel Number, the state and the channel that the `Link` uses to communicate with the channel object. This is defined as follows:

```
typedef CHANNEL_DATA
{
  byte vcn;
  byte state = INACTIVE;
  chan toChannel;
}
```

6.5.3.3 Channel Process

SPIN uses processes to represent components, thus a networked channel must be represented by a process. The process is given a CHANNEL_DATA structure to represent the channel, and an interface of channels that represent the possible method calls that can be made on the channel. There are two channel types, `NetChannelInput` and `NetChannelOutput`. Figure 59 presents the `NetChannelOutput` process.

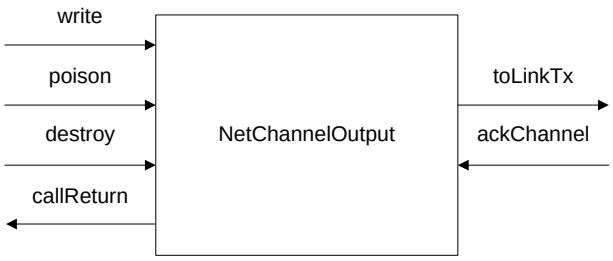


Figure 59: NetChannelOutput Process

On the left of Figure 59 the interface channels are provided. Each channel represents the calling of a method on `NetChannelOutput`, except `callReturn` which is read to simulate the end of a method call on the process.

On the right of Figure 59 are the channels connecting the channel process to the `Link` process. `toLinkTx` is a fixed channel that connects to the `LinkTx` where the input end of the virtual channel is connected. `ackChannel` is the channel coming from the `Link`, and is the channel defined in the CHANNEL_DATA type.

Figure 60 presents the `NetChannelInput` process. The method interface is on the left, and includes extended rendezvous and poison operations which were added to JCSP in version 1.1 [137], and thus require addition to the new JCSP Networking implementation. For completeness these operations are added to the SPIN model.

The `NetChannelInput` process has only one connection to the `Link` processes, the `fromLink` channel. This channel is the same as declared in the CHANNEL_DATA type. When a `Link` sends the `NetChannelInput` a message, it also sends the channel to send the response back to the `Link`. This is where channel mobility is required.

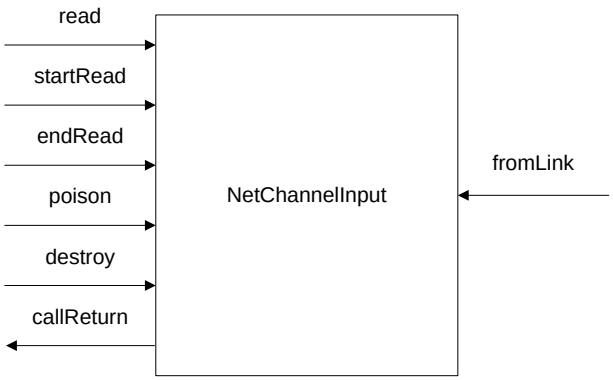


Figure 60: NetChannelInput Process

6.5.4 Link Processes

`Link` contains two processes: `LinkTx` and `LinkRx`. `LinkTx` receives messages from the channel processes and sends them to the remote `LinkRx`. Figure 61 represents the `LinkTx` process.

Figure 61: LinkTx Process

`input` receives messages from the channel processes. `txStream` represents the connection to the remote `LinkRx` process.

`LinkRx` receives messages from a remote `LinkTx` and sends them to the correct channel. It is represented in Figure 62.

Figure 62: LinkRx Process

`rxStream` represents the incoming stream from the remote `LinkTx`. `toLinkTx` connects to the complement `LinkTx`, and is used to send messages directly to the `LinkTx` and to attach to incoming messages to allow a subsequent acknowledgement to be sent directly to the `LinkTx`.

A `Link` represents a connection to another node, and is composed of a `LinkTx` and a `LinkRx`. Figure 63 represents the `Link` process.

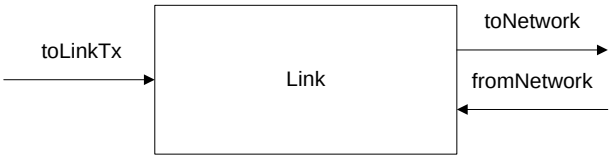


Figure 63: Link Process

6.5.5 Application Processes

There are two types of application process: an outputting application and an inputting application. These processes operate on the complement end interface channels that connect to a `NetChannelInput` or `NetChannelOutput`. The application process chooses non-deterministically to write to one of the method call channels and then reads from the `callReturn` channel, thus waiting for the operation to complete. `callReturn` returns either 0 or 1 to represent either an EXCEPTION or an OK return message. If an EXCEPTION is returned, then the application process terminates.

Full details of these processes are available in Appendix F, and are named `Sender` for an outputting application and `Receiver` for an inputting application.

6.5.6 Node

Within the model, two node types are defined: `InputNode` and `OutputNode`. An `InputNode` starts a number of `Receiver` processes with relevant `NetChannelInput` processes. An `OutputNode` starts a number of `Sender` processes and relevant `NetChannelOutput` processes. Figure 64 presents the `InputNode` process. The connection between the `Link` process(es) and the `NetChannelInput` process(es) is shown, although this is dynamic.

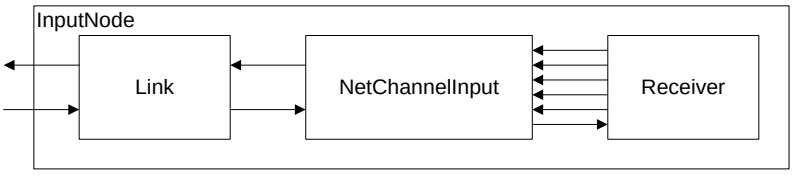


Figure 64: InputNode Process

Figure 65 illustrates the `OutputNode` process. In this circumstance, the connection between the `NetChannelOutput` and `Link` is static.

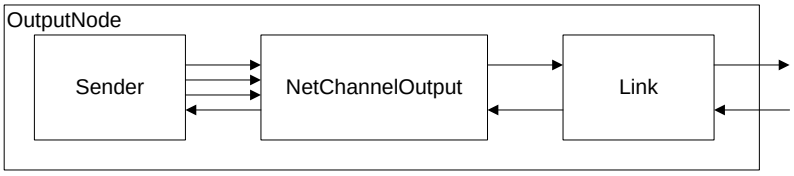


Figure 65: OutputNode Process

For both Nodes, the `toNetwork` and `fromNetwork` channels represent the `txStream` and `rxStream` across the network connecting the two remote Nodes. A process is also added that allows simulation of the network connection itself.

6.5.7 Network Process

To simulate network failure, a simple process to represent the network is added to the model. The process non-deterministically chooses to either send a message from the `OutputNode` to the `InputNode`, from the `InputNode` to the `OutputNode`, or fail and break the connection. In the later case, a LINK_LOST signal is sent to the two corresponding `LinkRx` processes, and a flag is set which the `LinkTx` processes check to determine if they should fail. Within the JCSP Networking implementation, the latter occurrence is detected when the `LinkTX` process tries to write to a closed stream. The setting of a flag achieves the same outcome.

Figure 66 presents the overall SPIN model developed for JCSP Networking. The two nodes are connected via the `Network` process.

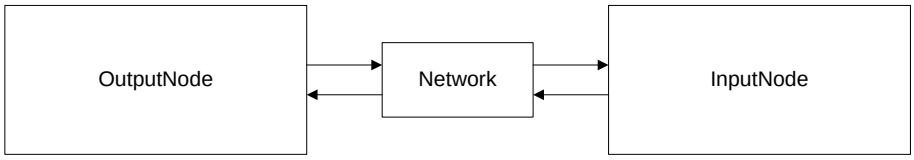


Figure 66: Simple JCSP Networking Model

6.5.8 Global Values

There are a number of global values within the model, and these are summarised below:

- *NUMBER_INPUTS* – the total number of input channels within the model.
- *NUMBER_OUTPUTS* – the number of output channels connected to a single input channel
- *TOTAL_OUTPUTS* – the total number of output channels – NUMBER_OUTPUTS * NUMBER_INPUTS
- *BUFFER_SIZE* – the size of the buffer to the channel processes. This is used to simulate the infinite buffer within the actual application. For the application to operate, BUFFER_SIZE should equal NUMBER_OUTPUTS. This value is manipulated to verify this assumption
- *CHANNEL_ARRAY* – a type declaration for the array of channel ends on a particular node – CHANNEL_DATA channels[TOTAL_OUTPUTS]. SPIN does not permit arrays to be passed as parameters into processes, therefore this must be declared globally. For channels above NUMBER_INPUTS on the `InputNode`, the channel state is set to INACTIVE.
- `chans` – all the channels within the model. As CHANNEL_ARRAY cannot be passed as a parameter to an individual process, this is declared globally – CHANNEL_ARRAY chans[2].
- `linkLost` – the flag used to indicate `Link` failure. This is initially set to false.

6.5.9 Basic Verification

Simple verification can be carried on a model comprising of a single `NetChannelOutput` connected to a `NetChannelInput` with BUFFER_SIZE = 1. This is the default assumption that for every connected output channel end to an input channel end, there is required a single place in the buffer to avoid deadlock. When passed through SPIN, the model is verified with no deadlock errors. This is enough to reasonably assume that having the `Link` processes inform the relevant output channels of connection failure overcomes the deadlock problem in the original JCSP Networking implementation.

The model also allows some indication of other problems in the original JCSP Networking implementation. In the new implementation, the `LinkRX` process

retrieves a channel from the `ChannelManager` and locks the state object of the channel before checking said state. Thereby, `LinkRX` is the only process acting on the channel state at any one time. This allows various behaviours to occur based on the state of the channel object. This feature was added to the new implementation when the model originally pointed out deadlock due to this occurrence not being taken into consideration. As the channel object can change state based on certain calls (poison, destroy), this would have caused inconsistent behaviour within the implementation.

The original implementation used no such state variable, and `LinkRx` would send a message to a channel object based purely on availability within the `IndexManager`. The `IndexManager` would only return the connecting output channel end connected to the networked channel object. When a channel object was destroyed, it was removed from the `IndexManager` prior to any clean up operations (rejection of pending messages). Therefore, a channel either existed within the `IndexManager` or it did not. There were no other possible states as no common protected state value was exposed. This meant that much of the behaviour required for more advanced functionality (poison, mobility, barriers) was not possible as there was no method to expose these states without reimplementation of the underlying mechanisms of JCSP Networking. As the new implementation exposes these properties, this problem has been overcome.

6.5.10 Advanced Verification

The simple verified model does not allow analysis of the common assumption of JCSP Networking that the channel connected to the `NetChannelInput` requires exactly one buffer space for each connected `NetChannelOutput`. With manipulation of the BUFFER_SIZE value, this can be analysed to provide a stronger insight into this assumption. Table 8 presents results from different verification scenarios. To enable verification of the model, the option within SPIN to use minimal automata to search is activated.

Table 8: SPIN Verification Results

NUMBER_OUTPUTS	1	2	3	4
BUFFER_SIZE				
0	FAIL	FAIL	FAIL	FAIL
1	3.06×10^5 states 351 depth	FAIL	FAIL	FAIL
2	2.78×10^5 states 351 depth	3.71×10^7 states 3264 depth	FAIL	FAIL
3	2.78×10^5 states 351 depth	3.71×10^7 states 3264 depth	PASS*[3]	FAIL
4	2.78×10^5 states 351 depth	3.71×10^7 states 3264 depth	PASS*	PASS*

Table 8 illustrates that a `NetChannelInput` requires one buffer space for each connected `NetChannelOutput` for connected `NetChannelOutputs` less than four. The number of states does not increase when the buffer size is increased beyond the required buffer size, except when a single `NetChannelOutput` to `NetChannelInput` has the buffer increased from 1 to 2, although search depth does not increase. The reason for the reduction in state space could be the usage of the minimal automata search option within SPIN, or that the `NetChannelOutput` requires less state space in conjunction with the `Link` processes at BUFFER_SIZE = 2. The `NetChannelOutput` also utilises the same size buffer as the `NetChannelInput` in the model, and this could have an effect in total required states.

6.6 Summary

The performance experiments performed on the original implementation of JCSP Networking have been repeated on the new implementation of JCSP Networking. Analysing the results from the new implementation, and the description of the new architecture presented within Chapter 5, against the original implementation within the context of the problems highlighted in Section 4.7, a number of observations can be made. These are summarised in the following subsections.

[3] Results marked with a * are gathered using the bit state compression technique due to the state space size. Thus, these results are deemed as approximations.

6.6.1 Interoperability

JCSP Networking no longer relies on Java serialization, although it can utilise this functionality for convenience when necessary. As channels now have the responsibility of converting data using a specific encoder/decoder, applications can be tailored to their context. By implementing this mechanism, it is now possible to implement a reduced version of JCSP Networking on reduced versions of Java. The lack of serialization capabilities is no longer a factor for basic communication.

Removing the object message header allows interoperability beyond Java. Communication is implemented on a base protocol which can be interpreted by numerous frameworks. Data transfer is a problem due to the different approaches taken to represent data in different frameworks, but the abstraction of encoding and decoding into a user customizable manner permits mechanisms to be developed to allow inter-framework communication of data if well defined data conversion is created.

For Ubiquitous Computing interoperability is important, and JCSP Networking now exhibits a level of interoperability which enables usage within various versions of Java and, if the same communication protocol is utilised, within different frameworks.

6.6.2 Performance

The performance of the new implementation in comparison to the original implementation shows slight improvement, but this can largely be attributed to the removal of the object based message header in the original implementation. PC networked channel bandwidth has increased from 275 bytes/ms to 310 bytes/ms, 10 bytes/ms lower than the optimum 320 bytes/ms throughput of the network. The PDA shows only a 2 bytes/ms improvement, which is considered insignificant. Therefore, no adverse loss in performance is observed in the new implementation.

As expected, for complex object serialization there is still a significant drop in performance. The removal of the object message header has improved performance, but not significantly.

Latency has been reduced, and specifically simple message transfer time is approximately half of the original implementation within the test framework. For a ping experiment, there has been a further reduction from approximately 36 ms to approximately 20.5 ms for synchronous message passing. For asynchronous messaging, the figure is approximately 8.25 ms. Object Streams record a ping time of approximately 6.75 ms, thus asynchronous channels have a 1.5 ms overhead. Original channels had an approximate 10 ms overhead. Latency for larger data packets has increased however, and is likely due to the removal of the `NetChannelInputProcess`. The increase of 70 ms for a roundtrip of 1 million bytes is not a significant increase in latency however.

Asynchronous channels perform uniformly better than synchronous channels within the new implementation, which is unlike the original implementation of JCSP Networking. The increase in performance is only slight, and as I/O priority has been reduced within the new experiments this allows the application to service I/O and thus not inflict problems due to buffering. Asynchronous channels now enable the high latency to be overcome, if the priority of the I/O is suitably set. As priority has been exposed to the JCSP Networking user, this problem has been overcome.

Lower priority I/O has not affected performance observably, although there is variance when running other operations with I/O. Considering serialization as a CPU intensive operation, particularly on the PDA, reducing I/O priority enables improvement for other computation at the expense of I/O but not at the expense of (de)serialization. The (de)serialization process is performed by the application process engaging in the I/O, and as this functionality has been folded into the passive `NetChannelInput` object, (de)serialization time depends on the application process performing the (de)serialization. Thus, (de)serialization is prioritised based on the priority of the application process.

As there were no adverse performance problems within the original implementation of JCSP Networking when considering Ubiquitous Computing, besides low serialization performance on the PDA, then the new implementation

can likewise be argued that the new implementation has no adverse performance problems when considering Ubiquitous Computing.

6.6.3 Resource Usage

Process usage in the new implementation of JCSP Networking has been reduced, specifically by removing the `NetChannelInputProcess` and various management processes within the original architecture. No temporary processes are created for handshaking, and therefore the only process increases come from application processes and inter-Node connections requiring `Link` processes. The latter is still a problem, and can be overcome by using polling statements on incoming connections, which has been shown to further improve performance [144]. This feature is not available in reduced Java versions, therefore cannot be implemented as a solution for resource constrained devices in all occurrences. However, as stated, it may be that small devices exhibit only a single incoming connection from another device, and `Link` processes are no longer a factor.

JCSP Networking no longer relies on a JVM capable of object serialization, and thus an initial problem of requiring a resource heavy JVM has been reduced. However, JCSP Networking is still implemented within Java, and as argued, Java may not be available within all devices. The introduction of a communication protocol which does not require Java serialization enables native applications to communicate with a JCSP Networking system utilising the same communication methods, but data encoding would still need to be agreed upon. Reliance on Java is therefore reduced, which is more practical for Ubiquitous Computing on a larger scale.

6.6.4 System Overhead

As the new implementation of JCSP Networking does not have fixed, high priority I/O, intense I/O operations do not impose as large an overhead when other computation is occurring. As the priority of the I/O is now flexible, higher priority I/O can be enabled for high computation to low communication scenarios, whereas lower priority I/O can be utilised in high communication to low computation scenarios. This reduces the risk of smaller devices being flooded, and enables a

more ubiquitous use of JCSP Networking beyond the cluster computing scenarios originally designed for.

Overheads associated with the object message header have been removed. Message headers are now relatively small, being at most 13 bytes in size, reduced from 249 bytes.

6.6.5 Scalability

As resource usage and system overhead has been reduced within the new implementation of JCSP Networking, it can be argued that scalability has likewise improved. There may still be scalability issues when considering multiple incoming connections into a single Node, although it may be possible to reduce this overhead. JCSP Networking is now more suitable for Ubiquitous Computing architectures, but not necessarily ideal. Java is still considered a problem, although the introduction of a protocol means that Java is not necessary on every device. As argued, applications with thousands of mobile agent processes are still difficult for Java to accommodate.

6.6.6 Stability

Error handling within the new implementation of JCSP Networking has been improved in comparison to the original implementation. Exceptions are now passed to the application level processes, and the problem of a `NetChannelOutput` becoming blocked while awaiting an acknowledgement from a disconnected Node has been overcome. By permitting better error handling, the usage of JCSP Networking within a Ubiquitous Computing environment has been improved, although further experiments will be required to fully analyse potential failures and how they are handled by the JCSP Networking architecture or passed to the application level processes.

6.6.7 Accessibility and Extensibility

Internal properties within JCSP Networking have now been exposed. This allows some modification of the architecture to suit individual purposes. The exposing of data encoding to the user also enables user specified data transfer. The

enablement of multiple configurations allows ubiquitous usage of JCSP Networking, and allows the numerous scenarios Ubiquitous Computing requires.

Extensibility has also been improved, and the interfaces allowing custom communication mechanisms have been simplified. However, adding new primitives to the Event Layer still requires access to the source code and modification of the `Link` processes. The layered architecture makes this simpler to achieve.

6.6.8 Conclusion

In this chapter, the experiments conducted on the original implementation of JCSP Networking have been repeated within the new implementation. Many of the issues raised about the original implementation have been overcome, without any adverse effects on performance. There are still problems when considering JCSP Networking within the context of Ubiquitous Computing, but these are now centred on limitations of Java and JVMs available on resource constrained devices. The introduction of a protocol enables communication outside Java, and the abstraction of data encoding further enables inter-framework communication.

Chapter 7 Channel Mobility

Previous chapters have focused on the properties of JCSP Networking when applied to a resource constrained environment. Consideration of JCSP Networking as a Ubiquitous Computing framework must also take into account the dynamic topologies required, with consideration on the practicalities of distributed process and channel mobility. In the following two chapters mobility is examined in this context. Mobility is seen as a key feature when considering JCSP Networking as an architecture for Ubiquitous Computing, as it provides the dynamic capabilities that are considered important in such an implementation context. Thus far the information presented has shown that JCSP Networking has no significant communication overhead in comparison to standard networking when considering the reduced framework the experiments have been conducted in, although resource usage over time may be a concern. In this chapter, various approaches to channel mobility are presented, with various properties examined in the context of each model. In Section 7.1 a definition of channel mobility is provided. Section 7.2 summarises potential channel mobility models, and Section 7.3 analyses properties of these models, Section 7.4 summarising these properties. Finally, Section 7.5 draws some conclusions on the suitability of these models within a Ubiquitous Computing scenario.

7.1 Defining Channel End Mobility

As discussed, channel mobility is the capability to migrate a connection from one component to another. The π-Calculus [10] models channel mobility by allowing names to be passed between process contexts. Mobility in the π-Calculus allows channel identifiers to be copied from one process to another, rather than strictly moved. However, if the location that a name is migrating from no longer utilises the name, then the channel name becomes unbound from the original location, and

thus is moved rather than copied. When a name arrives at a process, it becomes bound at that location. For example, Figure 67 presents a process tree, with the output end of channel *a* communicated from *V* to *R* via channel *b*. If *V* no longer uses *a* then it becomes unbound in *V*. Within *R*, *a* becomes bound. *W* has no knowledge of the migration of *a*, and *R* now has a new connection to *W*.

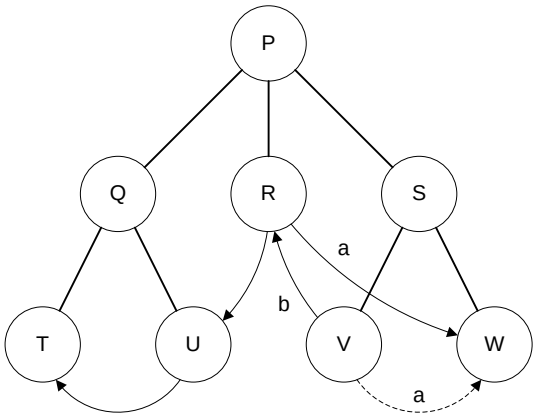


Figure 67: Channel Mobility

Figure 67 indicates how channel end mobility is achieved. It requires a channel end to be passed by another channel, or a communication that is communicable via another form of communication [88]. A simple analogy is that *R* has been provided with an address to communicate to *W*. As Chapter 5 discussed, the underlying mechanism of JCSP Networking relies on channel addresses, thus mobility is occurring on a very basic level as addresses are passed between Nodes.

From this description, it is possible to define what a mobile channel looks like at a basic level, which is essentially an address. As JCSP Networking utilises a channel end mechanism, it can be argued that output channel mobility is a case of migrating the address of the input end of a channel to another location. Input channel migration is more complicated, and this chapter focuses more on mechanisms to enable input channel mobility. Most models allow input channel mobility via address mobility also, although there are exceptions as highlighted in Appendix G.

7.2 Channel Mobility Models

Analysing current techniques for connection mobility, it is possible to extract seven different models that enable channel mobility. In this section, these seven models

are presented. The relevant interaction sequences, state diagrams and new protocol messages for these models are available in Appendix G and can be used to help illustrate exactly how these models operate. For the discussion presented here this is not necessary, and the general description is enough to analyse interesting properties.

7.2.1 One-to-One Networked Channels

Networked channels are generally considered to be Any-to-One in that any output end may connect to an input end. This makes mobility difficult as it is unknown how many output ends may be connected to an input end, and therefore informing output ends of the movement of an input end is not a one-to-one communication.

Muller [145] has presented a mobile channel protocol that utilises a one-to-one channel mobility model. Channel end (port) states vary based on whether the port is locally connected or remotely connected, and ports are aware of the address of their companion port. A full explanation of channel states can be found in [145]. A port is aware of the location of its companion and informs the companion of the new location on arrival. Mobility is easier in comparison to the Any-to-One model of networked channels as it can be guaranteed that the companion port has been notified of the new location.

The main disadvantage with this model is that networked channels become One-to-One connections instead of Any-to-One. This is not a major drawback as, if an Any-to-One architecture is required, a multiplexing process can receive from multiple processes and send to a single process. This incurs an overhead for transmission time, and requires a fixed process for each such channel. If there are few such channels required these limitations may be considered inconsequential.

7.2.2 Centralised Server

Mobile channel ends controlled by a server is the approach taken in the pony framework [120, 130]. Each channel is allocated an identifier unique to the application context (the set of Nodes that make up a single pony application). These identifiers are managed by a server which keeps the current location of the

channel. As the channel end is migrated, this location is updated. An output end connected to an input end can resolve this location, and then connect directly to the input end. If the input end should later move, the output end retrieves the new location from the central server. Therefore a channel end can be thought of as either being at the given location or not – in which case the server is checked for the new location.

The server requires messages to allow registration, resolution and updating of channel locations. The current JCSP Networking Channel Name Server implements most of these functions. pony has separated the functionality into two separate components, an Application Name Server, which allows registration and resolution of applications as opposed to channels, and a main node for each application. The main node is responsible of controlling channel mobility.

7.2.3 Message Box

Message boxes are the approach used for mobile agents [89], and was previously proposed as the model for JCSP Networking channel mobility [17]. The Node declaring the `NetChannelInput` creates a message box, which allows the `NetChannelOutput` to send to a single address, and the `NetChannelInput` to request the next message from the message box. The message box is fixed, so there are no new channel states, although the message box will require its own state model.

The main disadvantage of message boxes is that the Node declaring the message box must remain operational. As the declaring Node's execution may complete before the mobile channel end is no longer required, this can be a severe limitation.

7.2.4 Message Box Server

Message box and server models can be combined by creating message boxes on a server instead of locally [63]. Apart from the requirement of server creation, the operation of the message box is identical to the message box.

Utilising a server overcomes the main disadvantage of the message box, but does so by having all messages channels pass through the server node. Thus there is a

bottleneck in the architecture. This can put strain on a server Node, although multiple servers may overcome the problem.

7.2.5 Chain

The chain approach to mobility [51] requires each previous location of a channel end to forward messages on to the next location. When an input end arrives at a new location it informs the previous location of the new location. When an output end moves the previous location is sent with the migration message, which is used to send to the previous location. Thus a chain of connections is created, and any message must traverse the entire chain to get from one end to the other.

In the Any-to-One network channel architecture, there will be chains of various lengths in operation. The length from the original input location to the current input location is always determined by the number of migrations that the input end has made. The length of the output end(s) depends how far the outputting end has moved from its original location. Thus, as different output ends may traverse different distances, there will be multiple chain lengths in operation.

The main disadvantage of the chain model is the distance travelled for each message. The chain may also contain loops. A loop occurs when a message travels through the same node more than once. Each previous location of a channel end is a link in the chain and a channel end may move to any location during operation, therefore loops can be formed if a channel end moves through a Node where a link in the chain already exists. A further disadvantage occurs when a Node fails, which can cause multiple chains to break.

7.2.6 Reconfiguring Chain

To overcome the loop and transmission time problems of the chain model [59], the chain can reconfigure itself by finding shortcuts to a previous link if it is accessible. Any loop is therefore removed, and transmission time may become reduced whenever the chain is shortened.

To achieve reconfiguration, a migrating channel end takes all previous locations in the chain. On arrival, the locations are iterated through and reconnection is

attempted to the oldest possible link in the chain. Loops are removed as a Node can always shortcut to itself. Transmission time for messages can be reduced as the most direct route between two nodes is used instead of the total distance covered by the mobile end.

7.2.7 Mobile IP Model

Mobile IP [146] is used for physical device mobility within IP based networks. Connections are registered with a home agent which is responsible for forwarding messages onto the current location of the connection. When a connection migrates, it informs the home agent, which buffers messages until the new location is resolved. The new location address is generated by a foreign agent within the domain of the connection's new location. The home agent forwards received messages to the foreign agent, which forwards messages to the connection's new location. Whenever the mobile end moves, the foreign agent informs the home agent, and the same migration process occurs.

To enable mobility between sub-domains, tunnelling is used to allow messages to be sent to the new foreign agent. Tunnelling can be reproduced in a mobile channel context by utilising a chain of foreign agents that forward messages to the respective channel end location or next foreign agent. In effect, this creates a hybrid model of chaining, server and message box. The foreign agents act as gateways between domains.

The main disadvantage of this model is that there may still be loops within the chain of foreign agents. A mobile node may send to another mobile node within the same domain, but the message would travel to the home agent first – which may be within another domain. Intelligence built into the foreign agent may remove these loops, providing direct connection, but would require more complex reconfiguration of the architecture.

7.3 Analysing Channel Mobility Models

For analysis of the different channel mobility models, the layout of standard TCP/IP based communication networks is used. A network domain may consist of several

sub-domains, which may themselves consist of sub-domains. At the root of the domain tree is the global domain. Each node in the tree can be allocated an identifier to represent the domain in the hierarchy. Messages are sent between members of domains; messages being the communication from one machine to another. Figure 68 presents an example domain tree.

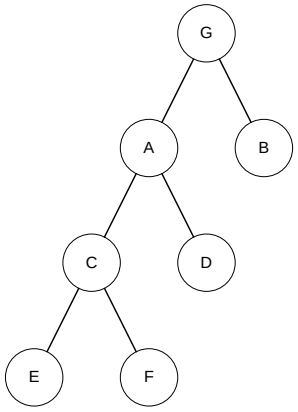


Figure 68: Domain Tree

Each node in the tree has an identifier based on its domain branch. For example, leaf E has the identifier *G.A.C.E*. A simplistic viewpoint is taken to connectivity in that members of a sub-domain may connect to a member of its parent domain. Thus any leaf in the tree can connect to any domain further up its branch until the global domain root node is reached. For example, a member of leaf *G.A.C.E* can connect to a member in three other domain nodes: *G.A.C, G.A* and *G*. This form of connectivity will be called addressability, implying that the node can address a member in a given domain unambiguously.

This view of addressability is taken to represent the fact that members of a given sub-domain may be given addresses which are also used in another sub-domain. For example, domain *G.A.C.E* may provide members with IP addresses in the standard local domain form 192.168.x.x. Domain *G.A.C.F* may also use the local domain addressing mechanism. Thus, a member of *G.A.C.E* may have an IP address 192.168.1.1, and so might a member of *G.A.C.F*. The domain tree structure ensures that this is not a problem.

As a sub-domain may address its parent domain, then it becomes obvious that a member of the parent domain may be connected to a member of a sub-domain.

However, this form of connection must be initiated by the member of the sub-domain. Therefore, connectivity is permitted down the domain tree, but not addressability. For the purposes of discussion, messages can travel either up or down the tree, but not both in a single operation. A message travelling up or down must be received by a domain member before it is sent in a different direction through the tree.

The analysis presented represents input channel end mobility, as this is the most complicated to achieve. For an output end, the majority of models permit the address or some other representation of the input channel end to be sent and a new output end to be created, effectively copying the output end at a new location. The π-Calculus [10] permits this form of copy name passing, and therefore can be considered to not be incorrect from the modelling viewpoint either. There are some exceptions which are presented in Appendix G.

To aid in analysis, a number of values are defined:

- *PROTOCOL* – a message in the protocol without data. This includes acknowledgement messages. As these messages should be of fixed size, the time taken to communicate one should be fixed.
- *ADDR* – the size of a channel location address structure. These structures are used to permit the output end of a channel to connect to a corresponding input end. *ADDR* may vary based on implementation, but not enough to be considered unfixed.
- *MESSAGE* – a message sent in a communication from one domain member to another. The size of *MESSAGE* is variable.

To represent mobility, M_n is used. The parameter n is the number of movement operations that have occurred from initial setup – M_0 representing a channel end that has not migrated.

There are four properties that are of interest. These are *Transmission time, Reconfiguration time, Reachability* and *Strength*. When defining an equation that

has an optional value based on circumstance, the optional value will be enclosed within square brackets [ ].

7.3.1 Transmission Time

Transmission time is the time taken for a sent data message to arrive at its destination. The time taken to transfer a message of a particular type (*PROTOCOL, ADDR* or *MESSAGE*) can be expressed using the function *t* and is based on the amount of data sent in the message. For discussion purposes, for a single communication between two members of any two domains, *t* is not affected by the actual distance up or down the domain tree travelled.

- *One-to-One networked channel* – transmission time in this model is the normal communication time between two domain members. Therefore, for any *n*, transmission time for $M_n = t(MESSAGE) + t(PROTOCOL)$.
- *Centralised server* – the connection between an input end and an output end is always direct. The only exception is when the input end has moved, leading to a message that must be resent, a message to indicate that the channel end has moved and a query for the new address from the server. Thus, transmission time for $M_n = t(MESSAGE) + t(PROTOCOL) [+ t(MESSAGE) + 2 \cdot t(PROTOCOL) + t(ADDR)]$.
- *Message box* – a message is transferred twice – once to the message box and once to the input channel end proper. The requesting message contains the current input channel end location. Prior to the first move of the channel, the request and subsequent send is local, as the input end is co-located with the message box. Thus, transmission time for $M_0 = t(MESSAGE) + t(PROTOCOL)$ and for $M_{n>0} = 2 \cdot t(MESSAGE) + t(ADDR) + t(PROTOCOL)$.
- *Message box server* – has the same transmission overhead as the message box, although the message box is always remote to the input end, thus there is no initial direct communication. Transmission time for $M_n = 2 \cdot t(MESSAGE) + t(ADDR) + t(PROTOCOL)$.
- *Chain* – sent messages must travel the entire length of the chain. As the length of the chain increases with each migration, transmission time also

increases. Acknowledgement and other protocol messages must also travel the entire length of the chain. Therefore, transmission time for M_n = *n·t(MESSAGE) + n·t(PROTOCOL)*.

- *Reconfiguring chain* – the chain has the ability to shorten whenever possible, thus there are worst and best case scenarios for transmission time. For the worst case scenario, any message must travel the entire length of the chain, so transmission time is the same as chain – for M_n = *n·t(MESSAGE) + n·t(PROTOCOL)*. For the best case scenario, the chain may connect directly between two domain members, thus providing optimum transmission time – for M_n = *t(MESSAGE) + t(PROTOCOL)*.
- *Mobile IP* – transmission time is based on the number of foreign agents to reach the destination. This is based on the number of nodes up and down a sub-tree that are traversed by the message. These values are represented by *up* and *down* respectively. Transmission time for M_n = *up·down·t(MESSAGE) + up·down·t(PROTOCOL).*

7.3.2 Reconfiguration Time

Reconfiguration time is the time taken to reconfigure the communication architecture to permit the new communication path created by the migration of a channel. Reconfiguration complexity is represented by a function, *r*. *r* takes three possible values: *EASY* for an architecture requiring little reconfiguration to allow two mobile channel ends to connect; *MODERATE* for an architecture that takes some extra functionality and link creation; and *HARD* for an architecture that requires a great deal of reconfiguration to allow mobility. The time represented by *r* will generally be small in comparison to the time taken to transfer messages between Nodes to allow reconfiguration. Message transfer time is taken into consideration for message transfer and acknowledgement. Channel transfer time for all models is either a protocol message or an address message. Further details are provided in Appendix G.

- *One-to-One networked channel* – reconfiguration of the underlying architecture involves changing the channel state and informing the

complement channel end of the new address. Messages sent involve the new address and possibly any message waiting on an input channel when it migrates. Thus, reconfiguration time for $M_n = r(EASY) + 2 \cdot t(ADDRESS) + 2 \cdot t(PROTOCOL)$ *[+ t(MESSAGE)].*

- *Centralised server* – reconfiguration involves sending an acknowledged message informing the server that the input channel end is about to migrate, sending an acknowledged message to the server with the new address, and a protocol message from the output end to enquire on the new address, and the address sent back as a response. Therefore, reconfiguration time for $M_n = r(EASY) + 6 \cdot t(PROTOCOL) + 2 \cdot t(ADDR)$.
- *Message box* – as messages are always sent and requested from the same location, and reconfiguration is a matter of sending the address of the message box to the new location. Thus reconfiguration time for $M_n = r(EASY) + t(ADDR) + t(PROTOCOL)$.
- *Message box server* – reconfiguration time for the message box server is the same as message box mobility. For $M_n = r(EASY) + t(ADDR) + t(PROTOCOL)$.
- *Chain* – the chain is similar to the message box, and requires redirection of the receiving channel (the channel linking the `LinkRX` to the channel object in Figure 37 of Section 5.1) to point to the new outgoing `Link`. A migration message contains the previous location, and the acknowledgement message contains the new address. Thus, reconfiguration time for $M_n = r(EASY) + 2 \cdot t(ADDR)$.
- *Reconfiguring chain* – there are best and worst case scenarios for reconfiguring the chain. When migrating, the channel end must take every previous location of the channel end and on arrival iterate through the list, checking connectivity to these previous locations. Therefore, the channel end must take at least one previous location, and may in fact take all previous locations. The best case scenario for $M_n = r(EASY) + 2 \cdot t(ADDR)$ and the worst case scenario is $M_n = r(HARD) + (n + 1) \cdot t(ADDR)$.
- *Mobile IP* – reconfiguration is based on how quickly the communication path through the foreign agents can be created. A mobile channel is sent via an existing channel, thus the backbone links between the foreign agents must

already exist. Reconfiguration therefore involves the foreign agent examining the migration message and determining where the registered channel must be redirected. The number of domains travelled is a consideration – these values can be represented by *up* and *down* respectively. Each migration message contains two addresses, and has a complement *ARRIVED* message with two addresses in most cases. The migration message must also be acknowledged through the communication path back to the sender of the migrating end. Thus, reconfiguration time for $M_n = r(MODERATE) + 2 \cdot up \cdot down \cdot 2 \cdot t(ADDR) + up \cdot down \cdot t(ADDR)$.

7.3.3 Reachability

Reachability is the set of domains within which a channel output end can successfully communicate to an input end using the specified mobility model. There are three sets of interest:

- *DOMAIN* – the domain in which the input end of the channel is located and all the sub-domains of this domain.
- *BRANCH* – the set of domains within the same branch as the input end, implying both up and down traversal of the domain tree.
- *GLOBAL* – is the set of all domains.

As it is possible for a node within a domain to connect up the tree, any model that allows such a connection is deemed to permit an output channel end that has migrated using such an existing connection to be connected to an input channel end down the tree via this connection. This is a generalisation. If the input or output ends were to move further, then the link would be broken in many cases.

- *One-to-One networked channel* – ports are sent via an existing connection to a new node, thus the new host is reachable from the existing one. Therefore, the first interaction allows connection into a sub-domain from the parent domain if the migrated port and its complement are on the same node. After this migration, then this is no longer the case as the port may have moved to a node not addressable from the new location. This can be

overcome by the sender of the port determining which node could connect to the other. Therefore, reachability is given as *BRANCH*.

- *Centralised server* – the server is used to maintain channel locations, therefore only domain members can connect directly to the server. Thus, normal reachability is *DOMAIN*. However, this would imply that two distinct sub-domains of the server's domain could communicate via a mobile channel, which is not the case. Thus, reachability is actually *DOMAIN* ∩ *BRANCH*.
- *Message box* – any node that can connect to the host of the mailbox can form an end of the mobile channel. Therefore, reachability can be initially thought of as *DOMAIN*. However, as the sender of a channel end may connect up the branch of the domain tree, it is possible that the host of the message box be told likewise to connect up the tree. This gives reachability of *DOMAIN* ∪ *BRANCH*.
- *Message box server* – as a server is being used, it must be possible for any receiver of a mobile channel end to be able to connect to the server. Unlike the centralised server approach, channel ends in two distinct sub-domains may communicate as the message is sent and retrieved from the server. Thus reachability is *DOMAIN*.
- *Chain* – as every location which the channel visits leaves a forwarding address, anywhere the channel migrates can be reached from the previous location. As a connection between any sub-domain and its parent is possible, the chain can effectively stretch anywhere through the tree. Reachability is therefore *GLOBAL*.
- *Reconfiguring chain* – as the chain only reconfigures itself based on connectivity to previous locations, the reachability of the reconfiguring chain is the same as that of the chain. Reachability is thus *GLOBAL*.
- *Mobile IP* – as messages are passed between domains via the agents within each domain, a channel end can effectively move anywhere. The path is created dynamically as a channel end is migrated. Thus, reachability is *GLOBAL*.

7.3.4 Strength

The strength of the mobility model relates to the robustness of the connection between the input and the output end. Robustness includes reliance on external elements; thus a server type system is considered to be relatively robust in comparison to an individual node. This is due to the possibility of multiple servers being used and servers being dedicated to specific tasks, as opposed to a single node which may terminate when computation is complete.

For strength there are three values:

- *WEAK* – a connection relying on a number of external entities.
- *MODERATE* – a connection relying on some external entities.
- *STRONG* – a direct connection between two nodes, requiring no external entities. All direct connections between two nodes within the same domain are considered *STRONG*.

The strengths of the different mobility models are:

- *One-to-One networked channel* – a port and its complement are always directly connected. Thus, the strength of the model is *STRONG*.
- *Centralised server* – as the connection between input and output ends is direct, the channel strength can be considered *STRONG* in most circumstances. The reliance on an external server does reduce the strength slightly. Strength is therefore *MODERATE* to *STRONG*.
- *Message box* – each channel requires that the original declarer of the input end remains operational and connected until the channel is no longer required. This does not lend itself well to standard distributed systems architectures as a node may disconnect when it has finished its own operations. The channel itself only requires two inter-node links, and is therefore reasonably strong in that respect. The strength of this model is therefore *MODERATE*.

- *Message box server* – the usage of a server removes the requirement of the node creating the input end remaining operational. Thus, strength is the same as the centralised server model – *MODERATE* to *STRONG*.
- *Chain* – the chain relies on every previous channel end location remaining active during the lifetime of the system. This is a serious weakness, as any one of these domain members may fail or disconnect for reasons outside the control of the nodes containing the channel ends. Strength is therefore *WEAK*.
- *Reconfiguring chain* – as the chain can potentially be shortened to the point where the output and input ends of the channel are directly connected; there is the potential for this model to be *STRONG*. Conversely, there is the potential that all previous locations are required for the chain to deliver messages. Strength is therefore *WEAK* to *STRONG*.
- *Mobile IP* – a reliance on domain agents routing messages to the correct location does mean that the inter-domain connections must remain operational. However, as these agents are effectively servers, dedicated to routing and reconfiguration, the strength of the model can be considered *MODERATE*.

7.4 Summary of Model Properties

Table 9 summarises the different mobile channel models by placing them in order from best to worst under the respective property headings.

Table 9: Summary of Mobile Channel Models

Transmission Time	Reconfiguration Time	Reachability	Strength
One-to-One networked channel	One-to-One networked channel	Chain	One-to-One networked channel
Centralised server	Message box server	Reconfiguring chain	Centralised server
Message box	Message box	Mobile IP	Message box server
Message box server	Chain	Message box	Mobile IP
Reconfiguring chain	Centralised server	Message box server	Message box
Mobile IP	Mobile IP	One-to-One networked channel	Reconfiguring chain
Chain	Reconfiguring chain	Centralised server	Chain

For transmission time, the one-to-one networked channel model provides the best scenario, followed closely by the centralised server model which is also normally directly connected. The two message box models allow transmission time that is fixed at twice the normal transmission time; the normal message box having an initial interaction advantage. The reconfiguring chain has the potential of directly connected channel ends, but may in fact have a greater transmission time if the chain cannot be reconfigured. The Mobile IP model also has the potential of direct connections, but may involve transmission via a number of domains. The reconfiguring chain model allows domains to be jumped if a direct connection up or down can be created, and thus the Mobile IP model is considered to have greater transmission time due to the number of intermediate domain agents that must be passed through. Finally, the chain model increases transmission time with each migration, with no potential for reconfiguration.

For reconfiguration time the one-to-one model provides the best case, followed by the two message box approaches which only require address transmission for migration. The chain requires an address for transmission, although a reconfiguration message to the previous location is required. As the centralised server model does not permit easy input end migration without the output end requiring reconfiguration, this model comes next. The Mobile IP model requires reconfiguring at multiple domain agents, whereas the reconfiguring chain attempts to shorten the chain by linking to the furthest location back in the chain possible.

For reachability, only three models allow a channel end to potentially move anywhere and remain connected to its complement. The chain models require no server to achieve this, and are therefore given a better reachability. The Mobile IP model requires the domain to have an agent to permit mobility. As the message box approaches allow a channel end and its complement to be in two separate sub-domains, these models come next. The one-to-one model potentially allows connection the entire length of a branch. The centralised server only allows two channel ends to be within the same branch below the server.

The directly connected models, one-to-one networked channel and centralised server, provide the strongest connection. The message box server, with the server managing the message box, provides the next strongest connection. As the Mobile IP model utilises server agents, it provides a fairly robust channel structure. The standard message box's reliance on nodes that may disconnect comes next. As the reconfiguring chain may rely on some external nodes, it is stronger than the chain model which gets weaker with every movement.

7.5 Conclusions

Examining these properties, it can be seen that the one-to-one networked channel model has the best transmission time, reconfiguration time and strength, although it does fair badly for reachability. The main drawback for the one-to-one model is the removal of the Any-2-One communication architecture present in standard networked channels. This problem is not an issue for a Ubiquitous Computing per se, but the π-Calculus does permit this form of name sharing. If the π-Calculus is seen as a formal mobility model for modelling Ubiquitous Computing architectures, then having a shared channel end is advantageous. The reachability problem is of more concern, as it means that channel ends cannot migrate too far from their complement. A further consideration is how the one-to-one architecture is enforced. This can be done by adding registration and deregistration messages to the protocol, and adding channel states for a channel that is registered (and thus only accepts messages from the correct output channel end).

The centralised server has low transmission time and high strength, although reachability is an issue. Reconfiguration time is poor compared to the majority of other models, although the difference between this approach and the one-to-one model is not great. This model is well suited for controlled environments such as cluster computing, which is where the pony framework [120, 130] is aimed. For Ubiquitous Computing however, it does not provide the potential reachability that may be required.

The message box approaches have reasonable, and predictable, transmission time and reconfiguration time. Strength is good, although the normal message box has

weaknesses. Reachability is better than the server and one-to-one channel models due to the two step transmission process, thus the message box offers greater potential. Reachability is still not global, and therefore certain models of interaction are not possible.

The chain based approaches provide global reachability, but do so at the detriment of transmission time and strength. Reconfiguration time is good however. Although the reachability permits the interactions that may be required in Ubiquitous Computing, the increased transmission time and weakness nullifies this advantage, leading to channels that are not suitable for systems requiring service guarantees. Ubiquitous Computing does have the constraint of stability placed upon it. Potentially, the reconfiguring chain provides a model that may be suitable for Ubiquitous Computing connection mobility.

The Mobile IP model provides global mobility within domains that have agents controlling channels. Transmission time may be slow, but it is more predictable than the chain based approaches, and potentially allows direct connections. Reconfiguration time is poor, although it is significantly better than the reconfiguring chain model. This model is also stronger than the chain based models. Therefore, the Mobile IP model provides a good model for connection mobility in many scenarios, including Ubiquitous Computing.

Therefore, there are two models that appear to provide the mobility required to support truly dynamic architectures within a global architecture. However, this is assuming that channel ends require this level of migration within Ubiquitous Computing. As Chapter 2 described, the idea of the global Ubiquitous Computer is possibly incorrect, and individually controlled ubiquitous domains may be a more suitable approach. Therefore, the server based approaches may be more suitable due to the control they provide.

What is apparent from these different models is that mobility may not be possible at the protocol level, due to the different requirements for different application contexts. For example, the cluster computing scenarios that pony is aimed at require a model that has good transmission time, and strong connections. This

server approach may not suit Ubiquitous Computing. Any protocol implemented across the various CSP based network environments must be general enough to provide the application context required for different scenarios. Picking one of the models described does not provide this. A focus could be placed on Ubiquitous Computing only however.

A potential solution is to adopt both the centralised server and Mobile IP model. As a node must resolve a new location if the input end moves in both models, the potential of either a domain agent forwarding the message or a server that merely provides a new address does not change how the node acts after the complement end of a channel has migrated. Although this may permit many scenarios, further research is required to discover if it satisfies them all.

Another consideration not discussed is the handover between local and networked channels that is caused by migration. If a networked channel and a local JCSP channel are to behave similarly, then it should be possible to send a local channel end down a networked channel, and for the local channel to become networked. The main difference between a local channel and a networked channel is that a networked channel has a location, and this must transparently be created and the required network infrastructure put in place to handle the new networked channel.

In summary, there are models of channel mobility that are suitable for Ubiquitous Computing but which are not suitable for other applications. Therefore, building a mobility model directly into the protocol and architecture is only reasonable within individual application contexts. This is a limitation to the different possible scenarios even Ubiquitous Computing promotes. Any framework with which JCSP Networking interacts with must also adopt the same channel mobility model if used in a Ubiquitous Computing scenario.

A problem also exists with channels that are sent as part of another data structure, as any protocol will have to take into account that a channel is sent with other data. The most probable candidate for this operation is a mobile process. The following chapter discusses potential process mobility, and notes why this is far more difficult to achieve between different platforms.

Chapter 8 Process Mobility

In this chapter, a discussion of how process mobility in JCSP Networking can be achieved is presented. Channel mobility models have been presented in the previous chapter, with potential models of channel mobility that suit Ubiquitous Computing scenarios highlighted. Consideration of how distributed mobility can be achieved allows a discussion on how suitable JCSP Networking is for the dynamic architectures of Ubiquitous Computing. Process mobility is enabled by channel mobility, although the migration of an actively running component is considered difficult. In this chapter, an approach to enable process mobility is discussed. Section 8.1 introduces process mobility in more detail, and Section 8.2 reviews other attempts at active component mobility. Section 8.3 discusses a technique to enable strong process mobility, and Section 8.4 illustrates a practical implementation of the approach. Finally Section 8.5 summarises the technique developed.

8.1 Introduction

Chapter 2 provided an abstract definition of process mobility. This was:

> *Process mobility is the ability to change the location of an actively running process.*

The key concept is “actively running”. Previous work on JCSP mobility [17] has focussed on single stopped processes, providing code mobility mechanisms necessary to move a process transparently from one system to another. Code mobility is not difficult in a framework such as Java, however work previously presented within this thesis has argued against reliance on Java as a platform. This negates the code mobility argument. Currently it is not possible to define a process for one framework and send the code for execution in another without some form

of virtualisation technology, or relying on typed processes and no mobility of code. Therefore, code mobility will not form any further discussion on process mobility presented here.

8.1.1 Defining a Mobile Process

Removing code mobility from strong mobility modifies what a strongly mobile component comprises. From the argument presented thus far, there is also the consideration of channel or connection mobility. Finally, the removal of code highlights that a mobile component can be partially defined by its type. Thus it is possible to redefine strong component mobility when considering process mobility:

- *Type* – the type of the process, defining its structure and behaviour.
- *State* – the state of the mobile component. This comprises of three parts:
    - *Connections* – the inter-component connections that are contained within the mobile component.
    - *Data* – the variables that are contained within the component. This also includes any sub-components.
    - *Behaviour* – the current execution state of the component.

Code can be considered as part of the type information if this is not known at the receiving Node of a mobile component, although the receiving Node will require some knowledge of the component in an abstract manner.

Connections form part of the state and due to channel mobility can also be considered variable. Thus, although initially a host process will know all external connections, it must be the case that all sub-components take their own connections with them. This is due to the dynamic nature of the connections within a component.

How the process executes can also define how the process can be viewed. If a process is migrated to a new location, and then executed in sequence with the new host process, the mobile process can be considered as a mobile service [108]. This is because the process has added functionality to the actively running host process. If the mobile process is to run in parallel with the host process, then this process

can be considered a mobile agent. This view is due to the idea of an agent performing a task on behalf of another component, and thus executing outside the normal running of another component.

The definition of strong mobility might also not be what we wish to achieved. Although strong mobility originally referred to a component which took its execution state with it, the current direction is a component that can move at any point in its execution and take its execution state with it. In this chapter, the latter definition is approached, as achieving this goal permits achievement of the former.

8.1.2 Transferring a Process

The ability to transfer an actively running process has been discussed previously in [17]. In particular the argument was made that complex process mobility is difficult. Complex process mobility involves the suspension of a network of interacting processes, the transfer of said process network, and the resumption of the process network at the receiving location at the same execution state that the network was suspended at. The problem with suspending a process network has been evident in process oriented architectures for some time (for example [138]). Complex process migration can also be related to strong mobility of code / agents [76], which is the mobility of execution and data state within a mobile component.

Process networks can be viewed in a tree structure. The initial process has a number of child processes, which have child processes, etc. Figure 69 illustrates a process tree view of a process network, and illustrates how process mobility is viewed in such a context.

Figure 69 shows the migration of process *T* from the context of *Q* to the context of *R*. As indicated, it is actually migration of a branch of the tree that is occurring, with the connection from *Q* to *T* migrated to form the connection from *R* to *T*.

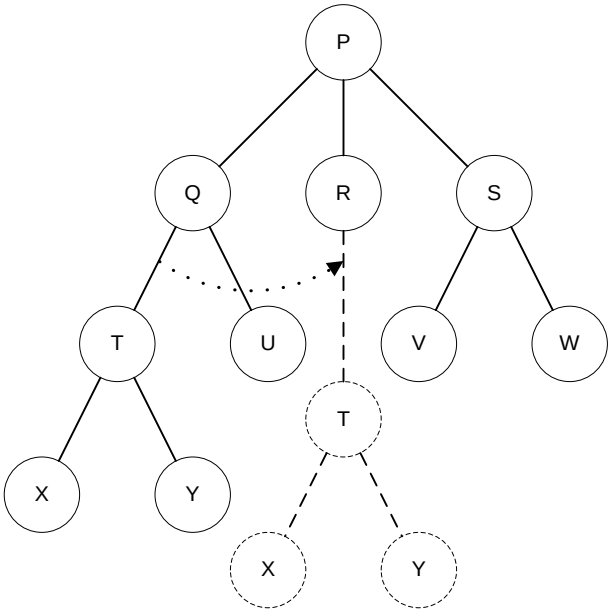


Figure 69: Process Branch Mobility

Examining process mobility from this view illustrates where a migration signal must come from, which is the connection between *Q* and *T*. Thus, either *Q* can transfer *T* to *R*, or if self referential processes are allowed, *T* can copy itself to R. The latter case does raise the question of whether the original copy should remain active. For the argument presented in this chapter, it is considered that strict mobility and not copying is in effect for processes.

The ability to move an entire branch of a process network to a new location is considered complex process mobility, whereas the ability to move a single leaf process is considered simple process mobility. The latter can be achieved by supplying a signal to the process to suspend. The former requires a mechanism that ensures that externally the process behaves as expected, but also appears willing to migrate at any point. If this idea is imposed within the migratory process, then it can be seen that each individual process must also appear to be willing to migrate at any point.

This chapter presents a method for allowing complex process mobility by capturing the behaviour of currently running processes. The methodology is by no means complete and verified, and some problems are highlighted. Some of these problems are related to current methods used to achieve primitives in architectures such as JCSP, and a proof of this shall be given. First, some information on other

approaches used to achieve strong mobility and process network suspension is presented.

8.2 Related Work

Picco [108] has defined strong mobility as the execution state of the mobile component being transferred transparently without specific coding to handle the mobility. Unfortunately, many mobility systems utilise Java, and thus do not offer this capability due to the inability to capture thread state in Java. Other approaches are used to attempt to artificially capture the execution state at a fixed point of execution, with the possibility of rolling back execution to the previously stored fixed point if migration occurs between capture points. This technique is referred to as checkpointing [147].

8.2.1 Java Based Approaches

Howell [148] has used checkpointing to capture execution state of programs by capturing the state of an entire JVM. This involves a modified Java Runtime Environment (JRE) but no actual modification to code or compiler. Although thread state is captured, it is the entire JVM that is checkpointed and not individual threads. The approach is also not portable as it requires a modified JRE. Although inter-framework mobility is negated by the lack of code mobility, an approach that can be replicated between platforms is better for Ubiquitous Computing.

Truyen [149] captured individual thread state by manipulating bytecode to insert code blocks to capture and resume execution state. By doing this, and abstracting Java `Threads` into tasks and creating their own scheduler, Truyen successfully captured thread behaviour without manipulating the JVM or JRE. The approach works on individual threads, not numerous interacting ones. Work was expanded [150] to accommodate remote object systems, where multiple threads are coordinating via Java RMI. Although interesting from a connection mobility point of view, no work on threads with internal threads was undertaken. The problem solved is particular to distributed object systems, where there is no encapsulated ownership of individual remote objects.

Zhu [151] developed a method that modified Java's Just-In-Time (JIT) compilation to capture individual thread states transparently. This approach is complicated due to some Java bytecode instructions having no direct correlation on the native machine. The method is also restrictive as it is Java specific, only captures individual thread state, and requires a modified JRE to operate.

Bouchenak [152-154] has proposed a solution that requires modification of the JVM, one of the main goals being elimination of overhead incurred by other approaches to thread migration. Bouchenak's approach gathered type information within the JVM for correct and complete reinterpretation of the thread at its destination. This approach is restricted by reliance on a modified JVM and allows migration of single threads only.

Sakamoto [155] applied a technique that used bytecode transformation to modify method calls to throw exceptions that would emit the execution state of a method. Points within method bodies are marked as possibly migratory and the resultant exception added to the surrounding method. The approach is interesting as it could be manipulated to mark methods as migration guarded in the same manner that methods can be guarded against multiple thread access. There is an overhead incurred, and the authors note limitations to their approach. There is also the limitation of single thread migration, and reliance on threads entering marked methods to allow the migration.

Ma [156] has provided strong process migration within Java-MPI (Message Passing Interface) using the Java debugging interface. No modified JVM or bytecode manipulation is required, and there is little overhead. A migration layer within the MPI framework is utilised to achieve strong migration. However, features of Java are still required and there is no capturing of multiple thread state.

Java mobile agent systems also attempt to capture execution state for transferral. The D'Agents framework [111] provides strong migration of threads, but does not allow multi-threaded migration – the authors noting that it is unclear whether this should be a necessity. As the term agent is itself ambiguous this is understandable.

The NOMADS system [157] provides strong mobility of multiple threads by allowing migration of Java thread groups, which allows migration of multiple mobile agents in a group. NOMADS executes within a Java-compatible virtual machine.

Java has a problem when considering thread migration. There is the problem that thread state is not explicitly exposed to the user, thus negating any simple method to allow thread migration. There is no concept of thread ownership, and thus it becomes difficult to decide whether a single thread or multiple threads should be migrated. This is not just a problem for threads, but for passive objects also. Java only provides weak encapsulation, thus an object may be owned by more than one thread. No consideration of object ownership has been taken in the above approaches except when involving Java RMI [150], the only solution that appears to consider connection mobility. For CSP / **occam** based approaches, strict encapsulation and boundaries are in place, with connectivity controlled via well defined channel interfaces. If adhered to, this removes many limitations Java only approaches face.

Serialization does provide mechanisms to enable transfer of object references by allowing aliasing within the object stream. However, the mechanisms do not support the type of migration required for mobility in this context. If an object is migrated as part of another object graph, and then modified at the original location and subsequently transferred, updates to the data state are lost in transfer due to the aliasing within the object stream. The lack of ownership of an object causes this to be a significant problem when considering both data and behavioural objects.

8.2.2 Generic Approaches

Fortino [158] has proposed mobile agent design using statecharts. A mobile agent retains its current historical state when migrated, and reintroduces this state on arrival. During execution, the mobile agent explicitly changes its execution state between state points. These state points can be considered checkpoints. Only at a checkpoint can a mobile agent choose to migrate. This is enforced by the agent interacting using events instead of standard methods. The weakness of this

approach is that the state points have to be created and stored to allow transfer. The authors also note that this approach is only suitable for single threaded agents.

Bettini [78] proposes making a procedure strongly mobile via the introduction of mark points, mark points being similar to checkpoints or state points. This approach is platform independent, modifying the design of a procedure as opposed to a specific implementation. The method is limited in that it does not consider multiple internally interacting components, and the method produces significantly more code for choice and iteration primitives.

Phillips [102] has developed a mobile ambient implementation within Java, although the technique to achieve mobility of agents is transferable. Between computations and communications, an agent checks if it has been called to move, and after migration the agent continues execution at this point. The technique is similar to introducing checkpoints, and takes into account child agents with a parent agent requesting that child agents migrate also. The technique is based on the asynchronous π-Calculus, and therefore does not consider committed events in a synchronous architecture. Recent work in this area [82] no longer discusses this approach, so it is unclear whether it has been expanded upon.

Generic approaches provide more insight into how a method to capture process network state can be developed. This is due to the view beyond threads, and Java in particular. The ability to place points within code at which processes must check whether or not they should migrate is of most interest, but consideration of CSP semantics must be taken into consideration. For further insight, approaches specific to CSP inspired platforms are examined.

8.2.3 CSP Based Approaches

Stopping a process network is not a new problem [138]. Welch discussed different approaches to terminate a process network and in particular how not to do it. Resetting is seen as practically what is required as opposed to stopping a process network, and resetting is more related to the capturing of state for migration. Resetting of a process network involves sending a reset signal through the network,

which each process receives and thus places itself into the reset state. This approach was prior to new additions to JCSP [137], which enables another solution.

Sputh [139] has criticised Welch's approach. The criticisms are somewhat JCSP specific, but computational processing and increased complexity are cited as problems, and also the handling of shared channel ends. Another problem overcome by Sputh's work not specifically highlighted is the black holing of incoming messages to prevent deadlock. Sputh overcomes this problem by allowing reset signals to travel both backwards and forwards through a process network. The reason that this is a problem is that black holing a message implies that the message is lost, which does not effectively capture the current execution state. Within an entire system this is not a problem, but a mobile process will only form part of a system and thus messages entering the mobile may be lost.

Sputh mentioned some problems with trying to reset a sub-network of processes using the JCSP-Poison technique. As a mobile process will be a sub-network this problem is imposed on process mobility also. Sputh has mentioned the problem of having two resetting process networks connected together via a channel, but the two process networks themselves having the possibility of being terminated independently. The problem scales, and resetting and terminating *n* process sub-networks is a problem if each may be terminated individually.

Welch [12] has expanded resetting to incorporate the suspension of mobile processes. This approach is the most complete solution thus far, but only suggests how suspension can be achieved with examples. A process must handle an incoming suspension signal externally from the process network, although only at certain points in execution. Strong mobility requires migration at any point during execution. Agent mobility implies externally and internally activated migration, thus relying solely on external signals may be a problem. Barnes does use a technique that retains current execution state via a state variable.

When considering mobility, using poison is not a suitable approach in all cases due to the different problem poison was meant to overcome. Poison is invariably injected into a process network from one of the leaves of the process tree (e.g. *X* in

Figure 67, page 140). It has been argued that a migration signal must come from the parent process or from the process itself, and travel down into all sub-processes. Poison flows through a system via channels, and sub-processes might not be connected, implying multiple poison signals entering the process network. Thus, poison does not enable complex process mobility in all circumstances.

Specific approaches within CSP based frameworks have been used to attempt to reset a network of processes. However, none of these approaches is sufficiently generic enough for the problem of capturing the current behavioural state of an actively running process network at any point. The solution presented in the following section attempts to overcome this problem.

8.3 Observably Strongly Mobile Processes

The approach proposed builds on the idea of strong mobility, checkpointing, state capture and processes having the choice to migrate at certain points. It builds upon the newer ideas presented by Welch [12], but does not use the graceful resetting technique originally proposed by Welch [138]. It exploits recent additions to JCSP [137], in particular the multi-way synchronisation capability provided by the `AltingBarrier`. We are not going to discuss how channel mobility is modelled here. To help illustrate, a small subset of CSP notation shall be used:

$$PROCESS := A, B, C, \ldots$$
$$EVENT := a, b, c, \ldots$$
$$Prefix := a \rightarrow P$$
$$Choice := (a \rightarrow P) \mid (b \rightarrow Q)$$
$$Parallel := A \parallel B$$
$$Sequential := A ; B$$
$$Renaming := P [\![a/b]\!]$$
$$Hiding := (a \rightarrow P) \setminus \{a\}$$

Processes are declared in upper case, and events in lower case. Prefix defines a new process from an event and process definition. For example, *Prefix* above means synchronise on a then behave as *P.* Specific input and output events (channel operations) are not defined as they are of no consequence as all events

must be usable as guards. Choice allows a process to choose between two possible guarded communications (e.g. *a* or *b*), and choice affects process behaviour depending on the chosen event. This is a generalised guarded alternative; more specific choices are used but are not examined here. A migration may be caused internally due to the process causing the move, or externally from the parent process. The Parallel operator yields a process in which its operand processes operate in parallel and which does not terminate until both its operand processes terminate. Parallel is normally defined with the events used to synchronise the processes, but this is of no consequence for the discussion presented. It is assumed that processes will only synchronise on shared events. Sequential means that once one process has finished the next process should be performed. Renaming allows an event name to be changed within a Process. For example, the above Renaming operation replaces event *b* with *a* in process *P*. Finally, Hiding is used to hide an event from being externally visible. For example, the above Hiding operation states that the event *a* is not observable outside of the defined process, and therefore externally the process behaves as *P*.

8.3.1 Simple Process Migration

Consider the definitions given for process mobility and strong mobility:

- Process mobility is the ability to change the location of an actively running process.
- Strong mobility is the ability to migrate a mobile component at any given point in its execution.

The goal is to allow processes to migrate at any point in their execution to another location and resume execution at the point of migration. To achieve this, a process is offered the choice to migrate at any point. Consider process *P* defined as:

$$P := a \rightarrow b \rightarrow P$$

P synchronises on *a*, then *b*, and then behaves as *P* (*a* then *b* then *P*). To offer migration, a new event is introduced called *migrate*. This event must be possible at any point in execution. Thus, *P* becomes:

$$P_{mobile} := P_a$$
$$P_a := (a \rightarrow P_b) \mid (migrate \rightarrow SKIP)$$
$$P_b := (b \rightarrow P_a) \mid (migrate \rightarrow SKIP)$$

P_{mobile} emits the same behaviour as *P*, but also has the opportunity to migrate. *SKIP* indicates successful completion of the process. This example does not outline how *migrate* is fired. *migrate* will normally occur from outside the process or process network, but as shall be shown is particular to where the migration attempt occurs and what exactly is to be migrated.

The problem is to retain the execution state of P_{mobile} after migration. As the process has been split into two separate process definitions, it becomes possible to start the process at any one of these definitions. What must occur is that the current execution state of the process must be stored or emitted somehow. This depends on the implementation platform (for example, Java would retain it as an internal attribute to the object), so specific details are left.

This shows how a simple process can be given the option to migrate, but this in itself is not new. All that is occurring here is that an option of a migration signal is given to the mobile. To expand this, consider two interacting processes:

$$P := a \rightarrow b \rightarrow P$$
$$Q := b \rightarrow c \rightarrow Q$$
$$R := P \parallel Q$$

Introducing mobility into these processes gives us the following:

$$P_{mobile} := P_a$$
$$P_a := (a \rightarrow P_b) \mid (migrate \rightarrow SKIP)$$
$$P_b := (b \rightarrow P_a) \mid (migrate \rightarrow SKIP)$$
$$Q_{mobile} := Q_b$$
$$Q_b := (b \rightarrow Q_c) \mid (migrate \rightarrow SKIP)$$
$$Q_c := (c \rightarrow Q_b) \mid (migrate \rightarrow SKIP)$$
$$R_{mobile} := (P_{mobile} \parallel Q_{mobile})$$

CSP enforces that an event can only be fired when all relevant processes agree to synchronise, thus for *migrate* to fire both *P* and *Q* must be willing to participate. For R_{mobile}, parallel execution of *P* and *Q* completes before the *SKIP* is reached. This can only occur when both *P* and *Q* have finished, thus *migrate* must have been fired bringing *P* and *Q* to a successful termination.

8.3.2 Parallelised Process Migration

A more complicated situation occurs when a process starts and then goes parallel before or after performing other interactions. This occurrence has been highlighted numerous times within the context of poison and resetting [138, 159]. The simple technique of synchronising on an event does not work in all circumstances, as there is no way of knowing whether or not the parallel completed successfully without migration or was paused due to migration. Consider two possibilities for internally parallel processes. A process may perform some events and then go parallel as the last operation, or the parallel may occur prior to other events. Both these eventualities cover any combination of events and parallelisation.

8.3.2.1 Processes Ending Parallelised

Consider the following process definition:

$$P := a \rightarrow b \rightarrow (Q \parallel R)$$

From the previous definition of creating mobile processes, *P* can be converted to the following:

$$P_{mobile} := P_a$$
$$P_a := (a \rightarrow P_b) \mid (migrate \rightarrow SKIP)$$
$$P_b := \left(b \rightarrow P_{par}\right) \mid (migrate \rightarrow SKIP)$$
$$P_{par} := (Q_{mobile} \parallel R_{mobile})$$

Q_{mobile} and R_{mobile} will synchronise on *migrate* if they are made mobile in the manner described, thus it becomes evident that P_{par} will only terminate when *migrate* occurs. This is all that is needed for a process that ends internally parallel.

8.3.2.2 Processes Beginning Parallelised

Processes that begin internally parallelised before performing other operations are more difficult. Consider the following:

$$P := (Q \parallel R) ; (a \rightarrow b \rightarrow P)$$
$$Q := c \rightarrow d \rightarrow SKIP$$
$$R := c \rightarrow e \rightarrow SKIP$$

P begins by performing both *Q* and *R* together, and then performing *a* then *b*. The problem faced is that (*Q* || *R*) can terminate due to *migrate* or normal operations. It must be possible to distinguish between successful termination of parallelised processes and migration termination of parallelised processes. The subtlety of the example presented is that *Q* may successfully terminate prior to *R* if *d* is executed first and vice versa if *e* is executed first. To check completion, the introduction of a further event, *finished*, is required to check successful completion of the parallelised processes. This event is not observable from outside *P* and is therefore hidden. By doing this, the following process definitions are generated:

$$Q_{mobile} := Q_c$$
$$Q_c := (c \rightarrow Q_d) \mid (migrate \rightarrow SKIP)$$
$$Q_d := (d \rightarrow Q_{finished}) \mid (migrate \rightarrow SKIP)$$
$$Q_{finished} := (finish \rightarrow SKIP) \mid (migrate \rightarrow SKIP)$$
$$P_{mobile} := P_{par}$$
$$P_{par} := ((Q_{mobile} \parallel R_{mobile}) \parallel ((migrate \rightarrow SKIP) \mid (finished \rightarrow P_a))) \setminus \{finished\}$$
$$P_a := (a \rightarrow P_b) \mid (migrate \rightarrow SKIP)$$
$$P_b := (b \rightarrow P_{par}) \mid (migrate \rightarrow SKIP)$$

The definition of R_{mobile} is similar to Q_{mobile} and not given. The rest of the definition of *P* is straightforward. To handle parallelisation, a separate process interacts with both *Q* and *R* via *migrate* and *finished*. Q_{mobile} operates as a standard mobile except when it reaches completion. At this point there are two options.

1. If R_{mobile} successfully completes, *finished* is selected, thus completing Q_{mobile}, R_{mobile} and the manager process within P_{mobile}, allowing P_{mobile} to continue.
2. *migrate* may occur, thus triggering the migration within all processes and not allowing P_{mobile} to continue to the next state point.

In either case, all the processes are terminated, and the relevant execution states either captured or continued.

8.3.3 Connected Mobiles

Another subtle problem with capturing the current behaviour of process networks is ensuring that any other connected process networks do not deadlock due to incorrect behaviour. Capturing the current state of the network, and the assumption that channels / events are also mobile overcomes some of the initial problems. However, if sub-process networks are independently mobile within a mobile process, more care must be taken. Consider the following:

$$P := a \rightarrow b \rightarrow P$$
$$Q := b \rightarrow c \rightarrow Q$$
$$R := P \parallel Q$$

If *P* and *Q* are independently mobile, then they cannot share the same *migrate* event as this will enforce the two processes to terminate. *migrate* can be renamed to overcome this.

$$R := P_{mobile}\ [\![migrate\_p/migrate]\!] \parallel Q_{mobile}\ [\![migrate\_q/migrate]\!]$$

This allows both *P* and *Q* to be independently mobile. If *R* must be mobile as well, then a further consideration must be taken into account where *R* receives a migration signal, subsequently signal that the sub-processes *P* and *Q* should terminate, and then signal that *R* has terminated. To do this, another process and the *finished* event are used again:

$$R_{mobile} := ((P_{mobile}\ [\![migrate\_p/migrate]\!] \parallel Q_{mobile}\ [\![migrate\_q/migrate]\!]) \parallel (migrate \rightarrow migrate\_p \rightarrow migrate\_q \rightarrow finished \rightarrow SKIP)) \setminus \{finished\}$$

As P_{mobile} and Q_{mobile} are independently mobile, they can be terminated in sequence within the new process. Once all processes have been terminated, the *finished* event is fired, thus signalling that R_{mobile} is ready for migration.

Giving each sub-process a unique *migrate* event and signalling each process in turn could also be used to shut down internally parallel processes. The reason not to do this is that it would involve an extra manger process for each sub-process network. Although shutting down processes in sequence is cautioned against [138], this was due to a lack of output guards and multi-way events being available. As this problem has recently been resolved [137], there is no longer the same concern.

8.3.4 Example – Numbers Process

The Numbers Process is used for the CommsTime benchmark [134], and consists of three processes: *PREFIX*, *SUCCESSOR*, and *DELTA2*. These processes are defined as:

$$PREFIX(x) := a!x \rightarrow IDENTITY$$
$$IDENTITY := b?x \rightarrow a!x \rightarrow IDENTITY$$
$$SUCCESSOR := c?x \rightarrow b!(x+1) \rightarrow SUCCESSOR$$
$$DELTA2 := a?x \rightarrow (c!x \rightarrow SKIP \parallel d!x \rightarrow SKIP)\ ;\ DELTA2$$
$$NUMBERS := (PREFIX(0) \parallel SUCESSOR \parallel DELTA2) \setminus \{a, b, c\}$$

Channel communication is defined using ! for output and ? for input. For this example, no consideration on how the states of the processes are retained is given, and it is assumed that when a process is restarted, the correct state is used. The *NUMBERS* process has channels *a*, *b* and c hidden, thus leaving *d* exposed as an external channel from *NUMBERS*, the others being internal. First consider *PREFIX*:

$$PREFIX_{mobile}\ (x) := PREFIX_{writing}\ (x)$$
$$PREFIX_{writing}\ (x) := (a!x \rightarrow PREFIX_{identity}\ ) \mid (migrate \rightarrow SKIP)$$
$$PREFIX_{identity} := IDENTITY_{mobile}$$

This allows termination of *PREFIX* and subsequent restarting at any point. If the internal *IDENTITY* process is to be restarted, it is assumed that the correct execution state of the internal process is chosen. *IDENTITY* itself is simple:

$$IDENTITY_{mobile} := IDENTITY_{reading}$$
$$IDENTITY_{reading} := (b?x \rightarrow IDENTITY_{writing}(x)) \mid (migrate \rightarrow SKIP)$$
$$IDENTITY_{writing}(x) := (a!x \rightarrow IDENTITY_{reading}) \mid (migrate \rightarrow SKIP)$$

It is assumed that when the *IDENTITY*$_{writing}$ process is terminated, it retains the last read value (*x*) for subsequent sending when the process is restarted. *SUCCESSOR*$_{mobile}$ is similar to *IDENTITY*$_{mobile}$ and is not provided.

DELTA2 requires more care due to the internal parallel. To ease the problem, a new process definition is introduced which is responsible for outputting a value on a channel:

$$WRITE(x) := out!x \rightarrow SKIP$$

DELTA2 is now redefined as:

$$DELTA2 := a?x \rightarrow (WRITE(x)[\![c/out]\!] \parallel WRITE(x)[\![d/out]\!]) ; DELTA2$$

Converting this into a mobile process requires a mobile version of the *WRITE* process, which must incorporate the *finished* and *migrate* events (Section 8.3.2.2). Thus:

$$WRITE_{mobile}(x) := WRITE_{writing}(x)$$
$$WRITE_{writing}(x) := (out!x \rightarrow WRITE_{finishing}) \mid (migrate \rightarrow SKIP)$$
$$WRITE_{finishing} := (finished \rightarrow SKIP) \mid (migrate \rightarrow SKIP)$$

Now *DELTA2*$_{mobile}$ can be defined:

$$DELTA2_{mobile} := DELTA2_{reading}$$
$$DELTA2_{reading} := (a?x \rightarrow DELTA2_{writing}(x)) \mid (migrate \rightarrow SKIP)$$
$$DELTA2_{writing}(x) := ((WRITE_{mobile}(x)[\![c/out]\!] \parallel WRITE_{mobile}(x)[\![d/out]\!])$$
$$\parallel ((migrate \rightarrow SKIP) \mid (finished \rightarrow DELTA2_{reading}))) \setminus \{finished\}$$

The definition of the *NUMBERS*$_{mobile}$ is now straight forward:

$$NUMBERS_{mobile} := (PREFIX_{mobile}(0) \parallel SUCCESSOR_{mobile} \parallel DELTA2_{mobile}) \setminus \{a, b, c\}$$

8.3.5 Limitations

The methodology described here is by no means complete, and there are limitations from the theoretical point of view. Limitations from an implementation point of view also exist, and shall be described presently. It first must be considered that this technique has been developed with practical implementation of JCSP process network mobility in mind, and no formal analysis has been undertaken to verify that the technique is correct in all circumstances.

The first limitation is the generalisation of choice. There are three choice types within CSP: external choice, non-deterministic choice and conditional choice. With this method, it is external choice that is the most likely to be considered, as *migrate* will likely be fired from outside the process. Non-deterministic choice of *migrate* implies that the process has itself decided to migrate. From a CSP point of view this is complicated as it implies that the external process is willing to migrate the mobile process. This may not be the case, and the external process may be performing other actions that do not consider migration. From an implementation point of view, especially in Java, a process may move itself as Java objects can reference themselves. Care would have to be taken to ensure that the inner process is terminated independently and moved without requiring the external process to interact with it.

Priority of choice has also not been taken into account. If migrate is offered at any point another event is offered, then the environment may not choose migrate over the other offered event. As the process must be willing to migrate at any point, no guaranteed selection of *migrate* is a problem. In implementation terms, priority of choice can be provided, although it is not always guaranteed [137]. This has some repercussions for implementation which shall be discussed shortly. *migrate* must always be possible when considering this approach to process state capture, and should not be arbitrary.

Interleaving has not been taken into account. Interleaving of processes means that the processes do not interact together. It can be considered as a parallel without

any shared events, and therefore interleaving could be converted into a parallel sharing the *migrate* event.

No consideration has been taken for how current execution and data state of a process is stored. This is implementation specific. The method described here could add such information emitted on a channel (*status*) after *migrate* and prior to successful termination. However, for a Java implementation, this status can be stored within the object.

No indication as to where *migrate* is fired from is given, and this relates back to the use of generalised choice. As processes can be considered strictly owned by a parent (or starting) process, it is the parent process or the process itself that has the ability to migrate the mobile. An ancestor of the parent process should not have access to the individual sub-processes of one of its children.

It has also been assumed that processes can be sent via channels, and that they can be successfully restarted within the context of the receiving process. None of these features are present in CSP, and therefore it is currently difficult to verify that this approach will work. Future work will hopefully lead to verification that the mobile version of a process emits the same behaviour as the non-mobile, and that the mobile process is also willing to offer *migrate* at any point in its execution.

8.4 Implementation

The method described in Section 8.3 takes an abstract view of process network mobility via state capture. In this section, an implementation of the *NUMBERS* process in JCSP shall be presented and modified to allow migration. An examination of specific features available within Java to aid the migration process and limitations due to the current implementation of JCSP are also presented.

8.4.1 NumbersInt Process in JCSP

The `NumbersInt` process in JCSP is an implementation of the *NUMBERS* process described in Section 8.3.4. Full code listings of this process and the mobile process version can be found in Appendix H. Here only necessary code segments are presented for discussion.

The JCSP implementation of `NumbersInt` is similar to the CSP definition, and has the same processes in operation: `PrefixInt`, `SuccessorInt`, and `Delta2Int`. There are also `IdentityInt` and `ProcessWriteInt` processes for necessary internal processes. The run methods (modified for clarity) of these processes are:

```
public class PrefixInt
{
  int prefix;
  ChannelInputInt b;
  ChannelOutputInt a;

  public void run()
  {
    a.write(prefix);
    new IdentityInt(a, b).run();
  }
}

public class IdentityInt
{
  ChannelInputInt b;
  ChannelOutputInt a;

  public void run()
  {
    while (true)
    {
      int x = b.read();
      a.write(x);
    }
  }
}

public class SuccessorInt
{
  ChannelInputInt c;
  ChannelOutputInt b;

  public void run()
  {
    while (true)
    {
      int x = c.read();
      b.write(x);
    }
  }
}

public class Delta2Int
{
  ChannelInputInt a;
  ChannelOutputInt c;
  ChannelOutputInt d;

  public void run()
  {
    ProcessWriteInt[] parWrite =
      {new ProcessWriteInt(c), new ProcessWriteInt(d)};
    Parallel par = new Parallel(parWrite);
```

```
    while (true)
    {
      int x = a.read();
      parWrite[0].value = x;
      parWrite[1].value = x;
      par.run();
  }
}

public class ProcessWriteInt
{
  ChannelOutputInt out;

  public void run()
  {
    out.write(value);
  }
}

public class Numbers
{
  ChannelOutputInt d;

  public void run()
  {
    One2OneChannelInt a = Channel.one2one();
    One2OneChannelInt b = Channel.one2one();
    One2OneChannelInt c = Channel.one2one();
    new Parallel(new CSProcess[]
    {
      new PrefixInt(0, b.in(), a.out()),
      new SuccessorInt(c.in(), b.out()),
      new Delta2Int(a.in(), c.out(), d)
    }).run();
  }
}
```

The `Delta2Int` process is defined in such a manner due to Java constraints, as the individual `ProcessWriteInt` processes are required to provide parallel output. The other approach would be to use inline code to represent the processes. The handler for the *finished* event is likewise implemented in this fashion in the mobile version of `Delta2Int` described in the next question.

8.4.2 MobileNumbersInt Process

Recent additions to JCSP [137] have added multi-way synchronisation via `AltingBarrier`, and guarded output (`AltingChannelOutputInt`) is offered with `One2OneChannelSymmetricInt`. This channel operates with an internal `AltingBarrier` to enable guarded output. Mobile `PrefixInt` is simple:

```
public class MobilePrefixInt
{
  int prefix;
  AltingChannelInputInt b;
  AltingChannelOutputInt a;
  AltingBarrier migrate;
  MobileIdentity identity;

  public void run()
  {
    Guard[] guards = {migrate, a};
    Alternative alt = new Alternative(guards);
    switch (state)
    {
      case WRITING:
      int selected = alt.priSelect();
      switch (selected)
      {
        case 0: break; // migrate
        case 1: // a
          a.write(prefix);
          state = IDENTITY;
          identity.run();
      }
      break;
      case IDENTITY:
        identity.run();
    }
  }
}
```

`state` holds the current execution state of the process, and is initially set to `WRITING`. If the first value (`prefix`) is successfully written, the state changes to `IDENTITY` and `IdentityInt` is executed. The `IdentityInt` process is just run, as this process will have been initialised within the constructor, or will have state based on the previous run of the process. Mobile `IdentityInt` is also trivial:

```
public class MobileIdentityInt
{
  AltingChannelInputInt b;
  AltingChannelOutputInt a;
  AltingBarrier migrate;

  public void run()
  {
    Guard[] guards = {migrate, b, a};
    Alternative alt = new Alternative(guards);
    boolean running = true;
    while (running)
    {
      switch (state)
      {
        case READING:
          boolean[] active = {true, true, false};
          int selected = alt.priSelect(active);
          switch (selected)
          {
            case 0: // migrate
              running = false;
              break;
            case 1: // b
              x = b.read();
              state = WRITING;
```

```
            }
            break;
          case WRITING:
            boolean[] active = {true, false, true};
            int selected = alt.priSelect(active);
            switch (selected)
            {
              case 0: // migrate
                running = false;
                break;
              case 2: // a
                a.write(x);
                state = READING;
            }
            break;
        }
    }
  }
}
```

The usage of flagged guards in this process is to allow simplicity of implementation, and two separate `Alternative` objects could have been used instead. This is a requirement enforced by Java / JCSP. `MobileSuccessorInt` is similar to `MobileIdentityInt` and its full code listing can be found in Appendix H.

Mobile `Delta2Int` is more complicated. First, consider the mobile version of `ProcessWriteInt`:

```
public class MobileProcessWriteInt
{
  AltingChannelOutputInt out;
  AltingBarrier migrate;
  AltingBarrier finished;

  public void run()
  {
    Guard[] guards = {migrate, out, finished};
    Alternative alt = new Alternative(guards);
    switch (state)
    {
      case WRITING:
        boolean[] active = {true, true, false};
        int selected = alt.priSelect(active);
        switch (selected)
        {
          case 0: break; // migrate
          case 1: // out
            out.write(value);
            state = FINISHING;
        }
        break;
      case FINISHING:
        boolean[] active = {true, false, true};
        int selected = alt.priSelect(active);
        switch (selected)
        {
          case 0: break; // migrate
          case 2: // finished
            state = WRITING;
        } // For reuse purposes
        break;
```

```
    }
  }
}
```

To capture that a process may be finished or not, a new process is defined:

```
public class CheckFinished
{
  AltingBarrier migrate;
  AltingBarrier finished;
  boolean isFinished;

  public void run()
  {
    Guard[] guards = {migrate, finished};
    Alternative alt = new Alternative(guards);
    isFinished = false;
    int selected = alt.priSelect();
    if (selected != 0)
      isFinished = true;
  }
}
```

The attribute `isFinished` is used to check if the process completed via the `migrate` event or the `finished` event. This value remains false unless `finished` is selected within the `Alternative`. With this process defined, `MobileDelta2Int` can now be defined:

```
public class MobileDelta2Int
{
  AltingChannelInputInt a;
  AltingChannelOutputInt c;
  AltingChannelOutputInt d;
  AltingBarrier migrate;
  MobileProcessWrite[] parWrite;
  CheckFinished checkFinished;
  AltingBarrier[] barrier = migrate.expand(2);

  public void run()
  {
    Guard[] guards = {migrate, a};
    Alternative alt = new Alternative(guards);
    CSProcess[] processes =
              {parWrite[0], parWrite[1], checkFinished};
    Parallel par = new Parallel(processes);
    boolean running = false;
    while (running)
    {
      switch (state)
      {
        case READING:
          int selected = alt.priSelect();
          switch (selected)
          {
            case 0:
              finished = false;
              break;
            case 1:
              x = a.read();
              state = WRITING;
          }
          break;
```

```
          case WRITING:
            parWrite[0].value = x;
            parWrite[1].value = x;
            barrier[0].enroll();
            barrier[1].enroll();
            par.run();
            if (!checkFinished.isFinished)
              running = false;
            else
            {
              state = READING;
              barrier[0].resign();
              barrier[1].resign();
            }
            break;
        }
      }
    }
}
```

The reason for starting the `ProcessWrite` and `CheckFinished` processes in parallel is to ensure that the underlying threads have finished before trying to restart the processes, which may not be the case in Java. Using a `ProcessManager` to spawn the `MobileProcessWrite` processes and allowing `MobileNumbersInt` to guard on migrate and finished can lead to exceptions caused by spawning too many threads. The process enrols and then resigns from the `AltingBarriers` of the `MobileProcessWrites` prior to activation and after termination. This is to ensure that the other processes can synchronise on migrate independently of the `MobileProcessWriteInt` being in operation. Whenever the process is started it must be enrolled on the `AltingBarrier`, and once it has successfully terminated, it must resign.

It is now possible to define a mobile `NumbersInt` process:

```
MobileNumbersInt(AltingChannelOutputInt d)
{
  AltingChannelOutputInt d;

  public void run()
  {
    AltingBarrier[] migrate = AltingBarrier.create(4);
    One2OneChannelSymmetricInt a = Channel.one2oneSymmetricInt();
    One2OneChannelSymmetricInt b = Channel.one2oneSymmetricInt();
    One2OneChannelSymmetricInt c = Channel.one2oneSymmetricInt();
    AltingBarrier innerMigrate = migrate[3];
    prefix = new MobilePrefixInt(0, b.out(), a.in(), migrate[0]);
    successor =
      new MobileSuccessorInt(c.in(), b.out(), migrate[1]);
    delta = new MobileDelta2Int(a.in(), c.out(), d, migrate[2]);
    new Parallel(new CSProcess[] {prefix,successor,delta}).run();
  }
}
```

The `innerMigrate AltingBarrier` is used to trigger the migration process externally, as shall be described in the following subsection.

8.4.3 Java Serialization to Help Migration

Customisation of the serialization process can be exploited to enable the suspension process. A class can be declared `Externalizable`, or specific methods overridden to customise the serialization behaviour. Whenever an instance of the class is written to or read from an object stream, these methods are called instead of the standard mechanism used. For example, the method called to serialize an instance of `MobileNumbersInt` is:

```
private void writeObject(ObjectOutputStream out) throws IOException
{
  innerMigrate.sync();
  out.writeObject(prefix);
  out.writeObject(successor);
  out.writeObject(delta);
}
```

The `innerMigrate` is waited upon by the writing process, thus it can be judged that the processes are in such a state that they can be written to the stream safely.

8.4.4 Implementation Limitations

There are implementation problems when considering this method within the context of JCSP. The first relates to certain assumptions made on the mobility of events and channels. As Chapter 7 described, distributed channel mobility is not a guaranteed feature, and still requires finalisation. General guarded events provided by the `AltingBarrier` also do not have a networked equivalent. Careful design may get round these problems.

The second problem comes from the lack of a networked `AltingBarrier`. As guarded output is currently implemented using an `AltingBarrier`, there is no such method to allow guarded network output. This means that networked output must be committed to, and the general approach of guarding all events cannot be used. However, as the mobile process will not span local machine boundaries, this is not an issue. The remote process communicated to via the network channel shall not be part of the mobile and should be unaware of the migration of the process.

Thus the networked output can be committed to in a write operation without fear of deadlock, only input need be guarded upon.

Another limitation comes from the lack of shared input and output guards (from the Any-2-One, One-2-Any and Any-2-Any channels). Therefore, certain forms of input and output cannot be guarded. A simple method to overcome this is to place a multiplex / demultiplex process within the channel to handle the relative input and output transactions. This comes at a cost of resources and performance for expansion / contraction of the shared end, the selection sequence within the process, and the need of an extra process.

A limitation also exists for this approach when using the current implementation of `AltingBarrier`, a possibility hinted at in [160]. This is not an error in the `AltingBarrier` itself, as it provides the mechanism required for multi-way synchronous event. However, for the approach to process mobility described, a prioritised `AltingBarrier` is required. The `AltingBarrier` operates by only allowing one process in the system to be in operation within an `AltingBarrier` at any one time, using a coordination object [137]. This leads to the following problem:

> *Given a set of processes A that synchronise on AltingBarrier a, if there are two or more disjoint subsets of A that always offer choice between their own AltingBarrier and a, then a can never be selected.*

For example consider Figure 70.

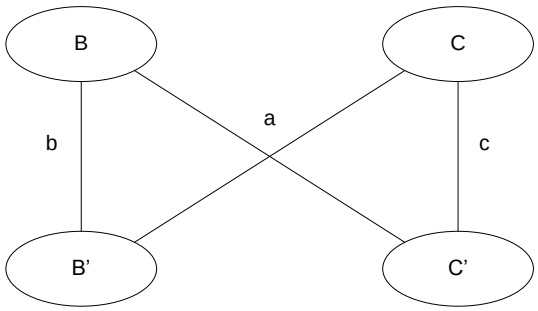


Figure 70: AltingBarrier Sample Process Network

The processes can be defined as follows:

$$B := a \rightarrow B \mid b \rightarrow B$$
$$B' := a \rightarrow B' \mid b \rightarrow B'$$
$$C := a \rightarrow C \mid c \rightarrow C$$
$$C' := a \rightarrow C' \mid c \rightarrow C'$$

a, *b* and *c* are `AltingBarriers`. To operate, a count on the number of required synchronisations within the `AltingBarrier` is kept. When this reaches 0 the relevant `AltingBarrier` is fired. In the above example, *a* has a count of 4 and *b* and *c* both have a count of 2. As only one process can operate on `AltingBarriers` at any one time, there are determinable outcomes.

B activates first and offers *a* and *b*, taking the counts down to 3 and 1 respectively. There are three possible outcomes:

1. *B'* activates next and offers *a* and *b*. The count on *b* reaches 0 and it is selected. Thus the offers on *a* are removed taking the count back to 4.
2. Either *C* or *C'* activates and offers *a* and *c*, taking their respective counts to 2 and 1. *B'* activates next and offers *a* and *b*, taking the count on *b* to 0 and selecting it. Thus two offers on *a* are removed taking the count back to 3.
3. *C* then *C'* are activated in succession (or vice-versa), thus taking the count on *a* to 1, but the count on *c* to 0. Thus *c* is selected and two offers on *a* are removed, taking the count on *a* back to 3.

Similar arguments can be given for *B'*, *C* and *C'* activating first. Thus it can be seen that it is impossible for *a* to ever be selected. This leads to a problem when implementing the migration method. As all processes must synchronise on *migrate* for migration to occur, no more than one disjoint set of sub processes can offer another guarded synchronisation among them. As the guarded input and output `One2OneChannelSymmetricInt` channel does this to provide guarded output, there is a danger that the internal processes will never sync on *migrate*. In fact, the `MobileNumbersInt` will suffer from this problem if the output channel (*d*) is always willing to accept messages.

To overcome this problem, the technique for making individual parts of the process network mobile can be used (Section 8.3.3). Effectively, this approach can be used

to the point where each individual process is given its own `AltingBarrier` that acts as a switch to turn off a process. However, this will come at a greater overhead for normal process operations and migration operations.

Relative overhead in comparison to normal operation is also a consideration. As each communication / synchronisation event must be guarded upon, there is the added overhead of performing Alternation on these events. For example, consider Table 10 which presents the CommsTime benchmark with fast integer channels performed normally, using guarded channels, and with mobile processes. Communication overhead alone is significant without considering the migration process itself. The times presented are the iteration times in microseconds.

Table 10: CommsTime for Mobiles

	CommsTime	CommsTime Symmetric	Mobile CommsTime Parallel Shutdown	Mobile CommsTime Sequential Shutdown
PC	62 micros	123 micros	168 micros	168 micros
PDA	681 micros	1922 micros	2915 micros	2920 micros

The other time to consider is the time taken to shutdown processes using these methods. This is presented in Table 11, which provides the shutdown times in milliseconds of `MobileNumbersInt` processes using the normal technique (Par) and the sequential technique (Seq). As a `Numbers` process consists of numerous internal processes, the number of processes suspended is up to five times the number of `Numbers` processes. Note that this time also incorporates the time taken to reclaim any threads used within the internal processes.

Table 11: Suspending Numbers Processes

Numbers Processes	1	2	4	8	16	32	64
PC Par Shutdown	0.14	0.36	0.76	1.03	1.32	2.22	3.88
PC Seq Shutdown	0.21	0.67	1.61	2.94	6.35	12.94	29.26
PDA Par Shutdown	3.2	9.58	20.57	42.03	53.51	95.79	103.33
PDA Seq Shutdown	2.04	14.23	41.69	96.63	183.2	371.02	752

Table 11 indicates that the time to shutdown processes increases in an approximately linear fashion when using the parallel shutdown technique. The sequential shutdown does have a higher overhead, as expected.

Another overhead incurred by implementation of this approach is the addition of extra processes to handle internal parallelisation. Much of the argument presented thus far has been on the removal of processes whenever necessary when using JCSP in a Ubiquitous Computing context. Therefore, careful consideration must be taken when designing a process network that is intended to be mobile.

8.5 Summary

In this chapter, a discussion on how process mobility can be achieved in JCSP and other process oriented architectures has been presented. Initially, other approaches to capture process state were analysed and found to have certain limitations. In particular, the approach of capturing thread state is found to be limited due to the lack of understanding of thread boundaries, which reflect on the fact that it is difficult to decide what should be migrated in a thread orientated system. These are the same problems that are apparent in Java object serialization, where aliasing problems highlight the issues with self referential and circular graph topologies. Process based approaches are more promising because of strict boundaries and ownership of processes and data, meaning that there is no ambiguity over what should be migrated. However, many of the proposed solutions for shutting down process networks lack capabilities when considering mobile process models.

A solution to these limitations has been proposed which should overcome many of the issues presented. However, it also has limitations. Due to subtleties within some designs, and certain limitations within current process oriented implementations, the method is currently not mechanical except in a specification point of view. Future work on implementing prioritised multi-way synchronisation that can be used in any context will overcome many of these problems.

A question is raised however on whether this is the type of mobility of a process that is required. Strong mobility (the ability to capture execution state at any point)

is probably not required and may indeed be impractical, especially as a process can take any of its external connections (channels) with it as it migrates and alting on every possible event incurs an overhead. CSP processes are generally defined by their external behaviour, so it is possible for a process to move at any point without concern over whether it behaves as expected. The only external communications to a process to consider are channels connected to the local execution environment. In other words, it is possible to limit the type of mobility to constrained mobility. Constrained mobility allows migration with execution state, but only at certain well defined state points. Ensuring that an entire process network does shutdown correctly prior to migration is therefore still considered difficult from this point of view.

Chapter 9 Conclusions and Future Work

In this chapter, final conclusions are drawn from the work presented in the rest of this thesis, and some future work presented. Section 9.1 discusses the suitability of JCSP Networking as a framework for Ubiquitous Computing, and Section 9.2 discusses mobility within the context of JCSP Networking and Ubiquitous Computing. Finally, Section 9.4 presents future work.

9.1 Suitability of JCSP Networking for Ubiquitous Computing

The major question asked was the suitability of JCSP Networking as a framework for Ubiquitous Computing. There are a number of different facets of JCSP Networking that have been examined, in particular towards performance in a more resource constrained environment than JCSP Networking was originally designed for. From the examination of the original architecture in Chapter 3 and the experimental data presented in Chapter 4 it can be judged that the original implementation of JCSP Networking had some fair performance characteristics, providing throughput on a PDA similar to the throughput from the underlying network connection.

Other features available or easily implemented within JCSP Networking also promote possible usage within the context of Ubiquitous Computing. In particular, Ubiquitous Computing requires a sense of adaptability and dynamic interactions which are possible in JCSP Networking utilising channel mobility and code mobility. The partially transparent interface between networked and local interaction provided by the channel mechanism within JCSP Networking allow much of this dynamic architecture to be implemented either locally or remotely, thus increasing the usefulness of the dynamic architectures beyond what the Java object model can provide.

However, the original implementation of JCSP Networking had some issues when considering some of the other requirements of Ubiquitous Computing architectures, particularly when considering more resource constrained and heterogeneous application areas.

9.1.1 Problems with the Current Implementation

Chapter 2 uncovered a number of properties that are desirable within a Ubiquitous Computing framework, beyond the dynamic architectures that mobility helps to support. To examine the problems within the original JCSP Networking implementation, some of these properties are returned to and examined within the context of Ubiquitous Computing. Other issues relating to these properties are also examined individually.

9.1.1.1 Interoperability

A key feature of Ubiquitous Computing is a sense of interoperability between numerous, heterogeneous platforms. Any feature of a framework that reduces inter-platform communication should be considered as a serious problem when considering Ubiquitous Computing. With JCSP Networking, such a problem exists with the heavy reliance on object serialization, which makes inter-framework communication difficult.

The main problem when considering interoperability and JCSP Networking is the usage of objects to describe messages. As these objects are serialized upon the outgoing stream, any framework wishing to communicate with JCSP Networking requires a method to interpret these object message headers. Although this can be built into a framework, it would require extra computational resources to do so. Not every version of Java supports object serialization and thus there is even a limitation for cross-Java communication.

Another issue relating to the reliance on object serialization is that data sent between two communicating systems within JCSP must be a Java object. This again requires other communicating platforms to be able to interpret serialized Java objects to allow communication. This limitation can be circumvented by converting

the data to be sent into a byte array, and providing the receiving framework the ability to strip the object header for the byte array, and reconstruct the data as required. This requires extra functionality and computation, and limits the overall interoperability between heterogeneous frameworks.

For ubiquity between frameworks, a ubiquitous protocol is required that enables the communication functionality within JCSP Networking to be replicated. The protocol should not be locked into a particular platform, but should permit inter-platform communication. Thus, data transfer becomes the key problem, unless a well defined data transfer mechanism is likewise developed. Common data transfer negates the usage of common data structures usually implemented in Java (i.e. cyclic graphs) as not all frameworks will allow such complex data structures. This problem is therefore hard, due to the different data structures and encoding mechanisms in place. The usage of existing data transfer techniques such as XML may overcome this problem somewhat, but this will reduce communication performance and require a greater amount of computational resources to achieve. On resource constrained devices, this will cause a problem.

In general, interoperability between diverse frameworks is hampered by the sole reliance within the original implementation of JCSP Networking on Java object serialization. Any such reliance on a specific framework feature is to be avoided whenever possible, and thus Java object serialization must be avoided.

9.1.1.2 Performance

The performance of JCSP Networking from a communication viewpoint is not far removed from the bare network communication mechanism on a small device. The experimental data presented has shown that a PDA performs at close to optimum throughput for large data sizes. Smaller data sizes show a performance reduction, but this is largely due to the extra message overhead in the original implementation of JCSP Networking, and the synchronisation that occurs when using a standard networked channel.

However, serialization on small devices can reduce throughput due to the extra computation time required to convert a Java object into an array of bytes. For

sufficiently complex objects, performance can drop significantly. Thus, serialization should be avoided whenever possible. Throughput is bound by the (de)serialization performance of the PDA used within the experiments presented, and therefore removing serialization will increase performance for basic communication. This argument against the usage of Java serialization in small devices leads to the question of Java's usage in general for communicating systems on resource constrained frameworks. If the sending of object data between two small devices should be avoided, then applications can be developed outside Java. Thus, the general argument that Java supports Ubiquitous Computing due to its ubiquity across platforms is weakened.

The bounding of the performance of object message communication by serialization may appear initially as incorrect. However, the performance characteristics of the PC and PDA show that serialization performance is within the bounds of the variance between the two devices. Thus, it can be deemed that serialization time is the largest contributor to object communication within the experimental framework.

Serialization does not appear to be related to object creation time. Object creation time is related to the amount of memory required for the object, and is thus based on memory allocation time. Serialization, and in particular deserialization, should also be related to memory allocation time as the object must be re-created. On small devices, this does not appear to be the case. The JVM utilised on the PDA within the experiments showed serialization performance below both I/O throughput of the network, and object creation time.

9.1.1.3 Resource Usage

Except for the reliance on Java serialization within the original implementation of JCSP Networking and the problems this causes, the major issue when considering JCSP Networking within Ubiquitous Computing environments is the high resource usage. The number of created processes within the original JCSP Networking architecture limits the usage of JCSP Networking on resource constrained devices. Numerous processes were spawned to service the architecture, and subsequently

this caused a limitation on the usage of JCSP Networking for large scaled, distributed systems with numerous small devices. Some processes were spawned and subsequently destroyed during connection between devices, and this process could also occur when a connection between the two devices already existed. Considering the limited resources on small devices, temporary process creation could cause the application to run out of memory.

Considering JCSP as a whole, it is arguable that any requirement of Ubiquitous Computing on Java is a limitation. A JVM requires extra resources to operate, and for the smallest scale devices this will likely negate the possibility of running Java and subsequently JCSP. Therefore, if the fundamental ideas of JCSP and JCSP Networking are of importance, then the requirement is to replicate these ideas within other frameworks but allow interaction with JCSP Networking.

9.1.1.4 System Overhead

Within the original implementation of JCSP Networking, system overhead is a problem. Throughput performance is reasonable, but resource usage is high. Another factor is the high priority given to I/O operations, which can lead to computation being starved of resources as I/O is serviced. This may or may not be an issue depending on the application context. However, the inability to modify this property causes a limitation. It has been shown that it is possible to flood a small device with messages, and thus break an application. Although the experimental data gathered utilises functionality within JCSP Networking which should not be used in such a manner, the same outcome could occur by having multiple fast devices communicating to a single slow device.

9.1.1.5 Scalability

An important characteristic of Ubiquitous Computing is the sense of scale envisioned within such environments. From the observations of resource usage and system overhead within the original implementation of JCSP Networking, it can be argued that scalability is a problem. As the number of inter-device connections increase, and likewise the abstractions used to communicate within those devices increases in number, resource usage and system overhead will increase. For

Ubiquitous Computing scenarios, JCSP Networking is unlikely to be useful for larger scaled applications.

9.1.1.6 Stability

JCSP Networking suffers from a number of stability problems. In particular, poor error handling within the underlying architecture causes a problem for error prone applications. Within Ubiquitous Computing, error handling is seen as a key feature, and JCSP Networking cannot provide a reliable level of error protection. Ubiquitous Computing environments envision numerous small devices interacting together, and these devices may fail. As user interaction is considered to be minimal and abstracted, it is unlikely that these devices can be easily reset. The main issue with JCSP Networking when considered in such a context is that a device failing could cause another device to fail due to the poor error handling to detect the disconnection of the device. This could spread across an entire Ubiquitous Computing environment.

Another stability problem relates to the high priority I/O. It has been shown that a device can be caused to fail due to flooding as I/O is serviced while the application cannot actually complete the I/O operation, thus leading to the internal buffering increasing beyond the capabilities of the device.

9.1.1.7 Accessibility and Extensibility

A key problem with JCSP Networking, related to many of the issues discovered when considering JCSP Networking in the context of Ubiquitous Computing, is the accessibility and extensibility of the architecture. The tightly coupled implementation leads to difficulties when attempting to add new features to JCSP Networking, or extend upon existing features. This leads to extensions being built using existing abstractions, and it has been shown that the existing architecture utilises numerous resources to achieve these abstractions. If extensions are built upon the existing primitives, then required resources for these extensions will also be high.

Access to low level properties within the original JCSP Networking implementation is limited, and thus JCSP Networking cannot be adapted to suit differing application areas. This is not a problem if JCSP Networking is utilised within its designed application area of cluster computing, but Ubiquitous Computing scenarios dictate versatility. The hiding of these properties from manipulation by application developers is a limitation.

9.1.1.8 Usage of Java Serialization

JCSP Networking relies heavily on Java object serialization, and this is a limitation across a number of properties of interest within Ubiquitous Computing. On small devices, serialization is slow, and the device is bound by the serialization process rather than I/O performance. Object messages in resource constrained scenarios should therefore be avoided.

The communication mechanism in JCSP Networking relies on Java serialization without consideration for the sent data. This inhibits communication between disparate frameworks as each requires functionality to be able to interpret the message header. Although it is possible to work round this limitation, it is not ideal. Also, the extra overhead associated with the (de)serialization of the message header could be reduced.

9.1.1.9 Usage of Java

Reliance on Java is also considered a problem. It cannot be considered that a JVM will be available for every possible device in a Ubiquitous Computing environment, particularly for the smallest factor devices. The argument that Java is the key platform for Ubiquitous Computing due to the ubiquity of Java is a weak one. Ubiquitous Computing should not rely solely on any platform, and it has been shown that object serialization reduces performance on small factor devices, and also inhibits inter-framework communication. This raises a question on why Java should be seen as the key Ubiquitous Computing platform when communication is better suited to mechanisms possible without Java.

A probable answer to Java being seen as the key platform is the code mobility mechanisms that enable some of the more dynamic architectures by utilising mobile components. However, research in the mobile agent field has also highlighted problems when viewing Java as the desirable method for such application areas. When considering numerous frameworks, code mobility becomes difficult to impossible.

9.1.1.10 Lack of Communication Protocol

A key problem when considering inter-framework communication relying on the ideas within JCSP Networking is the lack of a well defined and documented protocol. Although there are numerous implementations across different frameworks of the virtual channel model that is utilised within JCSP Networking, none of these frameworks can communicate as there is no universal protocol to determine how they should communicate. If the basic channel mechanism is seen as a suitable abstraction for Ubiquitous Computing, then a protocol to enable the channel abstraction between frameworks is required.

The lack of a protocol also inhibits mobility. Connection mobility protocols are required to enable the dynamic topologies within Ubiquitous Computing environments. If the base communication mechanism in JCSP Networking does not have a well defined protocol, then adding channel mobility to JCSP Networking becomes a problem.

9.1.2 Overcoming the Problems in JCSP Networking

To overcome the limitations and problems of JCSP Networking when considering the framework within a Ubiquitous Computing context, a new implementation of JCSP Networking has been developed. This new implementation overcomes the limitations of the original implementation in a number of ways.

9.1.2.1 Reduced Architecture

To overcome the resource overhead, the new implementation has reduced the number of required resources. By removing processes and either replacing them with shared data objects or folding the functionality into existing components, the

basic architecture is now more lightweight. There are still processes that could possibly be removed, although there are limitations when considering reduced Java implementations in this respect.

The basic networked channel mechanism is now lightweight, and utilises no extra processes beyond those required for the inter-device connection. By doing this, there is less chance of processes being left operational when references to channels are lost, and the removal of the `NetChannelInputProcess` is one of the key factors to resource usage reduction.

9.1.2.2 Removal of Reliance on Serialization

There is no longer reliance within the new architecture upon Java object serialization. By abstracting data encoding functionality into a user manipulative manner, and by removing the usage of objects as message headers, serialization is no longer required within JCSP Networking. It is still possible to utilise serialization if this is seen as a suitable mechanism for data transfer between two devices.

9.1.2.3 Abstraction of Data Encoding

As data encoding has been abstracted, it is possible to implement custom mechanisms to convert data when transferring between devices. If two devices agree upon the data encoding mechanism to be used, then it becomes possible to have inter-framework interoperability, and thus reliance on Java and JCSP Networking is removed. If other implementations of JCSP Networking on different frameworks exhibit similar behaviour while communicating with one another, then agreement on data encoding becomes the sole problem for inter-framework communication.

9.1.2.4 Communication Protocol

By developing a communication protocol in a platform independent manner, inter-framework communication has become further enabled. By defining the message types and headers, replication within other frameworks becomes possible. This strengthens the usage of JCSP Networking and communicating process architectures in general when considering Ubiquitous Computing. As devices can

have the protocol built into their functionality, data encoding mechanisms become the core problem for inter-device communication within a virtual channel model.

9.1.2.5 Performance

As reliance on serialization has been removed, performance has increased somewhat. Although communication performance within the original JCSP Networking implementation was not seen as a problem when considering the synchronous nature of the communication, any increase in performance should be seen as favourable.

9.1.2.6 Verified Model

To overcome the erroneous behaviour exhibited by JCSP Networking, a model has been developed using the SPIN model checker. The model has been verified with no errors, thus some of the erroneous behaviour of JCSP Networking has been removed. However, further work needs to be undertaken to further examine JCSP Networking in a larger Ubiquitous Computing context to investigate other possible problems that may occur.

In summary, the development of a new implementation of JCSP Networking and the creation of a communication protocol to enable inter-framework communication has improved the usefulness of JCSP Networking and possibly other communicating process architecture based frameworks within the context of Ubiquitous Computing.

9.2 Mobility

The key reason to investigate JCSP Networking within the context of Ubiquitous Computing was the potential availability of distributed mobility. Channel and process mobility models have been proposed as likely architectures to enable development of the complex and dynamic topologies that Ubiquitous Computing exhibits, and the possibility of implementing these constructs within JCSP Networking would enable construction of systems based on the channel and process mobility models.

9.2.1 Advantages of Communicating Process Architecture Mobility

What are the practical advantages of communicating process architectures in comparison to the common object-oriented approaches used in agent based systems when considering mobility? The clear advantage of using a process oriented approach to mobility is that there is no question about what should be moved during a migrate operation. As a process completely encapsulates everything internal to the process, then when a process migrates everything it owns should move with it. Thus, the definition of a mobile process can easily be described, and contains the channel connections, internal processes and data relevant to the process. This is unlike object-oriented mobility, where it is unclear whether an object should or should not move when it is shared, or how the connection between a stationary object and a mobile one should be managed.

Within communicating process architecture mobility, connection and component mobility are completely independent, which is unlike object based systems. If an object moves, then if it is to move everything it owns, a question is raised on how to handle shared objects. If a shared object is moved, should the remaining objects be linked to the object via, now networked, references, or should the remaining objects completely disconnect from the shared object or likewise the migrating object disconnect from the stationary shared object? If the migrating shared object is copied, what occurs when a previously connected object arrives at the new location of the migrating shared object? As object-orientation does not distinguish between connection and component, this becomes a problem. As communicating process architectures treat connections (channels) and components (processes) separately, there is no such problem. Any shared resource is protected within a process and accessed via a channel interface. If the shared resource is moved, then the channel interface owned by the process would be moved, but the other connection ends would not. If a process connected to the resource were to move, it would take its connection to the resource with it. Thus, there is no question of what to migrate and how to handle shared resources.

The notion of strong code mobility does not in essence capture process mobility. Data and behaviour state are similar, but strong code mobility centres on the code

rather than the component. Code may not be required in a migration, and is arbitrary. Much can be inferred by the type of the process, and this can be carried between different frameworks where code cannot. There are dangers if the definition of the type is different however. Connection migration is also not considered in code migration. Thus, strong component mobility should be considered separate from code mobility and include connection mobility, and code mobility considered arbitrary depending on circumstance.

9.2.2 Channel Mobility

Chapter 7 discussed possible options for channel mobility, the question being whether a suitable model of channel mobility can be developed that allows the dynamic topologies envisioned by Ubiquitous Computing. The call for protocols to enable mobility requires a decision on the type of mobility model best suited for Ubiquitous Computing. A general agreement is required to enable channel mobility across frameworks. From a Ubiquitous Computing context, many differing models may be suitable. If channel mobility is examined in a larger context, then a suitable model for Ubiquitous Computing may not be suitable for cluster based computing.

To develop a suitable protocol for channel mobility, a number of considerations must be taken. In particular, if a suitable mobility protocol is developed, the description of a mobile channel and a suitable model must be developed. A general description of a mobile channel can be developed based on the location of the channel end, but extra information may be required for certain models.

9.2.3 Process Mobility

The aim was to provide the strong mobility behaviour required for agent mobility with JCSP Networking, and thus permit the adaption and dynamic architectures for Ubiquitous Computing. Some work towards providing a suitable technique to enable strong process mobility within JCSP Networking has been developed, although usage across all possible applications is still questionable. By examining behaviour from an event viewpoint, instead of focusing on individual Java statements, it becomes easier to handle migration. In particular, the agreement of the mobile process to move is seen as a key property, instead of the migrating unit

being forced to move thus causing inconsistent state. As overzealous usage of the strong mobility idea is partially to blame for the attempts at forcing component migration like this.

Capturing process state should be possible based on event behaviour. It is also possible to discern what should be moved and when. Committing to external events of the mobile does not actually cause a problem, but internal events must be deactivated. However, the externally observable behaviour of a process dictates whether a process is strongly mobile.

Multi-framework applications make code mobility a problem, and thus behaviour mobility is limited. Therefore, code mobility should not be relied upon to enable the migration of events between frameworks. If code mobility is required, then limitations must be placed upon its usage to avoid problems when communicating with differing frameworks. The usage of a ubiquitous communication protocol overcomes inter-framework communication to a certain degree. Code mobility requires a framework such as Java which enables dynamic code loading, but if code mobility is limited then the argument of Java as the Ubiquitous Computing language is limited.

It is also questionable whether strictly strong mobility of processes is required in an application. Considering the overheads and other difficulties when migrating a process at any point during its execution, a much better approach to process mobility is constrained mobility. With this viewpoint of mobility, only at certain well defined points in execution is it possible to migrate a process. The requirement is that the process must be in a consistent state to allow migration, and many of the other problems associated with other software models when dealing with component mobility are overcome by enabling connection mobility. Further considerations on what should be taken with a mobile process are also required before deciding when and how to migrate a mobile process.

9.3 Summary

The two questions posed at the start of this thesis questioned two different aspects of JCSP Networking in the context of Ubiquitous Computing. The first question was

the suitability of JCSP Networking as a framework for Ubiquitous Computing applications, and this has both a positive and negative answer. The original implementation of JCSP Networking was not a suitable framework for developing Ubiquitous Computing applications, particularly due to the scalability problems caused by excessive resource usage and reliance on Java serialization. The new implementation overcomes these issues, but requiring a Java Virtual Machine to operate the JCSP Networking architecture is still seen as a problem. The implementation of the new communication protocol alleviates some of this problem.

The second question focused on the ability to build suitable mobility models to allow practical implementation of distributed channel and process mobility within JCSP Networking, and this question is left unanswered. Channel mobility models have been examined, and while possible suitable models for channel mobility within Ubiquitous Computing have been identified, further examination of these models within different contexts and frameworks is required. A possible method of process mobility has also been proposed, but it too requires further examination to determine how practical this method is in all circumstances.

9.4 Future Work

There are a number of future directions that have been opened from the work presented in this thesis. Firstly, it must be considered that the second posed research question has not been answered, which is due to further work required in the areas of implementable channel and process mobility models. For channel mobility, actual implementation of the proposed models is required for examination within the context of both Ubiquitous Computing and other usages of JCSP Networking principles. Possible solutions lie with providing hybrid approaches that accommodate both Ubiquitous Computing and cluster computing, such as utilising server based mobility which utilises message box or mobile IP style mobility when channels must stretch across domains. For process mobility, further analysis on what is actually required for component mobility needs to be conducted. This further analysis will enable a more concrete approach towards constrained process

mobility – process mobility that allows capturing of execution state but at fixed points.

One of the key areas of future work highlighted in this thesis is the further development of a generalised protocol and architecture for CSP based distributed computing. Further analysis of the requirements of the general protocol will enable the discovery of other message types and further development of the message header structure to incorporate these message types. Development of the protocol and architecture has benefits far outside the area of Ubiquitous Computing and will enable levels of communication between CSP frameworks which is currently severely lacking. Languages such as **occam**, Erlang, C++ and Python can all benefit from a unified approach to distributed channel based applications. Analysis and comparison of the protocol against existing approaches such as the session layer protocol Remote Procedure Call (RPC) can provide insight into further requirements. Further refinement of the architecture may also lead to further performance improvements and resource usage reduction, and the possibility of building a generic reference architecture library usable by all frameworks would be advantageous.

Further consideration within the architecture and protocol for channel mobility is another key direction highlighted in this work. A number of possible approaches to channel mobility have been discussed in Chapter 7, and further analysis of these approaches is required to discover which is the most suitable across a broad range of application areas. This further analysis includes implementation and case study work to examine the different models within different application areas. Also, it will enable development of the necessary structures within the architecture required to permit channel mobility. The discovery of the necessary protocol messages required to support channel mobility will enable the protocol to be updated to accommodate the necessary channel mobility model between the different CSP based frameworks. The development of such a channel mobility mechanism will provide a great deal of high level functionality which can be utilised in a number of contexts, and is itself an interesting area of future research.

Further work within the area of process mobility will enable a better understanding of the requirements for constrained process mobility. Although current work within the area of strong code mobility is leading towards the capturing of active component state at any point in its execution, this is largely due to the lack of sufficient connection mobility structures that enable a migrating component to stay connected to its communication partners. Therefore, process mobility should not aim for such strength, as it is likely to be difficult to achieve efficiently. Instead, further work in this area requires analysis of when a process should be able to move, and how complex process mobility can be achieved safely. The possibility of enabling process mobility between different frameworks is also an area of interest; this cannot be directly supported by of code mobility however.

With a generalised protocol and architecture, along with mechanisms for providing channel and process mobility, work into developing mechanisms that will enable transparent channel and process mobility is required. The ability of local channels or processes to be sent across a networked channel and for the necessary architecture put in place to support the now distributed application would provide transparent functionality. Although pony does provide some of this capability, it does so at a cost to performance, and a better approach is required. Enabling this functionality within the protocol and architecture will be advantageous, and may allow transparent handover of processes and channels between frameworks.

Another addition to the general CSP network protocol and architecture is the support for distributed multi-way synchronisation events. A networked `AltingBarrier` would enable distributed choice on multi-way events, and the addition of such a construct would bring JCSP Networking to the same level of functionality as the core package of JCSP. Such a construct cannot utilise the current approach to multi-way synchronisation within a single running application, due to the nature of the `AltingBarrier` implementation. This is due to the construct not permitting parallel access to the `AltingBarrier` by multiple processes. Although on a single machine, concurrent behaviour implies only one process may run, and hence single access is not considered an issue, with distributed parallel architectures, the implication is that only one device on the

network can be in engaged in the selection sequence at a time, although other processes may still be in operation. Further work in this area must focus on the necessary architecture to permit distributed multi-way synchronisation, and the necessary messages within the protocol to support these constructs. A centralised coordinator will not be sufficient to control all multi-way synchronisations within the network, and a method that performs some of the work on a local machine is required. This approach implies a two layered method of coordination (local and remote), with the local step consolidating and controlling as many local messages as possible prior to coordination with a server. However, there will still be an issue with a single process being in coordination with the main networked coordinator at a time.

With a suitable architecture for general CSP networking in place, further work within the area of Ubiquitous / Pervasive Computing is required to analyse the suitability of such an approach for Ubiquitous Computing applications. This work requires case study type analysis, and certain features of Ubiquitous / Pervasive Computing must be taken into consideration. The sheer scale of Ubiquitous Computing requires careful decisions and analysis into how a CSP based approach can help understanding and reasoning, and careful consideration must also be taken into the resource constrained nature of many Ubiquitous Computing devices. Possible comparison with other approaches to Ubiquitous Computing is also required, such as comparing the mobility features of a general CSP networking architecture to mobile agent approaches. When considering resource constraints, further analysis of smaller runtimes, such as the Transterpreter [161], may show more suitability, and therefore work in this area to implement and examine the network architecture and possible code mobility requirements is a further future area of interest.

Further work within the core functionality of JCSP is also required. During the development of a process mobility mechanism, it was highlighted that the current implementation of the `AltingBarrier` could not achieve the functionality required by the approach. This requires the development of a prioritised version of a multi-way synchronous event, which would allow certain event combinations to

be chosen. The ability to know when to wait for further offers, and when to chose an event with a waiting count of zero is an open question, and is difficult to achieve, considering that certain further offers may contain higher priority events.

Another problem highlighted when process mobility was considered was the current lack of shared channel guards. Adding this functionality to the core package of JCSP would enable all channels to be used guards within an alternative. Consideration must also be taken to how well the problem to this solution will scale. Developing a shared channel guard is possible using multiple `AltingBarrier` events, but would require an `AltingBarrier` for each shared end. A solution that provides fast resolution of choice on shared events is more suitable. Possible areas of investigation include utilising the other capabilities in the Java concurrency library to allow the level of functionality that is required.

In summary, the work highlighted within this thesis has shown that there is still a great deal of future areas that require further examination and development before the required functionality to support Ubiquitous Computing can be suitable achieved.

References

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

[2] J. Kerridge, J.-O. Haschke, and K. Chalmers, "Mobile Agents and Processes using Communicating Process Architectures," in P. H. Welch, S. Stepney, F. A. C. Polack, F. R. M. Barnes, A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson (Eds.), *Communicating Process Architectures 2008*, pp. 397-409, IOS Press, Amsterdam, 2008.

[3] J. Kerridge and K. Chalmers, "Ubiquitous Access to Site Specific Services," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), *Communicating Process Architectures 2006*, pp. 41-58, IOS Press, Amsterdam, 2006.

[4] M. Weiser, "The Computer for the 21st Century," *Scientific American,* September, 1991.

[5] M. Weiser, R. Gold, and J. S. Brown, "The Origins of Ubiquitous Computing Research at PARC in the Late 1980s," *IBM Systems Journal,* 38(4), pp. 693-696, 1999.

[6] B. N. Schilit, N. Adams, R. Gold, M. M. Tso, and R. Want, "The PARCTAB Mobile Computing System," in *Fourth Workshop on Workstation Operating Systems*, pp. 34-39, IEEE Computer Society, 1993.

[7] R. Want, A. Hopper, V. Falcão, and J. Gibbons, "The Active Badge Location System," *ACM Transactions on Information Systems (TOIS),* 10(1), pp. 91-102, 1992.

[8] K. Henricksen, J. Indulska, and A. Rakotonirainy, "Infrastructure for Pervasive Computing: Challenges," in *Informatik Workshop on Pervasive Computing and Information Logistics*, pp. 214-222, 2001. Available from: http://henricksen.id.au/publications/Informatik01.pdf

[9] R. Campbell, J. Al-Muhtadi, P. Naldurg, G. Sampemane, and M. D. Mickunas, "Towards Security and Privacy for Pervasive Computing," in M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa (Eds.), *Software Security - Theories and Systems*, *Lecture Notes in Computer Science 2609*, pp. 77-82, Springer Berlin / Heidelberg, 2003.

[10] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes, I," *Information and Computation,* 100(1), pp. 1-40, 1992.

[11] R. Milner, "Ubiquitous Computing: Shall we Understand It?," *The Computer Journal,* 49(4), pp. 383-389, 2006.

[12] P. H. Welch and F. R. M. Barnes, "Communicating Mobile Processes - Introducing occam-pi," in A. E. Abdallah, C. B. Jones, and J. W. Sanders (Eds.),

Communicating Sequential Processes: The First 25 Years - Symposium on the Occasion of 25 Years of CSP, *Lecture Notes in Computer Science 3525*, pp. 175-210, Springer Berlin / Heidelberg, 2005.

[13] 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.

[14] 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.

[15] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, Inc., 1985.

[16] A. W. Roscoe, *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[17] 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.

[18] D. Saha and A. Mukherjee, "Pervasive Computing: A Paradigm for the 21st Century," *IEEE Computer,* 36(3), pp. 25-31, 2003.

[19] D. M. Konidala, C. Y. Yeun, and K. Kim, "A Secure and Privacy Enhance Protocol for Location-based Services in Ubiquitous Society," in *IEEE Global Telecommunications Conference 20044*, pp. 2164-2168, IEEE Computer Society, 2004.

[20] C. A. da Costa, A. C. Yamin, and C. F. R. Geyer, "Toward a General Software Infrastructure for Ubiquitous Computing," *IEEE Pervasive Computing,* 7(1), pp. 64-73, 2008.

[21] M. Weiser and J. S. Brown, "The Coming Age of Calm Technology," in P. J. Denning and R. M. Metcalfe (Eds.), *Beyond Calculation: The Next Fifty Years of Computing*, Copernicus, 1998.

[22] A. A. Araya, "Questioning Ubiquitous Computing," in *1995 ACM 23rd Annual Conference on Computer Science*, pp. 230-237, ACM Press, 1995.

[23] G. Banavar and A. Bernstein, "Software Infrastructure and Design Challenges for Ubiquitous Computing Applications," *Communications of the ACM,* 45(12), pp. 92-96, 2002.

[24] G. D. Abowd and E. D. Mynatt, "Charting Past, Present, and Future Research in Ubiquitous Computing," *ACM Transactions on Computer-Human Interaction (TOCHI),* 7(1), pp. 29-58, 2000.

[25] F. Zhu, M. W. Mutka, and L. M. Ni, "Service Discovery in Pervasive Computing Environments," *IEEE Pervasive Computing,* 4(4), pp. 81-90, 2005.

[26] E. Niemela and J. Latvakoski, "Survey of Requirements and Solutions for Ubiquitous Software," in *3rd International Conference on Mobile and Ubiquitous Multimedia*, *ACM International Conference Proceeding Series 83*, pp. 71-78, ACM Press, 2004.

[27] G. Koloniari and E. Pitoura, "Filters for XML-based Service Discovery in Pervasive Computing," *The Computer Journal,* 47(4), pp. 461-474, 2004.

[28] K. Henricksen and J. Indulska, "A Software Engineering Framework for Context-Aware Pervasive Computing," in *Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, pp. 77-86, IEEE Computer Society Press, 2004.

[29] R. S. Cardoso and F. Kon, "Mobile Agents: A Key for Effective Pervasive Computing," in *Second Pervasive Computing Workshop of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, 2002. Available from: http://www.ime.usp.br/~speicys/publications/oopsla2002.pdf

[30] S. Schuhmann, K. Herrmann, and K. Rothermel, "A Framework for Adapting the Distribution of Automatic Application Configuration," in *5th International Conference on Pervasive Services*, pp. 163-172, ACM, 2008.

[31] M. Satyanarayanan, "Pervasive Computing: Vision and Challenges," *IEEE Personal Communications,* 8(4), pp. 10-17, 2001.

[32] S.-W. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, P. Steenkiste, and N. Hu, "Software Architecture-Based Adaptation for Pervasive Systems," in H. Schmeck, T. Ungerer, and L. Wolf (Eds.), *Trends in Network and Pervasive Computing - ARCS 20022299*, pp. 217-233, Srpinger Berlin / Heidelberg, 2002.

[33] J. E. Bardram, "Activity-based Computing: Support for Mobility and Collaboration in Ubiquitous Computing," *Personal and Ubiquitous Computing,* 9(5), pp. 312-322, 2005.

[34] J. Kjeldskov and M. B. Skov, "Exploring Context-awareness for Ubiquitous Computing in the Healthcare Domain," *Personal and Ubiquitous Computing,* 11(7), pp. 549-562, 2007.

[35] British Computer Society, "Grand Challenges in Computing Research," British Computer Society, 2005.

[36] B. Rao and L. Minakakis, "Evolution of Mobile Location-based Services," *Communications of the ACM,* 46(12), pp. 61-65, 2003.

[37] J. P. Munson and V. K. Gupta, "Location-Based Notification as a General-Purpose Service," in *2nd International Workshop on Mobile Commerce*, pp. 40-44, ACM, 2002.

[38] R. José, A. Moreira, H. Rodrigues, and N. Davies, "The AROUND Architecture for Dynamic Location-Based Services," *Mobile Networks and Applications,* 8(4), pp. 377-387, 2003.

[39] H. Gellersen, C. Fischer, D. Guinard, R. Gostner, G. Kortuem, C. Kray, E. Rukzio, and S. Streng, "Supporting Device Discovery and Spontaneous Interaction with Spacial References," *Personal and Ubiquitous Computing,* Online First, 2008.

[40] A. Friday, N. Davies, and E. Catteral, "Supporting Service Discovery, Querying and Interaction in Ubiquitous Computing Environments," *Wireless Networks,* 10(6), pp. 631-641, 2005.

[41] M. Weiser, "Some Computer Science Issues in Ubiquitous Computing," *Communications of the ACM,* 36(7), pp. 75-84, 1993.

[42] T. Kindberg and A. Fox, "System Software for Ubiquitous Computing," *IEEE Pervasive Computing,* 1(1), pp. 70-81, 2002.

[43] J. P. Sousa, B. Schmerl, P. Steenkiste, and D. Garlan, "Activity-oriented Computing," in S. K. Mostefaoui, Z. Maamar, and G. Giaglis (Eds.), *Advances in Ubiquitous Computing: Future Paradigms and Directions*, IGI Publishing, Herschey, PA, 2008.

[44] J. Lindenberg, W. Pasman, K. Kranenborg, J. Stegeman, and M. A. Neerinex, "Improving Service Matching and Selection in Ubiquitous Computing Environments: A User Study," *Personal and Ubiquitous Computing,* 11(1), pp. 59-68, 2005.

[45] K. Henricksen, J. Indulska, T. McFadden, and S. Balasubramaniam, "Middleware for Distributed Context-Aware Systems," in R. Meersman, Z. Tari, M.-S. Hacid, J. Mylopoulos, B. Pernici, O. Babaoglu, H.-A. Jacobsen, M. Kifer, and S. Spaccapietra (Eds.), *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, *Lecture Notes in Computer Science 3760*, pp. 846-863, Springer Berlin / Heidelberg, 2005.

[46] W. K. Edwards, M. W. Newman, J. Z. Sedivy, and T. F. Smith, "Bringing Network Effects to Pervasive Spaces," *IEEE Pervasive Computing,* 4(3), pp. 15-17, 2005.

[47] D. Garlan and B. Schmerl, "Component-Based Software Engineering in Pervasive Computing Environments," in *The 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*, 2001. Available from: http://www.cs.cmu.edu/afs/cs/project/able/ftp/cbse01/cbse01-submission.pdf

[48] D. Hoareau and Y. Mahéo, "Middleware Support for the Deployment of Ubiquitous Software Components," *Personal and Ubiquitous Computing,* 12(2), pp. 167-178, 2008.

[49] F. Zambonelli and M. Luck, "Agent Hell: A Scenario of Worst Practices," *IEEE Computer,* 37(3), pp. 96-98, 2004.

[50] Y. Jung, J. Lee, and M. Kim, "Multi-agent Based Community Computing System Development with the Model Driven Architecture," in *Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 1329-1331, ACM, 2005.

[51] J. M. Molina, J. M. Corchado, and J. Bajo, "Ubiquitous Computing for Mobile Environments," in A. Moreno and J. Pavón (Eds.), *Issues in Multi-Agent Systems*, Birkhäuser Basel, 2007, pp. 33-57.

[52] S. Hartwig, J.-P. Strömann, and P. Resch, "Wireless Microservers," *IEEE Pervasive Computing,* 1(2), pp. 58-66, 2002.

[53] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light, "The Personal Server: Changing the Way We Think About Ubiquitous Computing," in G. Borriello and L. E. Holmquist (Eds.), *4th International Conference on Ubiquitous Computing*, *Lecture Notes in Computer Science 2498*, pp. 223-230, Springer Berlin / Heidelberg, 2002.

[54] T. Moors, M. Mei, and A. Salim, "Using Short-range Communication to Control Mobile Device Functionality," *Personal and Ubiquitous Computing,* 12(1), pp. 11-18, 2008.

[55] G. D. Abowd, L. Iftode, and H. Mitchell, "Guest Editors' Introduction: The Smart Phone: A First Platform for Pervasive Computing," *IEEE Pervasive Computing,* 4(2), pp. 18-19, 2005.

[56] M. Tokoro, "The Society of Objects," in *Conference on Object Oriented Programming Systems Languages and Applications*, pp. 3-12, ACM, 1993.

[57] D. B. Lange and M. Oshima, "Mobile Agents with Java: The Aglet API," *World Wide Web,* 1(3), pp. 111-121, 1998.

[58] C. A. Iglesias, M. Garijo, and J. C. Gonzalez, "A Survey of Agent-Oriented Methodologies," in J. P. Müller, M. P. Singh, and A. S. Rao (Eds.), *5th International Conference on Intelligent Agents: Agents Theories, Architectures, and Languages*, *Lecture Notes in Computer Science 1555*, Springer Berlin / Heidelberg, 1998.

[59] F. Baude, D. Caromel, F. Huet, and J. Vayssière, "Communicating Mobile Active Objects in Java," in M. Bubak, H. Afsarmanesh, R. Williams, and B. Hertzberger (Eds.), *High Performance Computing and Networking: 8th International Conference, HPCN Europe 2000*, *Lecture Notes in Computer Science 1823*, pp. 633-643, Springer Berlin / Heidelberg, 2000.

[60] B. Bauer, "UML Class Diagrams Revisited in the Context of Agent-Based Systems," in M. Wooldrige, G. Weiß, and P. Ciancarini (Eds.), *Second International Workshop on Agent-Oriented Software Engineering*, *Lecture Notes in Computer Science 2222*, pp. 101-118, Springer Berlin / Heidelberg, 2002.

[61] M. Wooldrige and P. Ciancarini, "Agent-Oriented Software Engineering: The State of the Art," in P. Ciancarini and M. Wooldrige (Eds.), *First International Workshop on Agent-Oriented Software Engineering*, *Lecture Notes in Computer Science 1957*, pp. 55-82, Springer Berlin / Heidelberg, 2001.

[62] H. S. Nwana, "Software Agents: An Overview," *Knowledge Engineering Review,* 11(3), pp. 205-244, 1996.

[63] A. R. Silva, A. Ramão, D. Deugo, and M. M. da Silva, "Towards a Reference Model for Surveying Mobile Agent Systems," *Autonomous Agents and Multi-Agent Systems,* 4(3), pp. 187-231, 2001.

[64] E. A. Kendall, P. V. M. Krishna, C. V. Pathak, and C. B. Suresh, "Patterns of Intelligent and Mobile Agents," in *Second International Conference on Autonomous Agents*, pp. 92-99, ACM, 1998.

[65] I. Dickinson and M. Wooldrige, "Towards Practical Reasoning Agents for the Semantic Web," in *Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 827-834, ACM, 2003.

[66] D. Kinny and M. Georgeff, "Modelling and Design of Multi-Agent Systems," in J. P. Müller, M. Wooldrige, and N. R. Jennings (Eds.), *Intelligent Agents III: Agent Theories, Architectures, and Languages*, *Lecture Notes in Computer Science 1193*, pp. 1-20, Springer Berlin / Heidelberg, 1996.

[67] Z. Guessoum and J.-P. Briot, "From Active Objects to Autonomous Agents," *IEEE Concurrency,* 7(3), pp. 68-76, 1999.

[68] A. F. Garcia, C. J. P. de Lucena, and D. D. Cowan, "Agents in Object-Oriented Software Engineering," *Software: Practice and Experience,* 34(5), pp. 489-521, 2004.

[69] B. Bauer, J. P. Müller, and J. Odell, "An Extension of UML by Protocols for Multiagent Interaction," in *Fourth International Conference on Mutliagent Systems*, pp. 207-214, IEEE Computer Society, 2000.

[70] M. Luck and M. d'Inverno, "A Formal Framework for Agency and Autonomy," in *Proceedings of the First International Conference on Multi-Agent Systems*, pp. 254-260, AAAI Press / MIT Press, 1995.

[71] M. Luck, N. Griffiths, and M. d'Inverno, "From Agent Theory to Agent Construction: A Case Study," in J. E. Müller, M. Wooldrige, and N. R. Jennings (Eds.), *Intelligent Agents III Agent Theories, Architectures, and Languages*, *Lecture Notes in Computer Science 1193*, pp. 49-63, Springer Berlin / Heidelberg, 1997.

[72] M. Duvigneau, D. Moldt, and H. Rölke, "Concurrent Architecture for a Multi-agent Platform," in F. Giunchiglia, J. Odell, and G. Weiß (Eds.), *Third International Workshop on Agent-Oriented Software Engineering*, *Lecture Notes in Computer Science 2585*, pp. 59-72, Springer Berlin / Heidelberg, 2002.

[73] H. Xu and S. M. Shatz, "A Framework for Model-based Design of Agent-oriented Software," *IEEE Transactions on Software Engineering,* 29(1), pp. 15-30, 2003.

[74] E. Gonzalez, C. Bustacara, and J. Avila, "Agents for Concurrent Programming," in J. F. Broenink and G. H. Hilderink (Eds.), *Communicating Process Architectures 2003*, pp. 157-166, IOS Press, Amsterdam, 2003.

[75] Z.-h. Yu and Y.-l. Cai, "On Modeling and Analyzing Multi-agent Systems Using π-calculus," *Journal of Shanghai University (English Edition),* 11(1), pp. 58-63, 2007.

[76] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Transactions on Software Engineering,* 24(5), pp. 342-361, 1998.

[77] P. Tröger and A. Polze, "Object and Process Migration in .NET," in *Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 139-146, IEEE Computer Society, 2003.

[78] L. Bettini and R. de Nicola, "Translating Strong Mobility into Weak Mobility," in G. P. Picco (Ed.), *Mobile Agents: 5th International Conference, MA 2001*, *Lecture Notes in Computer Science 2240*, pp. 182-197, Springer Berlin / Heidelberg, 2001.

[79] C. Ghezzi and G. Vigna, "Mobile Code Paradigms and Technologies: A Case Study," in K. Rothermel and R. Popescu-Zeletin (Eds.), *First International Workshop on Mobile Agents*, *Lecture Notes in Computer Science 1219*, pp. 39-49, Springer Berlin / Heidelberg, 1997.

[80] L. Cardelli, "Abstractions for Mobile Computation," in J. Vitek and C. D. Jensen (Eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. vol. 1603, Springer Berlin / Heidelberg, 1999, pp. 51-94.

[81] G.-C. Roman, G. P. Picco, and A. L. Murphy, "Software Engineering for Mobility: A Roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, pp. 241-258, ACM Press, 2000.

[82] A. Phillips, N. Yoshida, and S. Eisenbach, "A Distributed Abstract Machine for Boxed Ambient Calculi," in D. Schmidt (Ed.), *Programming Languages and Systems: 13th European Symposium on Programming, ESOP 2004*, *Lecture*

Notes in Computer Science 2986, pp. 155-170, Springer Berlin / Heidelberg, 2004.

[83] G. Fortino, F. Frattolillo, W. Russo, and E. Zimeo, "Mobile Active Objects for Highly Dynamic Distributed Computing," in *Proceedings International Parallel and Distributed Processing Symposium. IPDPS 2002,*, pp. 118-125, IEEE Press, 2002.

[84] R. R. Brooks, "Mobile Code Paradigms and Security Issues," *IEEE Internet Computing,* 8(3), pp. 54-59, 2004.

[85] A. Lopes, J. L. Fiadeiro, and M. W. Wemelinger, "Architectural Primitives for Distribution and Mobility," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 41-50, ACM, 2002.

[86] Y. Zheng and A. T. S. Chan, "MobiGATE: A Mobile Computing Middleware for the Active Deployment of Transport," *IEEE Transactions on Software Engineering,* 32(1), pp. 35-50, 2006.

[87] F. Oquendo, "π-ADL: An Architecture Description Language Based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures," *ACM SIGSOFT Software Engineering Notes,* 29(3), pp. 1-14, 2004.

[88] R. Milner, "Turing, Computing and Communication," in D. Goldin, S. A. Smolka, and P. Wegner (Eds.), *Interactive Computation: The New Paradigm*, Springer Berlin/Heidelberg, 2006, pp. 1-8.

[89] X. Zhong and C.-Z. Xu, "A Reliable Connection Migration Mechanism for Synchronous Transient Communication in Mobile Codes," in *International Conference on Parallel Processing 20041*, pp. 431-438, IEEE Computer Society, 2004.

[90] D. May and H. Muller, "Copying, Moving and Borrowing Semantics," in A. Chalmers, M. Mirmehdi, and H. Muller (Eds.), *Communicating Process Architectures 2001*, pp. 15-26, IOS Press, Amsterdam, 2001.

[91] F. R. M. Barnes and P. H. Welch, "Mobile Data, Dynamic Allocation and Zero Aliasing: an **occam** Experiment," in A. Chalmers, M. Mirmehdi, and H. Muller (Eds.), *Communicating Process Architectures 2001*, pp. 243-264, IOS Press, Amsterdam, 2001.

[92] C. A. R. Hoare and H. Jifeng, "A Trace Model for Pointers and Objects," in R. Guerraoui (Ed.), *Proceedings ECOOP'99 - Object-Oriented Programming: 13th European Conference.*, *Lecture Notes in Computer Science 1628*, p. 668, Springer Berlin / Heidelberg, 1999.

[93] T. Locke, "Towards a Viable Alternative to OO - Extending the occam/CSP Programming Model," in A. G. Chalmers, M. Mirmehdi, and H. Muller (Eds.), *Communicating Process Architectures 2001*, pp. 329-349, IOS Press, Amsterdam, 2001.

[94] J. Vitek, M. Serrano, and D. Thanos, "Security and Communication in Mobile Object Systems," in J. Vitek and C. Tschudin (Eds.), *Second International Workshop on Mobile Object Systems: Towards the Programmable Internet1222*, pp. 177-194, Springer Berlin / Heidelberg, 1997.

[95] J. Potter, J. Noble, and D. Clarke, "The Ins and Outs of Objects," in *The Australian Software Engineering Conference*, pp. 80-89, IEEE Press, 1998.

[96] J. L. Fiadeiro and A. Lopes, "CommUnity on the Move: Architectures for Distribution and Mobility," in F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever (Eds.), *Second International Symposium on Formal Methods for Components and Objects*, *Lecture Notes in Computer Science 3188*, pp. 177-196, Springer Berlin / Heidelberg, 2004.

[97] P. J. McCann and G.-C. Roman, "Compositional Programming Abstractions for Mobile Computing," *IEEE Transactions on Software Engineering,* 24(2), pp. 97-110, 1998.

[98] L. Andrade, P. Baldan, H. Baumeister, R. Bruni, A. Corradini, R. de Nicola, J. L. Fiadeiro, F. Gadducci, S. Gnesi, P. Hoffman, N. Koch, P. Kosiuczenko, A. Lapadula, D. Latella, A. Lopes, M. Loreti, M. Massink, F. Mazzanti, U. Montanari, C. Oliveira, R. Pugliese, A. Tarlecki, M. W. Wemelinger, M. Wirsing, and A. Zawlocki, "AGILE: Software Architecture for Mobility," in M. Wirsing, D. Pattinson, and R. Hennicker (Eds.), *16th International Workshop on Recent Trends in Algebraic Development Techniques Lecture Notes in Computer Science 2755*, pp. 1-33, Springer Berlin / Heidelberg, 2003.

[99] A. Lopes and J. L. Fiadeiro, "Adding Mobility to Software Architectures," *Electronic Notes in Theoretical Computer Science,* 97, pp. 241-258, 2004.

[100] P. J. McCann and G.-C. Roman, "Modeling Mobile IP in Mobile UNITY," *ACM Transactions on Software Engineering and Methodology (TOSEM),* 8(2), pp. 115-146, 1999.

[101] S. D. Zilio, "Mobile Processes: A Commented Bibliography," in F. Cassez, C. Jard, B. Rozoy, and M. D. Ryan (Eds.), *4th Summer School on Modeling and Verification of Parallel Processes*, *Lecture Notes in Computer Science 2067*, pp. 206-222, Springer Berlin / Heidelberg, 2000.

[102] A. Phillips, S. Eisenbach, and D. Lister, "From Process Algebra to Java Code," in *ECOOP Workshop on Formal Techniques for Java-like Programs: FTfJP'2002*, 2002. Available from: http://www.cs.ru.nl/ftfjp/2002.html

[103] L. Cardelli, "Mobile Ambient Synchronization," Digital Equipment Corporation, Systems Research Centre, Palo Alto, Technical Report 1997-013, July 1997.

[104] S. Papastavrou, G. Samaras, and E. Pitoura, "Mobile Agents for WWW Distributed Database Access," *IEEE Transactions on Knowledge and Data Engineering,* 12(5), pp. 802-820, 2000.

[105] R. Gray, D. Kotz, G. Cybenko, and D. Rus, "Mobile Agents: Motivations and State-of-the-Art Systems," Dartmouth College 2000.

[106] C. Spyrou, G. Samaras, E. Pitoura, and P. Evripidou, "Mobile Agents for Wireless Computing: The Convergence of Wireless Computational Models with Mobile-Agent Technologies," *Mobile Networks and Applications,* 9(5), pp. 517-528, 2004.

[107] G. Cabri, L. Leonardi, and F. Zambonelli, "MARS: A Programmable Coordination Architecture for Mobile Agents," *IEEE Internet Computing,* 4(4), pp. 26-35, 2000.

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

[109] D. Chess, C. Harrison, and A. Kershenbaum, "Mobile Agents: Are They Good Enough," in J. Vitek and C. Tschudin (Eds.), *Second International workshop*

Mobile Object Systems Towards the Programmable Internet, *Lecture notes in Computer Science 1222*, pp. 25-45, Springer Berlin / Heidelberg, 1997.

[110] D. B. Lange, "Mobile Objects and Mobile Agents: The Future of Distributed Computing?," in E. Jul (Ed.), *12th European Conference Object-Oriented Programming: ECOOP'981445*, pp. 1-12, Springer Berlin / Heidelberg, 1998.

[111] R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson, and D. Rus, "D'Agents: Applications and Performance of a Mobile-Agent System," *Software: Practice and Experience,* 32(6), pp. 543-573, 2002.

[112] G. Fortino and W. Russo, "Multi-coordination of Mobile Agents: a Model and Component-based Architecture," in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pp. 443-450, ACM, 2005.

[113] M. Izatt, P. Chan, and T. Brecht, "Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications," *Concurrency: Practice and Experience,* 12(8), pp. 667-685, 2000.

[114] F. Bellifemine, A. Poggi, and G. Rimassa, "JADE: a FIPA2000 Compliant Agent Development Environment," in *Fifth International Conference on Autonomous Agents*, pp. 216-217, ACM, 2001.

[115] U. Pinsdorf and V. Roth, "Mobile Agent Interoperability Patterns and Practice," in *Ninth Annual IEEE International Conference and Workshop on* pp. 238-244, IEEE Computer Society, 2002.

[116] E. F. d. A. Lima, P. D. d. L. Machado, J. C. A. de Figueiredo, and F. R. Sampaio, "Implementing Mobile Agent Design Patterns in the JADE Framework," *TILAB Journal,* (EXP in Search of Innovation - Special Issue on JADE), 2003. Available from: http://jade.tilab.com/papers/EXP/Ferreira.pdf

[117] P. H. Welch, "Java Threads in the Light of occam/CSP," in P. H. Welch and A. W. P. Bakkers (Eds.), *WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, pp. 259-284, IOS Press, Amsterdam, 1998.

[118] C. Petitpierre, "Synchronous Active Objects Introduce CSP's Primitives in Java," in J. Pascoe, P. H. Welch, R. Loader, and V. Sunderam (Eds.), *Communicating Process Architectures 2002*, pp. 109-122, IOS Press, Amsterdam, 2002.

[119] C. Petitpierre, "A Development Method Boosted by Synchronous Active Objects," in J. F. Broenink and G. H. Hilderink (Eds.), *Communicating Process Architectures 2003*, pp. 17-32, IOS Press, Amsterdam, 2003.

[120] M. Schweigler, *A Unified Model for Inter- and Intra-Process Concurrency*. PhD Thesis, The University of Kent, Canterbury, 2006.

[121] 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.

[122] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec, and A. Zivkovic, "Object Serialization Analysis and Comparison in Java and .NET," *ACM SIGPLAN Notices,* 38(8), pp. 44-54, 2003.

[123] M. Phillippsen, B. Haumacher, and C. Nester, "More Efficient Serialization and RMI for Java," *Concurrency: Practice and Experience,* 12(7), pp. 495-518, 2000.

[124] Inmos Limited, "The T9000 Transputer Instruction Set Manual," SGS-Thompson Microelectronics 1993.

[125] 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.

[126] K. Chalmers and S. Clayton, "CSP for .NET Based on JCSP," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), *Communicating Process Architectures 2006*, pp. 59-76, IOS Press, Amsterdam, 2006.

[127] 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.

[128] 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.

[129] 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.

[130] 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.

[131] 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.

[132] K. Chalmers, J. Kerridge, and I. Romdhani, "Performance Evaluation of JCSP Micro Edition: JCSPme," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), *Communicating Process Architectures 2006*, pp. 31-40, IOS Press, Amsterdam, 2006.

[133] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey, "A Benchmark Suite for High Performance Java," *Concurrency: Practice and Experience,* 12(6), pp. 375-388, 2000.

[134] 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.

[135] K. Chalmers, J. Kerridge, and I. Romdhani, "A Critique of JCSP Networking," in P. H. Welch, S. Stepney, F. A. C. Polack, F. R. M. Barnes, A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson (Eds.), *Communicating Process Architectures 2008*, pp. 271-291, IOS Press, Amsterdam, 2008.

[136] A. Ripke, A. R. Allen, Y. Feng, and S. C. Allison, "Distributed Computing using Channel Communications and Java," in P. H. Welch and A. W. P. Bakkers (Eds.), *Communicating Process Architectures 2000*, pp. 49-62, IOS Press, Amsterdam, 2000.

[137] 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.

[138] P. H. Welch, "Graceful Termination - Graceful Resetting," in A. W. P. Bakkers (Ed.), *OUG-10: Applying Transputer Based Parallel Machines*, pp. 310-317, IOS Press, 1989.

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

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

[141] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, 2003.

[142] Formal Systems (Europe) Ltd., "FDR: User Manual and Tutorial, version 2.82," Formal Systems (Europe) Ltd., 2005.

[143] P. H. Welch and F. R. M. Barnes, "A CSP Model for Mobile Channels," in P. H. Welch, S. Stepney, F. A. C. Polack, F. R. M. Barnes, A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson (Eds.), *Communicating Process Architectures 2008*, pp. 17-33, IOS Press, Amsterdam, 2008.

[144] H. H. Happe, "TCP Input Threading in High Performance Distributed Systems," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), *Communicating Process Architectures 2006*, pp. 203-213, IOS Press, Amsterdam, 2006.

[145] H. Muller and D. May, "A Simple Protocol to Communicate Channels over Channels," in D. Pritchard and J. Reeve (Eds.), *Proceedings 4th International Euro-Par Conference: Euro-Par'98 Parallel Processing*, *Lecture Notes in Computer Science 1470*, pp. 591-600, Springer Berlin / Heidelberg, 1998.

[146] C. E. Perkins, "Mobile IP," *IEEE Communications Magazine,* 40(5), pp. 66-82, 2002.

[147] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Transactions on Software Engineering,* 13(1), pp. 23-31, 1987.

[148] J. Howell, "Straightforward Java Peristence Through Checkpointing," in R. Morrison, M. Jordan, and M. Atkinson (Eds.), *Proceedings of the 3rd International Workshop on Peristence and Java (PJW3): Advances in Peristent Object Systems*, pp. 322-334, Mogran Kaufmann Publishers, Inc., 1999.

[149] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, "Portable Support for Transparent Thread Migration in Java," in D. Kotz and F. Mattern (Eds.), *Agent Systems, Mobile Agents, and Applications*, *Lecture Notes in Computer Science 1882*, pp. 29-43, Springer Berlin / Heidelberg, 2000.

[150] D. Weyns, E. Truyen, and P. Verbaeten, "Serialization of Distributed Execution-state in Java," in M. Aksit, M. Mezini, and R. Unland (Eds.), *Objects, Components, Architectures, Services, and Applications for a Networked World: International Conference NetObjectDays, NODe 2002*,

Lecture Notes in Computer Science 2591, pp. 41-61, Springer Berlin / Heidelburg, 2003.

[151] W. Zhu, C.-L. Wang, W. Fang, and F. C. M. Lau, "A New Transparent Java Thread Migration System Using Just-In-Time Recompilation," in T. Gonzalez (Ed.), *The 16th IASTED International Conference on Parallel and Distributed Systems: PDCS 2004*, pp. 766-771, ACTA Press, 2004.

[152] S. Bouchenak and D. Hagimont, "Zero Overhead Java Thread Migration," Insitut National de Rechierche en Informatique et an Automatique 2002.

[153] S. Bouchenak, D. Hagimont, and N. De Palma, "Efficient Java Thread Serialization," in *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java*, *ACM International Conference Proceeding Series 42*, pp. 35-39, Computer Science Press, Inc., 2003.

[154] S. Bouchenak, D. Hagimont, S. Krakowiak, N. de Palma, and F. Boyer, "Experiences Implementing Efficient Java Thread Serialization, Mobility and Persistence," Instiut National de Recherche en Informatique et an Automatique 2002.

[155] T. Sakamoto, T. Sekigucki, and A. Yonezawa, "Bytecode Transformation for Portable Thread Migration in Java," in D. Kotz and F. Mattern (Eds.), *Agent Systems, Mobile Agents, and Applications*, *Lecture Notes in Computer Science 1882*, pp. 16-28, Springer Berlin / Heidelberg, 2000.

[156] R. K. K. Ma, C.-L. Wang, and F. C. M. Lau, "M-JavaMPI: A Java-MPI Binding with Process Migration Support," in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002*, pp. 255-232, IEEE Computer Society, 2002.

[157] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, and R. Jeffers, "Strong Mobility and Fine-Grained Resource Control in NOMADS," in D. Kotz and F. Mattern (Eds.), *Agent Systems, Mobile Agents, and Applications*, *Lecture Notes in Computer Science 1882*, pp. 2-15, Springer Berlin / Heidelberg, 2000.

[158] G. Fortino, W. Russo, and E. Zimeo, "A Statecharts-based Software Development Process for Mobile Agents," *Information and Software Technology,* 46(13), pp. 907-921, 2004.

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

[160] P. H. Welch, F. R. M. Barnes, and F. A. C. Polack, "Communicating Complex Systems," in *11th IEEE International Conference on Engineering of Complex Systems, 2006: ICECCS 2006*, pp. 107-120, IEEE Computer Society, 2006.

[161] 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.

[162] Sun Microsystems, "Java 2 SDK, Standard Edition Documentation," 2001. Available from: http://java.sun.com/j2se/1.3/docs/guide/

[163] Sun Microsystems, "Java Platform, Standard Edition 6 API Specification," 2006. Available from: http://java.sun.com/javase/6/docs/api/

Appendix A Serialization in Java

The information presented in this appendix is gathered from the Java SDK documentation [162] and Java API [163].

A.1 Serialization Components

Serialization in Java is provided by the use of a number of different interfaces and objects. The most important of these are the `Serializable` interface and the `ObjectInputStream` and `ObjectOutputStream` objects. The former is used to mark a class as being capable of serialization, and is inherited by all subclasses. Marking a class as serializable does not guarantee serialization, as there may be other objects within the sent object graph that are not serializable. In this situation, an exception will be thrown.

The object streams perform the encoding and decoding of object information. These objects are placed on other I/O streams that are responsible for the transfer / storage of the object information. Two methods are added to standard streams, `readObject` to read an object from an `ObjectInputStream`, and `writeObject` to write an object to an `ObjectOutputStream`. All sent classes and objects are recorded in a lookup table. If an object of an already sent class, or an already sent object, is written to the stream then a reference to the relevant class or object is sent instead of full details. As this lookup table can become large over time, it is possible to call reset on the output stream to clear this table, which also sends a signal to the input end. Another method to reduce the lookup table size is to use the `writeUnshared` method, which means that an object is always written as new on the stream, but the class definition is retained. Over time the lookup table will increase, but not significantly. The `writeUnshared` method was added in Java 1.4, and is therefore not supported by the JVM on the PDA.

The serialization of an object implementing `Serializable` is automatic unless the class implements specific methods. The `writeObject` method is called when an object is written to an `ObjectOutputStream`, and `readObject` is called when an object is read from an `ObjectInputStream`. This allows customisable serialization behaviour. Two other methods – `writeReplace` and `readResolve` – allow an object to be replaced by another when written to or read from the stream.

Serialization can also be controlled by classes implementing the `Externalizable` interface. `Externalizable` requires two methods to be implemented by the implementing class, `writeExternal` and `readExternal`. It is also the responsibility of the implementing class to coordinate with its super class to externalize its attributes, and also provide a no argument constructor for use by any sub classes which will also be externalizable. Externalization can perform better than serialization, due to the greater control provided to the developer, but it does require more development and care to implement.

A.2 Serialization Functionality

A number of control signals are used to control object serialization upon a stream. As a case study, the serialized representation of an Integer object is presented here, with the relevant control signals highlighted.

Aside from the serialization functionality proper, Java Reflection (the ability to interrogate an object to discover its properties and methods) is also used to gather the values within the object and the subsequent recreation of the object from its full name (e.g. `java.lang.Integer`). Reflection does have an overhead, but not on the data sent on the stream.

The control signals and flags used by Java serialization are available in the Java API documentation [163]. The values are repeated in Table 12 to allow easier presentation of the serialization operation.

Table 12: Serialization Control Signals and Flags

Signal	Type	Value
baseWireHandle	int	8257536
PROTOCOL_VERSION_1	int	1
PROTOCOL_VERSION_2	int	2
SC_BLOCK_DATA	byte	8
SC_ENUM	byte	16
SC_EXTERNALIZABLE	byte	4
SC_SERIALIZABLE	byte	2
SC_WRITE_METHOD	byte	1
STREAM_MAGIC	short	-21267
STREAM_VERSION	short	5
TC_ARRAY	byte	117
TC_BASE	byte	112
TC_BLOCKDATA	byte	119
TC_BLOCKDATALONG	byte	122
TC_CLASS	byte	118
TC_CLASSDESC	byte	114
TC_ENDBLOCKDATA	byte	120
TC_ENUM	byte	126
TC_EXCEPTION	byte	123
TC_LONGSTRING	byte	124
TC_MAX	byte	126
TC_NULL	byte	112
TC_OBJECT	byte	115
TC_PROXYCLASSDESC	byte	125
TC_REFERENCE	byte	113
TC_RESET	byte	121
TC_STRING	byte	116

When a new `ObjectOutputStream` is created, an initial handshake message is sent to allow correct behaviour at the receiving end. This message consists of two 16-bit values, `STREAM_MAGIC` and `STREAM_VERSION`. These signals are only sent once, and are therefore not considered part of a sent object.

When reset is called on an `ObjectOutputStream` a signal is sent to the complement `ObjectInputStream` to inform it of the reset. This is the `TC_RESET` signal. Again, this is not considered as part of normal serialized data.

Figure 71 presents the serialized form of an `Integer` object. Red signifies a control signal, yellow a string (prefixed with a two byte length header in white), and green signifies values defining the class. The data part of the object is given in blue.

	0	1	2	3	4	5	6	7	8	9
0	TC_OBJECT	TC_CLASSDESC	Name length (17)		j	a	v	a	.	l
10	a	n	g	.	I	n	t	e	g	e
20	r	Class Serialization Identifier (1360826667806853064)								Flags
30	Variable count (1)		I(nteger)	Name length (5)		v	a	l	u	e
40	TC_ENDBLOCKDATA	TC_CLASSDESC	Name length (16)		j	a	v	a	.	l
50	a	n	g	.	N	u	m	b	e	r
60	Class Serialization Identifier (-8742448824652078987)								Flags	Variable count (0)
70		TC_ENDBLOCKDATA	TC_BASE	value						

Figure 71: Serialized Integer Object

The first value is `TC_OBJECT`, which is used to represent the type of object being sent. The other possible data types are `TC_ARRAY` (array), `TC_CLASS` (class), `TC_ENUM` (enum constant), `TC_LONGSTRING` (long string), `TC_NULL` (null value), `TC_REFERENCE` (reference to previous item on stream), and `TC_STRING` (a string). After the `TC_OBJECT` signal, a `TC_CLASSDESC` is sent to indicate the start of a new class description. If the class has previously been used on the stream, `TC_REFERENCE` is used instead.

The name of the class (bytes 4 to 20) is sent next, with a 16-bit string length header (2 - 3). An Integer object has the full name `java.lang.Integer`. Bytes 21 to 28 represent the 64-bit serialization identifier for the class, and this is used to ensure the correct class type is being used by each end of the stream. A byte representing flag values is then sent to allow correct interpretation of the serialization process. The flag values are `SC_WRITE_METHOD`, `SC_SERIALIZABLE`, `SC_EXTERNALIZABLE`, `SC_BLOCKDATA` and `SC_ENUM`. These indicate whether the class uses a `writeObject` method, uses normal serialization, uses externalization, uses block data externalization, or is an enum type.

The number of internal fields within the class is then sent as a 16-bit value (30 – 31). Integer only has a single internal field, the value of the primitive int. The types of all the internal fields are sent, which may involve further class descriptions, thus starting the description process for these classes. For the primitive `int` in the Integer object, the type is represented by the letter I (ASCII 73). The name of the field is then sent as a string (5 – 9) with a 16-bit length header. The name of the `int` field is value in the `Integer` object. A control signal (`TC_ENDBLOCKDATA`) is then sent to indicate the end of the class description.

If the class has a parent, then the description of this class is also sent on the stream using the aforementioned method. This is required as the parent class may declare fields of its own not visible to the child object, but are still used by methods from the parent class. `Integer` extends `Number` (`java.lang.Number`) which has no declared fields. Finally, a signal is sent to indicate that the field values are to be sent (`TC_BASE`) followed by the field values. For the `Integer` object, this is a four byte value representing the `int`.

Serialized representations of commonly used objects are provided here with no description. These objects are byte array, Integer array, the JCSP data message, the acknowledge message, and the various test classes.

Each data type in Java has a signature letter to distinguish it within a serialization stream. These are provided in Table 13 to allow clearer understanding of the serialization data presented.

Table 13: Java Data Type Signatures

Data Type	Signature
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
Object	L
Object name	L<*name*>;
Array	[<*type*>

A.3 Byte Array

	0	1	2	3	4	5	6	7	8	9
0	TC_ARRAY	TC_CLASSDESC	Name length (2)		[	B	Class Serialization Identifier			
10					Flags	Variable count (0)	TC_ENDBLOCKDATA	TC_BASE	Length	
20				Elements						

Figure 72: Serialized Byte Array

A.4 ChannelMessage.Data

	0	1	2	3	4	5	6	7	8	9
0	TC_OBJECT	TC_CLASSDESC	Name length (32)		o	r	g	.	j	c
10	s	p	.	n	e	t	.	C	h	a
20	n	n	e	l	M	e	s	s	a	g
30	e	$	D	a	t	a	Class Serialization Identifier			
40					Flags	Variable count (2)		Z (boolean)	Name length (12)	
50	a	c	k	n	o	w	l	e	d	g
60	e	d	L (Object)	Name length (4)		d	a	t	a	TC_STRING
70	Name length (18)		L	j	a	v	a	/	l	a
80	n	g	/	O	b	j	e	c	t	;
90	TC_ENDBLOCKDATA	TC_CLASSDESC	Name length (27)		o	r	g	.	j	c
100	s	p	.	n	e	t	.	C	h	a
110	n	n	e	l	M	e	s	s	a	g
120	e	Class Serialization Identifier								Flags
130	Variable count (0)		TC_ENDBLOCKDATA	TC_CLASSDESC	Name length (20)		o	r	g	.
140	j	c	s	p	.	n	e	t	.	M
150	e	s	s	a	g	e	Class Serialization Identifier			
160					Flags	Variable count (3)		J (long)	Name length (9)	
170	d	e	s	t	l	n	d	e	x	J (long)
180	Name length (11)		s	o	u	r	c	e	l	n
190	d	e	x	L (Object)	Name length (12)		d	e	s	t
200	V	C	N	L	a	b	e	l	TC_STRING	Name length (18)
210		L	j	a	v	a	/	l	a	n
220	g	/	S	t	r	i	n	g	;	TC_ENDBLOCKDATA
230	TC_BASE	destIndex								sourceIndex
240								destVCNLabel	acknowledged	data

Figure 73: Serialized ChannelMessage.Data

A.5 ChannelMessage.Ack

	0	1	2	3	4	5	6	7	8	9
0	TC_OBJECT	TC_CLASSDESC	Name length (31)		o	r	g	.	j	c
10	s	p	.	n	e	t	.	C	h	a
20	n	n	e	l	M	e	s	s	a	g
30	e	$	A	c	k	Class Serialization Identifier				
40				Flags	Variable count (0)		TC_ENDBLOCKDATA	TC_CLASSDE	Name length (27)	
50	o	r	g	.	j	c	s	p	.	n
60	e	t	.	C	h	a	n	n	e	l
70	M	e	s	s	a	g	e	Class Serialization Identifier		
80						Flags	Variable count (0)		TC_ENDBLOCKDATA	TC_CLASSDESC
90	Name length (20)		o	r	g	.	j	c	s	p
100	.	n	e	t	.	M	e	s	s	a
110	g	e	Class Serialization Identifier							
120	Flags	Variable Count (3)		J (Long)	Name length (9)		d	e	s	t
130	l	n	d	e	x	J (Long)	Name length (11)		s	o
140	u	r	c	e	l	n	d	e	x	L (Object)
150	Name length (12)		d	e	s	t	V	C	N	L
160	a	b	e	l	TC_STRING	Name length (18)		L	j	a
170	v	a	/	l	a	n	g	/	S	t
180	r	i	n	g	;	TC_ENDBLOCKDATA	TC_BASE	destIndex		
190						sourceIndex				
200				destVCNLabel						

Figure 74: Serialized ChannelMessage.Ack

A.6 Integer Array

	0	1	2	3	4	5	6	7	8	9
0	TC_ARRAY	TC_CLASSDESC	Name length (20)		[	L	j	a	v	a
10	.	l	a	n	g	.	l	n	t	e
20	g	e	r	;	Class Serialization Identifier					
30			Flags	Variable count (0)		TC_ENDBLOCKDATA	TC_BASE	Length		
40		Elements								

Figure 75: Serialized Integer Array

A.7 TestObject

	0	1	2	3	4	5	6	7	8	9
0	TC_OBJECT	TC_CLASSDESC	Name length (10)		T	e	s	t	O	b
10	j	e	c	t	Class Serialization Identifier					
20	?	u	Flags	Variable Count (2)		[(array)	Name length (4)		d	b
30	l	s	TC_STRING	Name length (19)		[	L	j	a	v
40	a	/	l	a	n	g	/	D	o	u
50	b	l	e	;	[(array)	Name length (4)		i	n	t
60	s	TC_STRING	Name length (20)		[	L	j	a	v	a
70	/	l	a	n	g	/	l	n	t	e
80	g	e	r	;	TC_ENDBLO	TC_BASE	TC_ARRAY	TC_CLASSDESC	Name length (19)	
90	[	L	j	a	v	a	.	l	a	n
100	g	.	D	o	u	b	l	e	;	Class
110	Serialization Identifier							Flags	Variable Count (0)	
120	TC_ENDBLO	TC_BASE	Length				TC_ARRAY	TC_CLASSDESC	Name length (20)	
130	[	L	j	a	v	a	.	l	a	n
140	g	.	l	n	t	e	g	e	r	;
150	Class Serialization Identifier								Flags	Variable
160	Count (0)	TC_ENDBLOCKDATA	TC_BASE	Length						

Figure 76: Serialized TestObject

A.8 TestObject2 and TestObject3

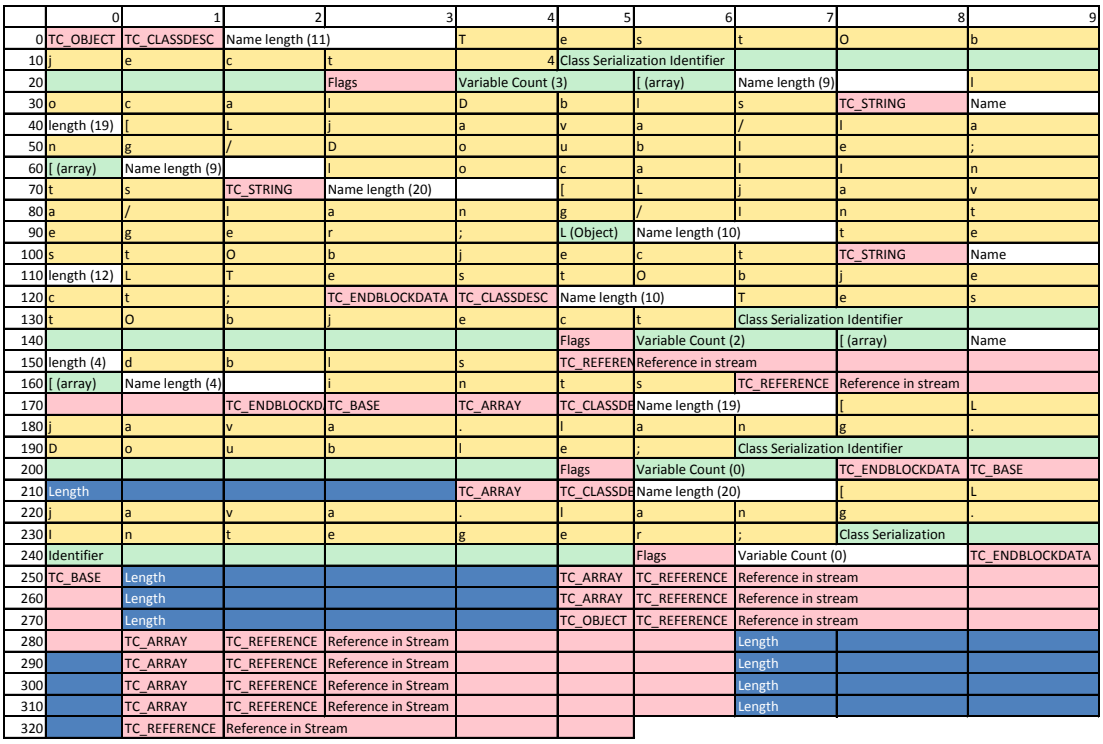
TestObject2 and TestObject3 are identical in description except for their name. The difference is only in byte 14, which will be 2 or 3 respectively.

	0	1	2	3	4	5	6	7	8	9
0	TC_OBJECT	TC_CLASSDESC	Name length (11)		T	e	s	t	O	b
10	j	e	c	t	2	Class Serialization Identifier				
20				Flags	Variable Count (2)		[(array)	Name length (9)		l
30	o	c	a	l	D	b	l	s	TC_STRING	Name
40	length (19)	[	L	j	a	v	a	/	l	a
50	n	g	/	D	o	u	b	l	e	;
60	[(array)	Name length (9)		l	o	c	a	l	l	n
70	t	s	TC_STRING	Name length (20)		[	L	j	a	v
80	a	/	l	a	n	g	/	l	n	t
90	e	g	e	r	;	TC_ENDBLOCKDATA	TC_CLASSDESC	Name length (10)		T
100	e	s	t	O	b	j	e	c	t	Class
110	Serialization Identifier							Flags	Variable Count (2)	
120	[(array)	Name length (4)		d	b	l	s	TC_REFERENCE	Reference in Stream	
130			[(array)	Name length (4)		i	n	t	s	TC_REFERENCE
140	Reference in Stream				TC_ENDBLOCKDATA	TC_BASE	TC_ARRAY	TC_CLASSDESC	Name length (19)	
150	[	L	j	a	v	a	.	l	a	n
160	g	.	D	o	u	b	l	e	;	Class
170	Serialization Identifier							Flags	Variable Count (0)	
180	TC_ENDBLO	TC_BASE	Length				TC_ARRAY	TC_CLASSDESC	Name length (20)	
190	[	L	j	a	v	a	.	l	a	n
200	g	.	l	n	t	e	g	e	r	;
210	Class Serialization Identifier								Flags	Variable
220	Count (0)	TC_ENDBLOCKDATA	TC_BASE	Length				TC_ARRAY	TC_REFERENCE	Reference in
230	Stream			Length				TC_ARRAY	TC_REFERENCE	Reference in
240	Stream			Length						

Figure 77: Serialized TestObject2 and TestObject3

A.9 TestObject4 and TestObject5

TestObject4 and TestObject5 are identical in description except for their name. The difference is only in byte 14, which will be 4 or 5 respectively.



	0	1	2	3	4	5	6	7	8	9
0	TC_OBJECT	TC_CLASSDESC	Name length (11)		T	e	s	t	O	b
10	j	e	c	t	4	Class Serialization Identifier				
20				Flags	Variable Count (3)		[(array)	Name length (9)		l
30	o	c	a	l	D	b	l	s	TC_STRING	Name
40	length (19)	[	L	j	a	v	a	/	l	a
50	n	g	/	D	o	u	b	l	e	;
60	[(array)	Name length (9)		l	o	c	a	l	l	n
70	t	s	TC_STRING	Name length (20)		[	L	j	a	v
80	a	/	l	a	n	g	/	l	n	t
90	e	g	e	r	;	L (Object)	Name length (10)		t	e
100	s	t	O	b	j	e	c	t	TC_STRING	Name
110	length (12)	L	T	e	s	t	O	b	j	e
120	c	t	;	TC_ENDBLOCKDATA	TC_CLASSDESC	Name length (10)		T	e	s
130	t	O	b	j	e	c	t	Class Serialization Identifier		
140						Flags	Variable Count (2)		[(array)	Name
150	length (4)	d	b	l	s	TC_REFEREN	Reference in stream			
160	[(array)	Name length (4)		i	n	t	s	TC_REFERENCE	Reference in stream	
170			TC_ENDBLOCKD	TC_BASE	TC_ARRAY	TC_CLASSDE	Name length (19)		[	L
180	j	a	v	a	.	l	a	n	g	.
190	D	o	u	b	l	e	;	Class Serialization Identifier		
200						Flags	Variable Count (0)		TC_ENDBLOCKDATA	TC_BASE
210	Length				TC_ARRAY	TC_CLASSDE	Name length (20)		[	L
220	j	a	v	a	.	l	a	n	g	.
230	l	n	t	e	g	e	r	;	Class Serialization	
240	Identifier						Flags	Variable Count (0)		TC_ENDBLOCKDATA
250	TC_BASE	Length				TC_ARRAY	TC_REFERENCE	Reference in stream		
260		Length				TC_ARRAY	TC_REFERENCE	Reference in stream		
270		Length				TC_OBJECT	TC_REFERENCE	Reference in stream		
280		TC_ARRAY	TC_REFERENCE	Reference in Stream				Length		
290		TC_ARRAY	TC_REFERENCE	Reference in Stream				Length		
300		TC_ARRAY	TC_REFERENCE	Reference in Stream				Length		
310		TC_ARRAY	TC_REFERENCE	Reference in Stream				Length		
320		TC_REFERENCE	Reference in Stream							

Figure 78: Serialized TestObject4 and TestObject5

Appendix B Test Object Class Definitions

B.1 TestObject

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

    public TestObject()
    {
    }

    public TestObject(int size)
    {
        ints = new Integer[size];
        dbls = new Double[size];
        for (int i = 0; i < size; i++)
        {
            ints[i] = new Integer(i);
            dbls[i] = new Double(i * 1000);
        }
    }

    public static TestObject create(int size)
    {
        return new TestObject(size);
    }
}
```

B.2 TestObject2

```
public class TestObject2 extends TestObject
{
    private Integer[] localInts;
    private Double[] localDbls;

    public TestObject2()
    {
    }

    public TestObject2(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] = new Integer(i);
            dbls[i] = new Double(i * 1000);
            localInts[i] = new Integer(i * 1000000);
            localDbls[i] = new Double(i * 1000000000);
        }
    }
```

```
    public static TestObject create(int size)
    {
        return new TestObject2(size);
    }
}
```

B.3 TestObject3

```
public class TestObject3 extends TestObject
{
    private Integer[] localInts;
    private Double[] localDbls;

    public TestObject3()
    {
    }

    public TestObject3(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 * 1000);
        }
    }

    public static TestObject create(int size)
    {
        return new TestObject3(size);
    }
}
```

B.4 TestObject4

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

    public TestObject4()
    {
    }

    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 * 1000);
        }
    }


    public void setTest(TestObject testObject)
    {
        this.testObject = testObject;
    }
```

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

B.5 TestObject5

```
public class TestObject5 extends TestObject
{
    private TestObject testObject;
    private Integer[] localInts;
    private Double[] localDbls;

    public TestObject5()
    {
    }

    public TestObject5(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 * 1000);
        }
    }

    public TestObject5(TestObject5 testObject)
    {
        int size = testObject.ints.length;
        ints = new Integer[size];
        dbls = new Double[size];
        localInts = new Integer[size];
        localDbls = new Double[size];
        this.testObject = testObject;
        for (int i = 0; i < size; i++)
        {
            ints[i] = localInts[i] = testObject.ints[i];
            dbls[i] = localDbls[i] = testObject.dbls[i];
        }
    }

    public void setTest(TestObject testObject)
    {
        this.testObject = testObject;
    }

    public static TestObject create(int size)
    {
        TestObject5 tObj1 = new TestObject5(size);
        TestObject5 tObj2 = new TestObject5(tObj1);
        tObj1.setTest(tObj2);
        return tObj1;
    }
}
```

Appendix C Performance Characterisation Data

C.1 Java Grande Benchmark Arithmetic Operations

Figure 79 presents the results from the Java Grande Benchmark Suite arithmetic tests.

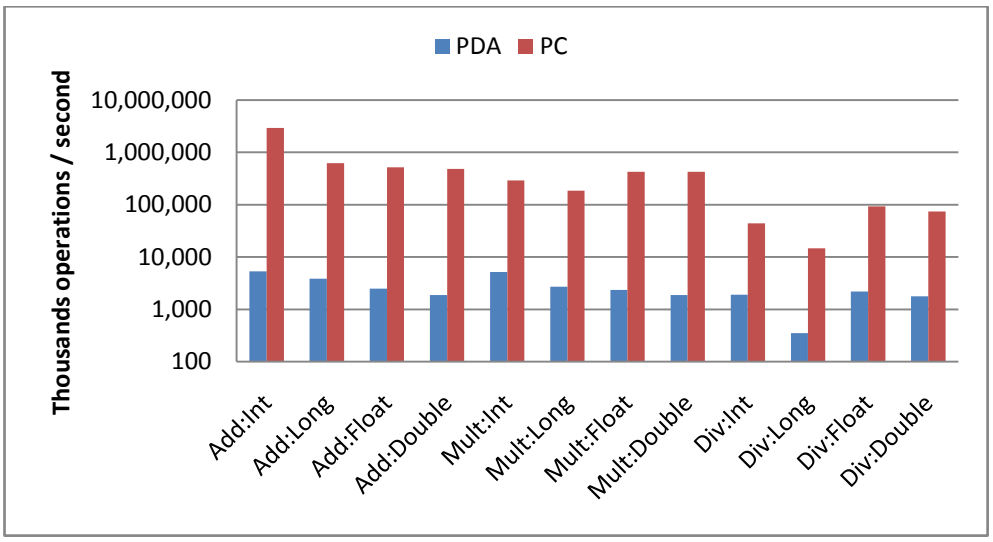


Figure 79: Arithmetic Benchmark Results

The results presented are in operations per second across the various arithmetic operations and across the four primitive numeric types of Java. In general, the ratio between the results for the PDA and the PC ranges between 1.5 (division operations) and 2.5 (addition operations) orders of magnitude.

C.2 Object Creation Time

Figure 80 presents the results from the Java Grande Benchmark Suite object creation tests. The different object creation methods are:

- *Base* – the base Java Object.
- *Simple* – simple object.
- *Simple:Constructor* – simple object with a defined constructor.

- *Simple:1Field* – simple object with one field.
- *Simple:2Field* – simple object with two fields.
- *Simple:4Field* – simple object with four fields.
- *Simple:4fField* – simple object with four float fields.
- *Simple:4LField* – simple object with four long fields.
- *Subclass* – an object that extends another object.
- *Complex* – an object that contains another object.
- *Complex:Constructor* – an object that contains another object using a constructor.

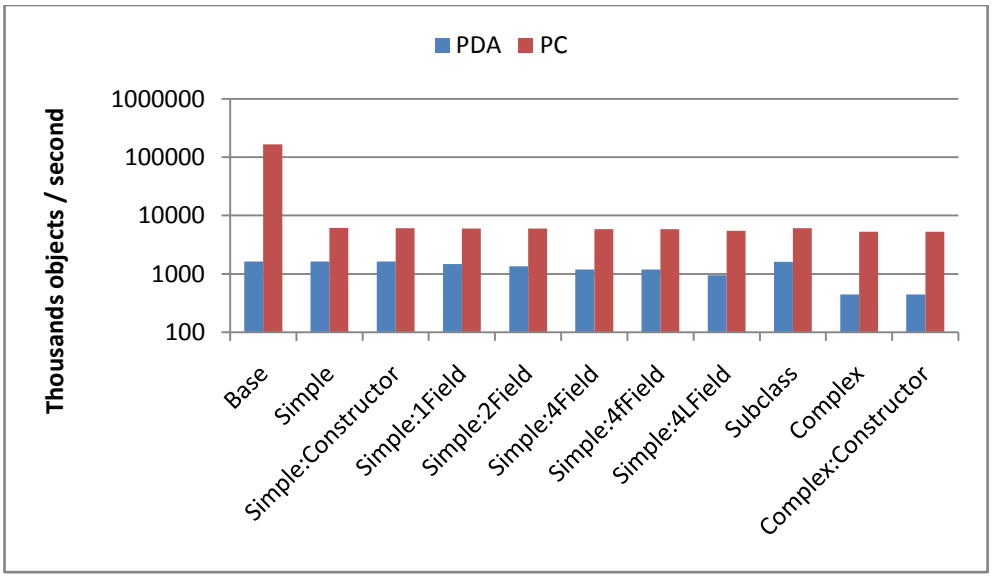


Figure 80: Object Creation Benchmark Results

Aside from the time taken to create a base Java object and a complex object, the difference in creating objects between the PDA and PC is just below one order of magnitude. The PDA takes roughly the same time to allocate an object without consideration for internal fields. This accounts for the time taken to create a complex object.

C.3 Array Creation Time

Figure 81 presents the results from creating various array types of various sizes. The three primitive numeric types int, long, and float are provided. The object array is an array of the base Java Object. The time taken to allocate the object array does

not incorporate object creation time, therefore each element in the array will be set to null.

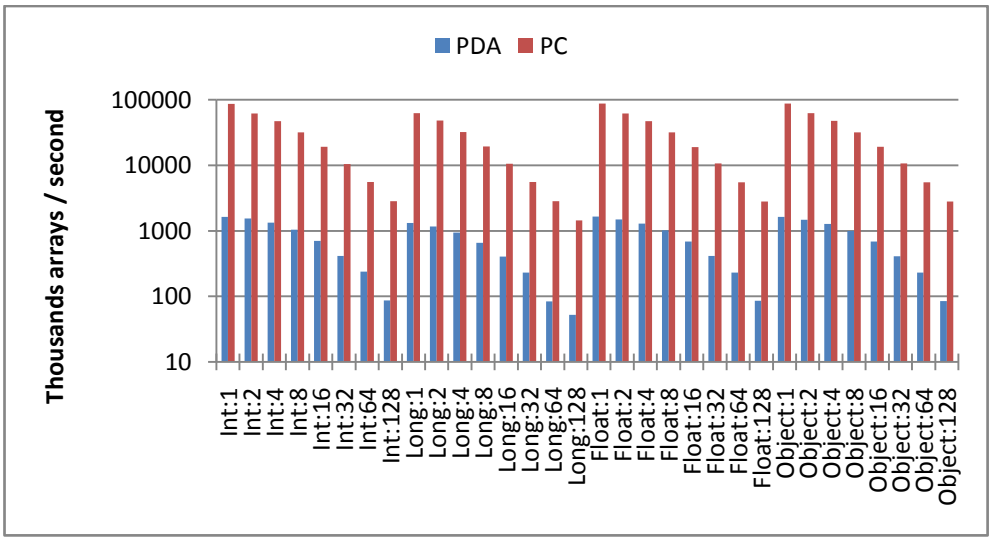


Figure 81: Array Creation Benchmark Results

The results show similar patterns within the PDA and the PC for allocating the arrays, although the PC has close to linear performance decline, whereas the PDA has exponential decline. The difference in performance is about 1.5 orders of magnitude between the PDA and PC.

C.4 Serialization

Figure 82 presents the results from the Java Grande Suite serialization tests. In these tests, various objects are serialized into a file and the throughput in bytes per second recorded. Therefore, these results are not only the time taken to convert the object into bytes via serialization, but also the time taken to write said bytes to file.

The PC generally has serialization throughput 2.5 orders of magnitude greater than that of the PDA. Part of this throughput difference will be due to the I/O time for writing the bytes to the file.

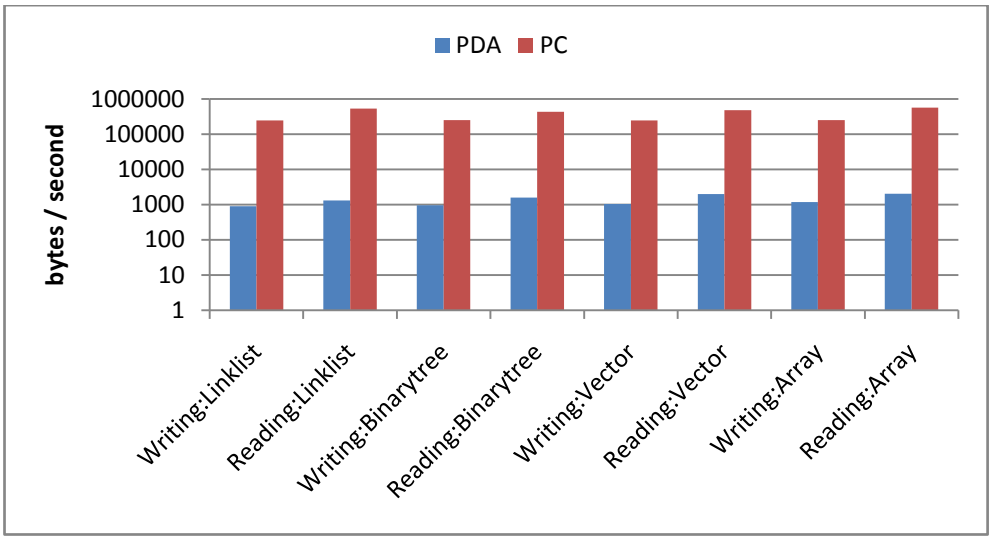


Figure 82: Serialization Benchmark Results

C.5 Multithreaded Benchmarks

C.5.1 Fork / Join Time

Figure 83 presents the results from the Java Grande Benchmark Suite fork / join test. The fork / join test measures the time taken to fork the number of threads, and subsequently join the threads after they have all been forked. Thus thread creation and subsequent destruction is captured. The run method of each thread is trivial, and the benchmark removes this time from the results.

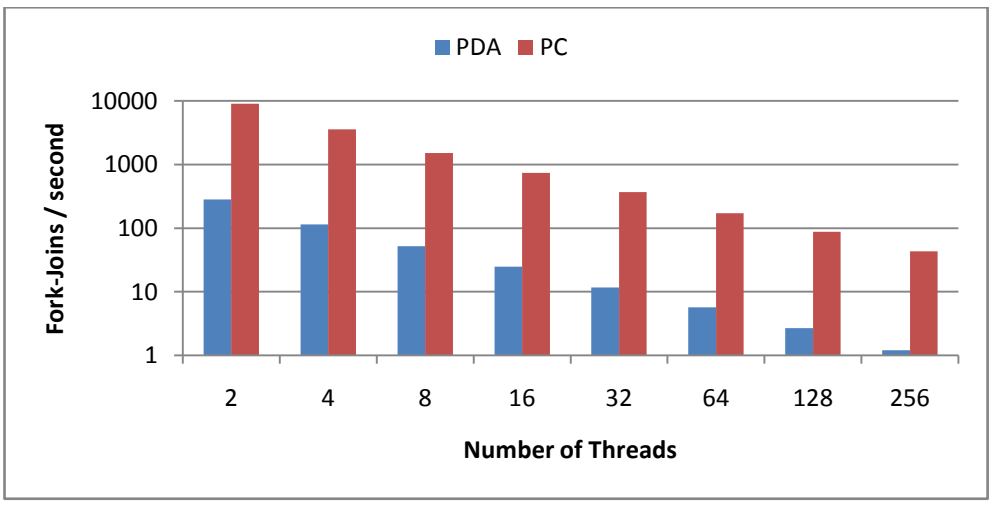


Figure 83: Fork / Join Benchmark Results

These results show a difference in thread fork / join operations of 1.5 orders of magnitude between the PDA and the PC. The decline is linear on the logarithmic

scale used in the figure, which is as expected. The decline is therefore exponential based on the number of threads increasing by a factor of two per test.

C.5.2 Thread Synchronisation Time

Figure 84 presents the results from the thread synchronisation tests within the Java Grande Benchmark Suite. In these tests, the relative threads compete for access to a shared data object, either by using a synchronised method, or by using an object as a monitor lock.

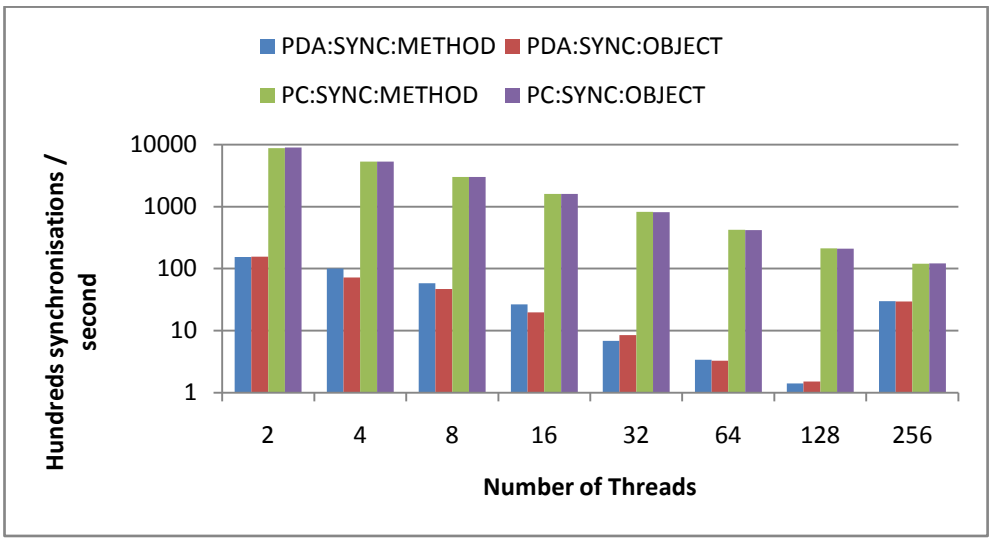


Figure 84: Synchronisation Benchmark Results

As these results show, the PC performs these operations approximately two orders of magnitude faster than the PDA. However, the result for 256 threads is considerably off scale in comparison to the others. This feature is deemed to be an issue of the PDA trying to deal with too many threads competing for the shared resource, and is considered a false result.

C.6 JCSP Specific Test Results

C.6.1 CommsTime

Figure 85 presents the results for the standard CommsTime test in JCSP. The results represent the time – in microseconds – it takes for a Prefix, Parallel Delta and Successor process to produce a number. These processes are connected together such that the sequence of natural numbers is produced. The actual code definitions for these processes are available in Appendix H.

These processes can be used to test the communication time between two processes via a channel. The figures presented represent the communication time over four channels, and thus the communication time per channel is this figure divided by four. This can also lead to an approximation of the context switch time by further dividing this figure by two. This does not take into consideration any time it might take for the internal processes of the Parallel Delta being started and subsequently completed, which incurs an internal synchronisation.

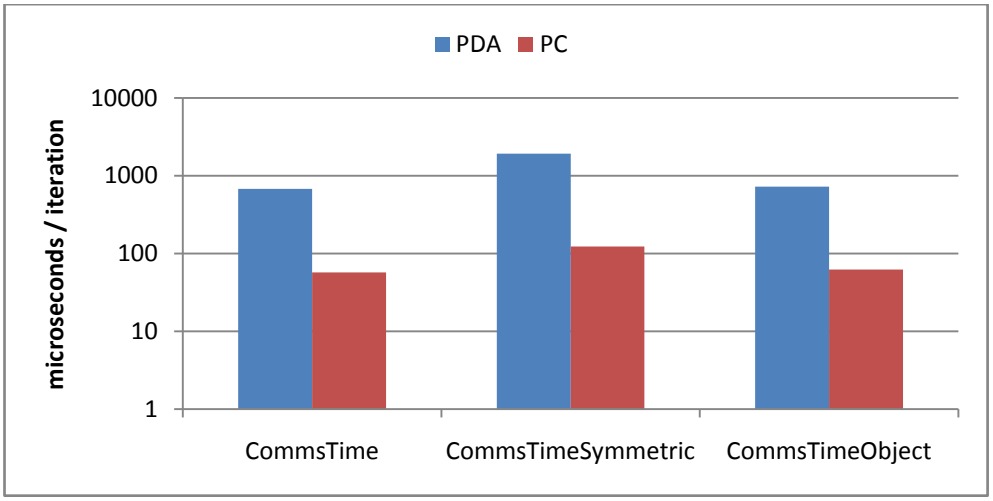


Figure 85: CommsTime Benchmark Results

Three separate CommsTime tests are presented:

- *CommsTime* – is the CommsTime test performed using channels that communicate primitive int data.
- *CommsTimeSymmetric* – is the CommsTime test performed using channels that communicate primitive int data and also have guarded output. This is incorporated using an internal AltingBarrier within the channel.
- *CommsTimeObject* – is the CommsTime test performed using Integer objects rather than primitive integers. This incurs an overhead due to boxing and unboxing the value to allow arithmetic operations upon the internal primitive integer of the Integer object.

As these results show, there is a difference of one order of magnitude between the PC and the PDA. Relatively speaking, there is little difference between

communicating an object and communicating a primitive integer using a channel. There is an overhead incurred by using an AltingBarrier within a channel however.

C.6.2 Stressed Alternative

Figure 86 presents results from various configurations of the Stress Alternative test within the PDA and the PC. The configurations on the x-axis are the number of channels and the number of processes per channel. Thus, 5 x 10 implies 5 channels connected to the reader, with each having 10 processes, for a total of 50 processes. The value presented is the time taken to select a ready channel fairly in an Alternative.

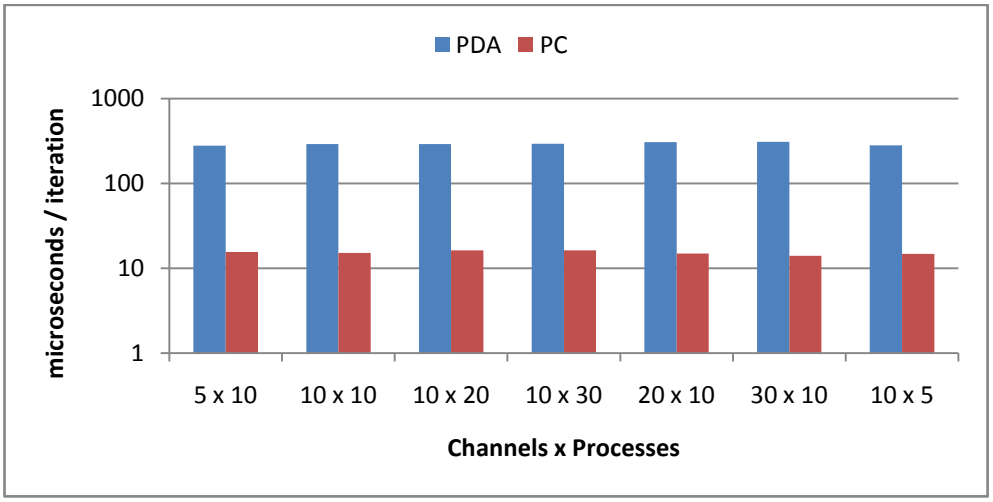


Figure 86: Stressed Alt Benchmark Results

The time taken to select a ready channel does not vary on the channel / process configurations presented. In general, the PC performs 1.25 orders of magnitude better than the PDA in these tests.

Appendix D Experimentation Results

D.1 Serialization of Test Objects

D.1.1 Java Grande Serialization Benchmarks

Figure 87 through Figure 90 present the times taken to serialize and deserialize Integer object arrays of size 0 to 100, and the different test object types also ranging in size from 0 to 100. These results are the Java Grande benchmark results, and thus include file I/O time. The results represent the average time taken to serialize or deserialize an object in milliseconds.

The phenomena in the PC serialization results (Figure 87), where TestObject2 and TestObject4 increase together and TestObject, TestObject3, and TestObject4 increase together is directly attributable to the complexities of the different objects (see Table 2). The number of unique objects for both TestObject2 and TestObject4 increase by a factor of 4 relative to the size of the arrays, whereas the others only increase by a factor of 2.

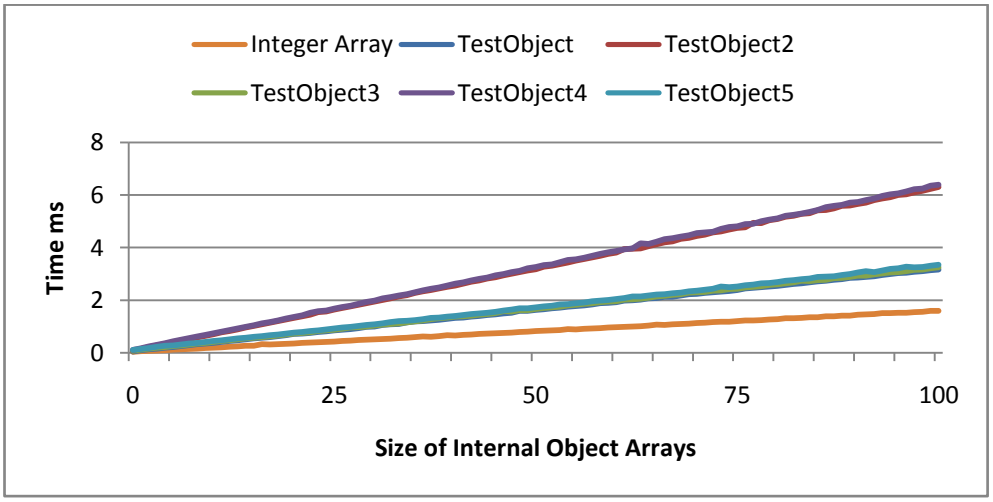


Figure 87: PC Java Grande Serialization Time

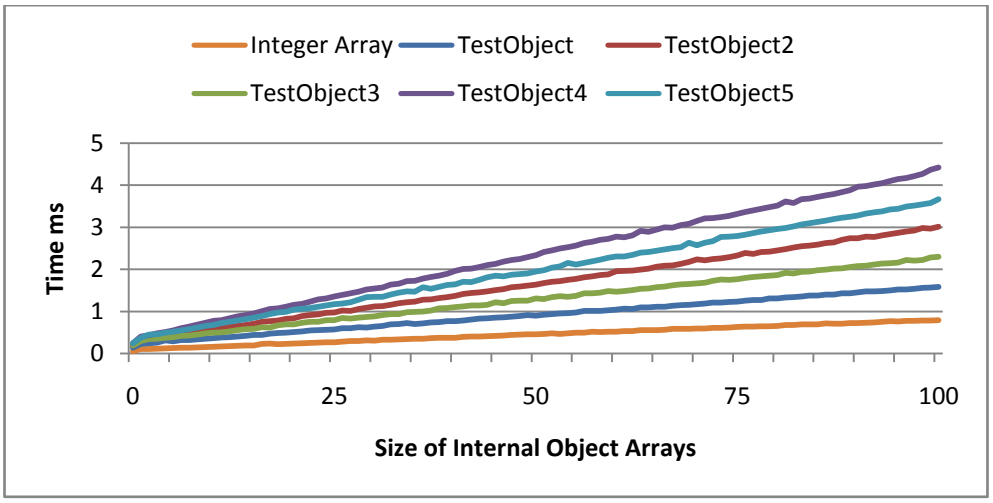


Figure 88: PC Java Grande Deserialization Time

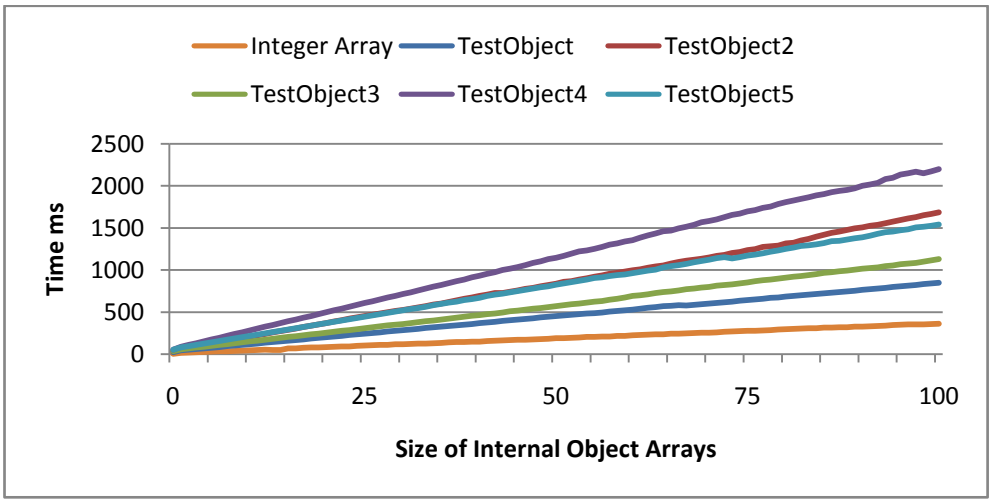


Figure 89: PDA Java Grande Serialization Time

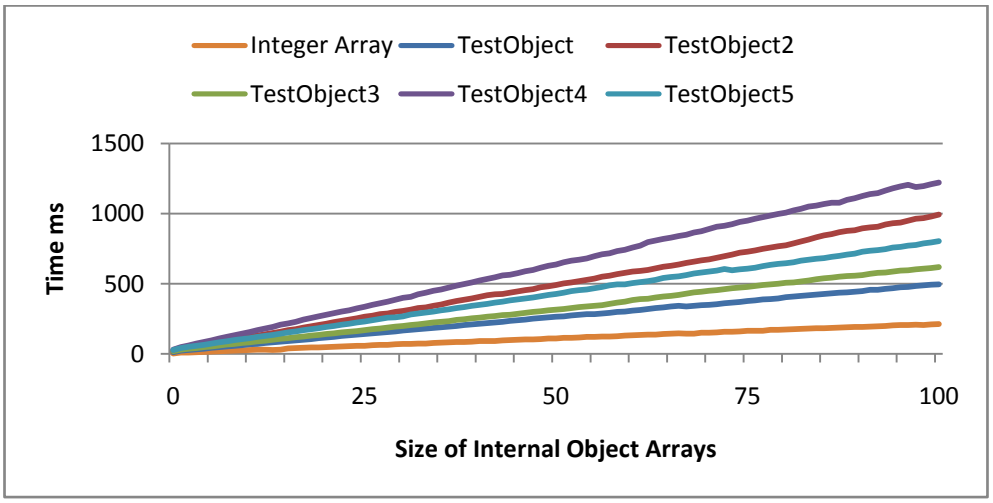


Figure 90: PDA Java Grande Deserialization Time

D.1.2 Serialization into Memory

Figure 91 to Figure 94 presents the results from serializing and deserializing the test objects in memory. The results presented are the average time in milliseconds to perform a single operation on the relevant object type and size.

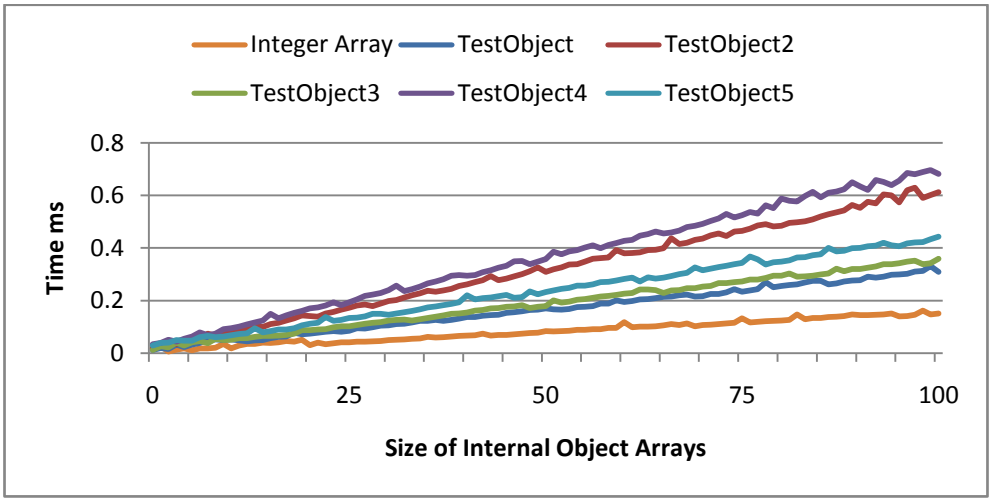


Figure 91: PC Memory Serialization Time

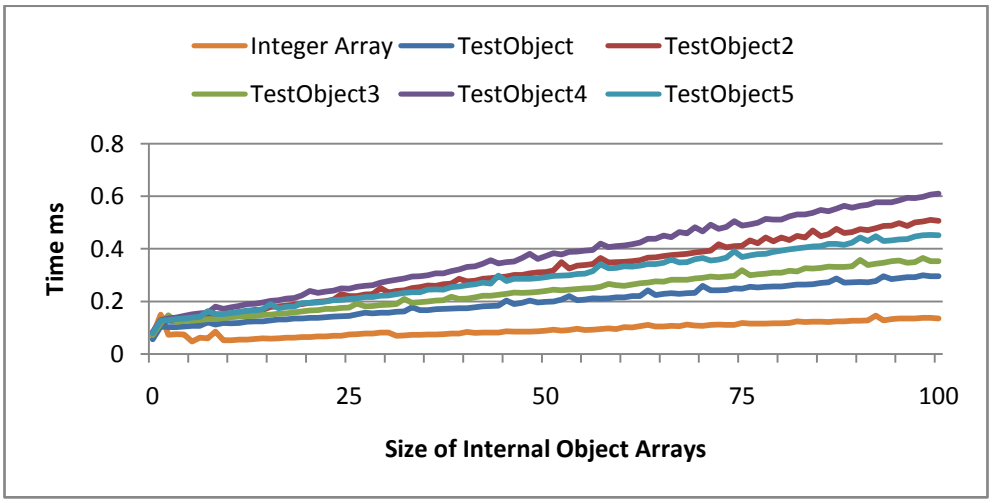


Figure 92: PC Memory Deserialization Time

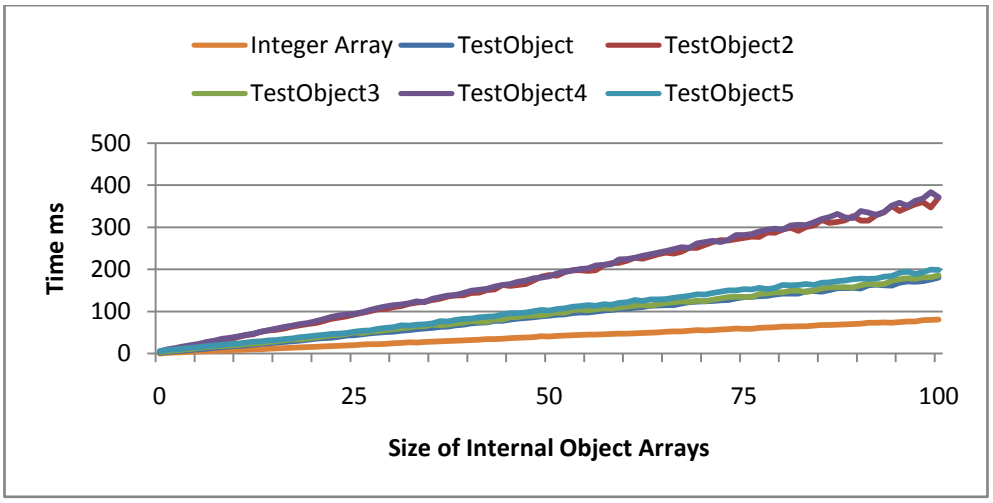


Figure 93: PDA Memory Serialization Time

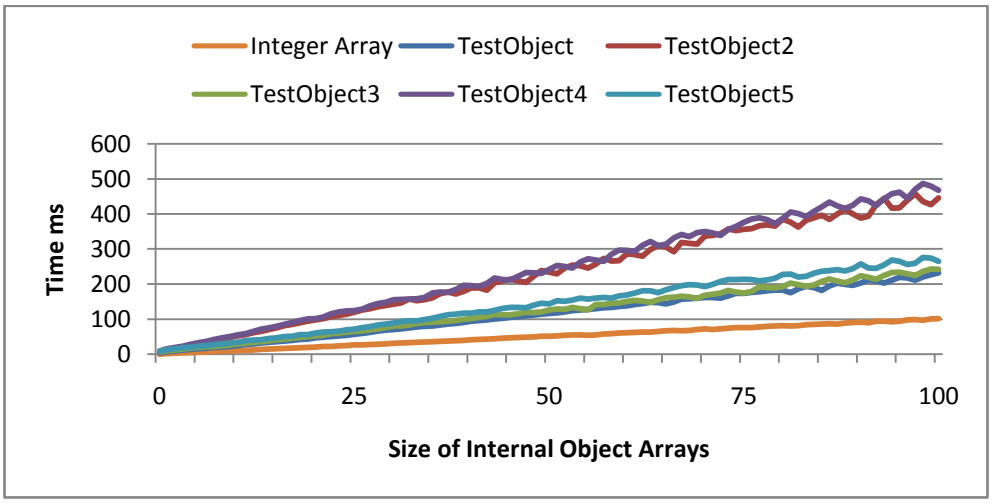


Figure 94: PDA Memory Deserialization Time

D.2 Network Performance

D.2.1 Send and Receive

Figure 95 presents the times recorded for the PC sending large data packets via the different communication mechanisms, while Figure 96 presents the times recorded for the PC to receive large data packets via the different communication mechanisms. Similarly, Figure 97 presents the recorded times for the PDA to send large data packets and Figure 98 presents the times for the PDA to receive large packet sizes. All times are in milliseconds.

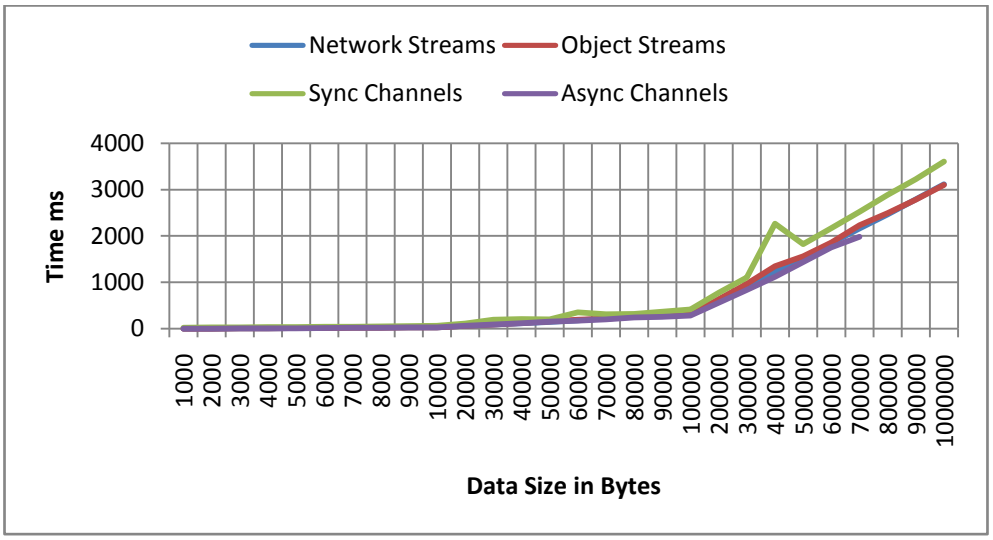


Figure 95: PC Sending Data

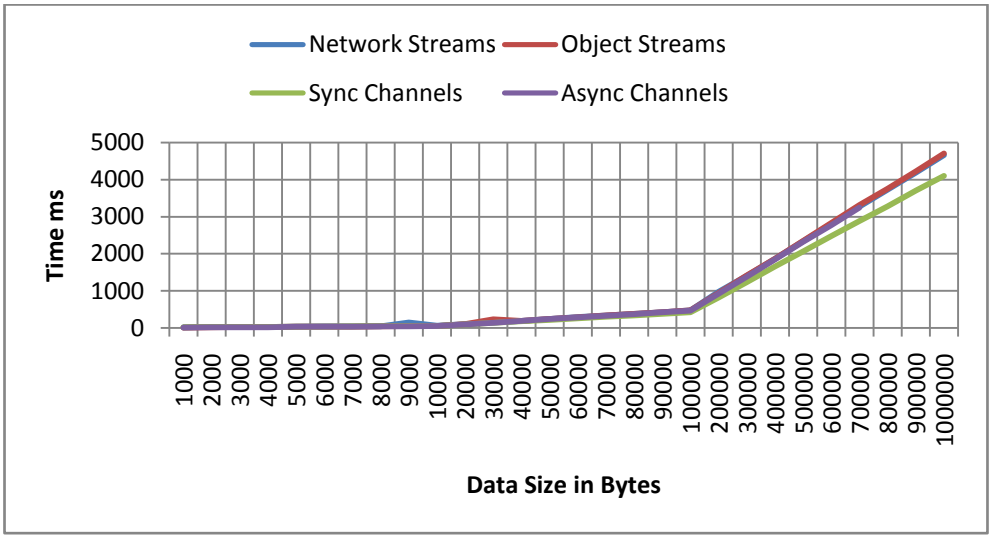


Figure 96: PC Receiving Data

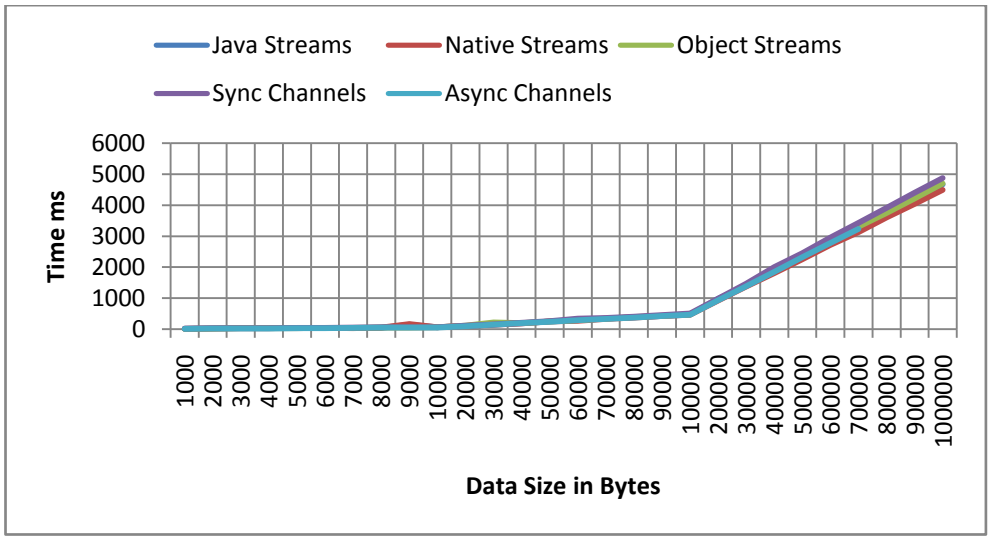


Figure 97: PDA Sending Data

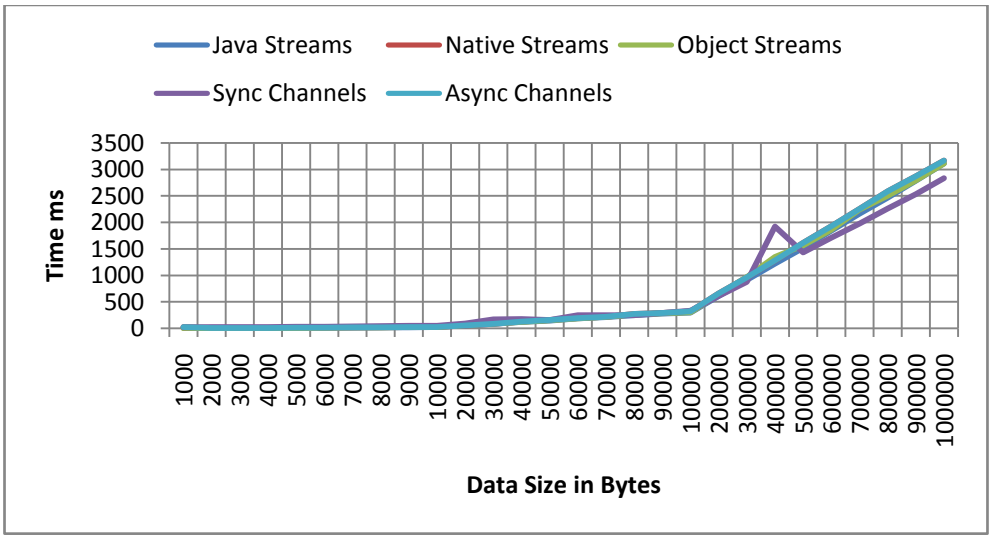


Figure 98: PDA Receiving Data

D.2.2 New Send and Receive

Figure 99 presents the times recorded for the PC to send large data packets using the new implementation of JCSP Networking, and Figure 100 presents the recorded times for the PC receiving large data sizes using the new JCSP Networking implementation. Similarly, Figure 101 presents the times recorded for the PDA to send large data packets and Figure 102 presents the times for the PDA to receive large data packets.

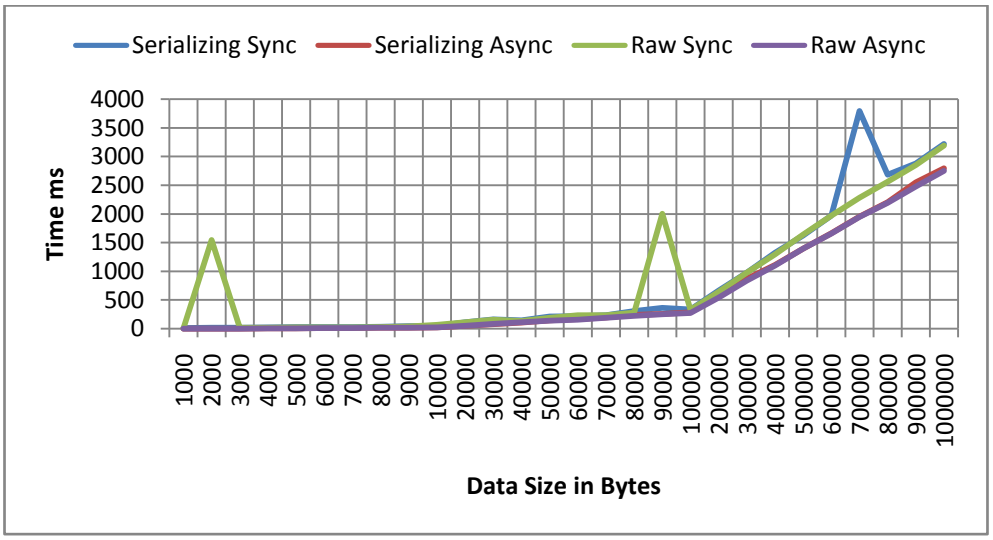


Figure 99: PC Sending Data New JCSP

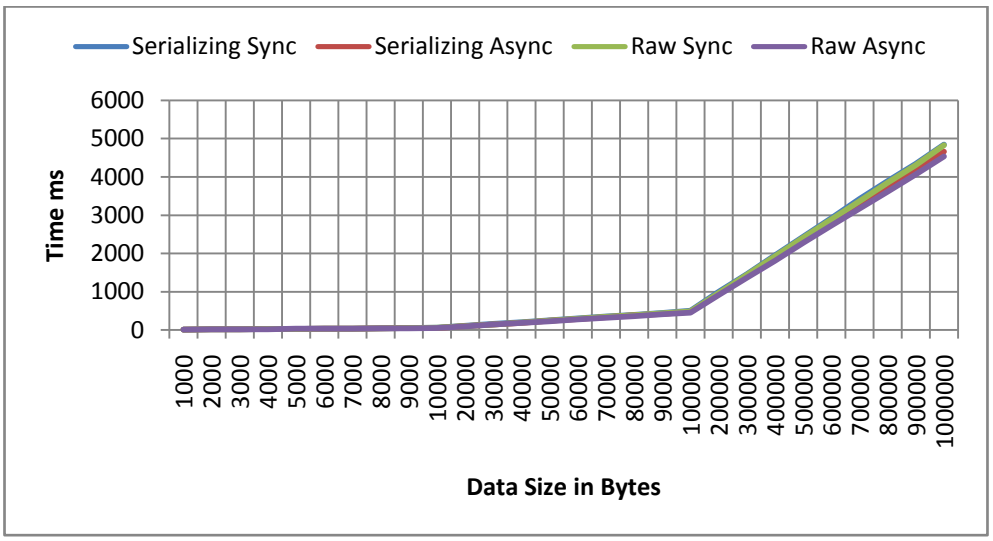


Figure 100: PC Receiving Data New JCSP

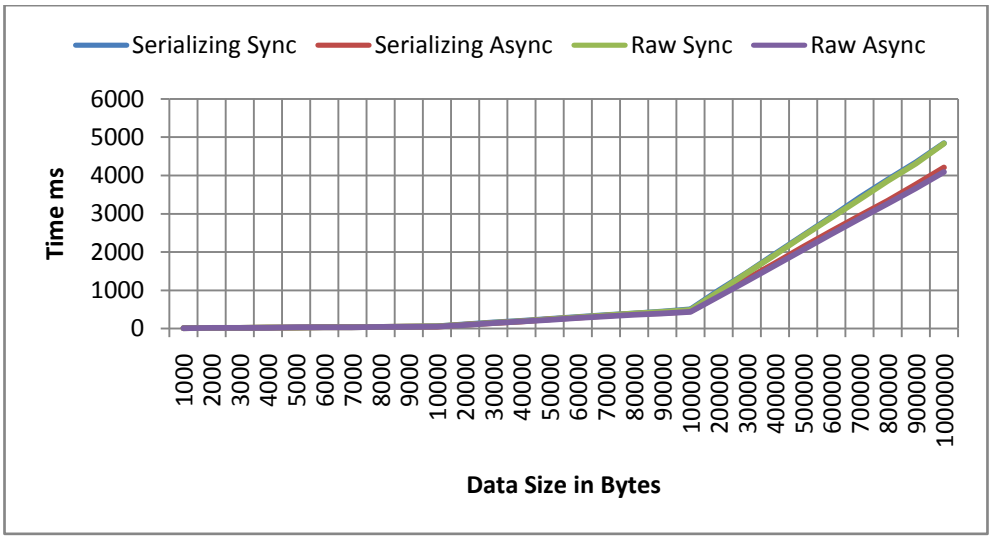


Figure 101: PDA Sending Data New JCSP

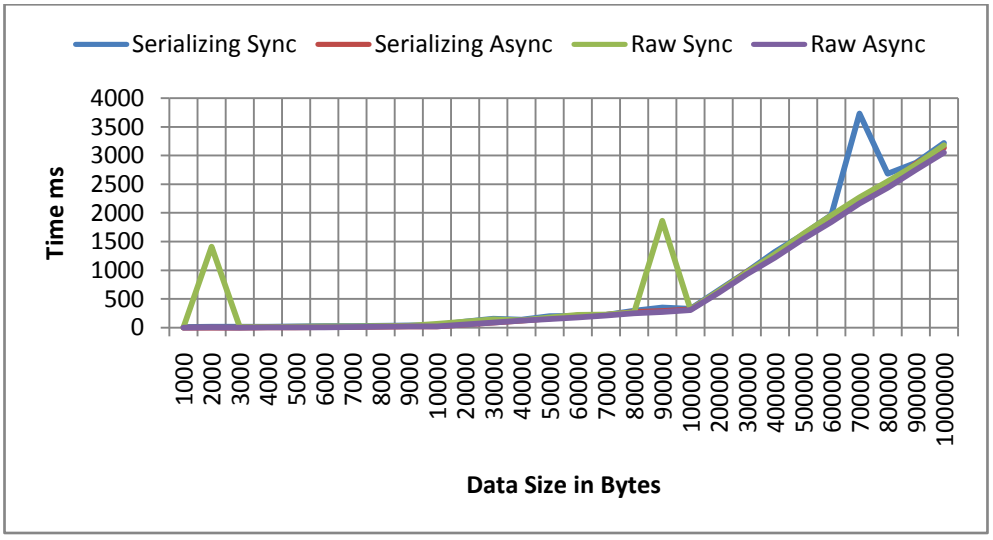


Figure 102: PDA Receiving Data New JCSP

D.2.3 Roundtrip

Figure 103 presents the times recorded on the PC during a roundtrip operation of large data sizes from the PC to the PDA and back. Figure 104 presents the times recorded on the PC for this operation, but from the PDA to PC and back. Figure 105 presents the times recorded on the PDA during a roundtrip operation from the PDA to the PC and back, and Figure 106 presents the times recorded on the PDA for a roundtrip operation from the PC to the PDA and back.

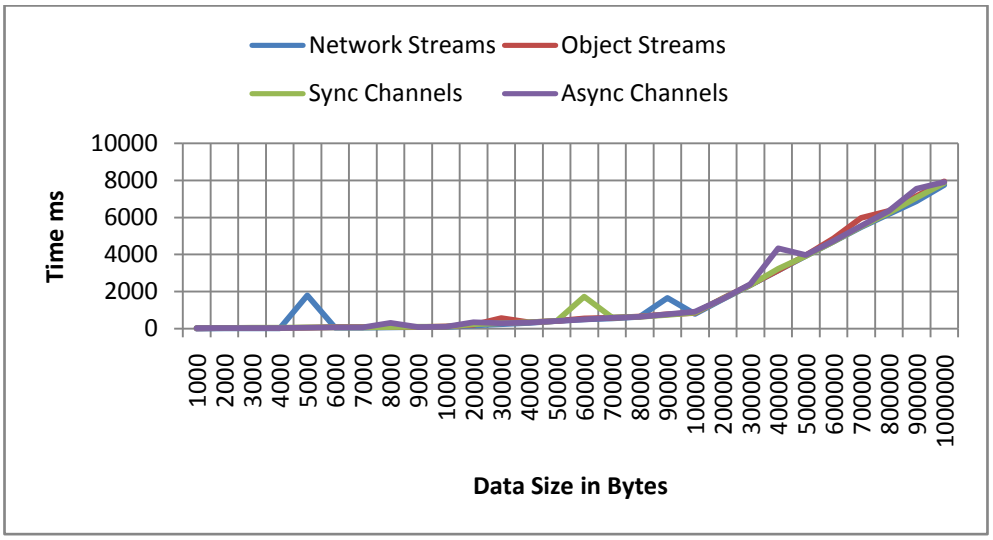


Figure 103: PC Time PC to PDA Roundtrip Data

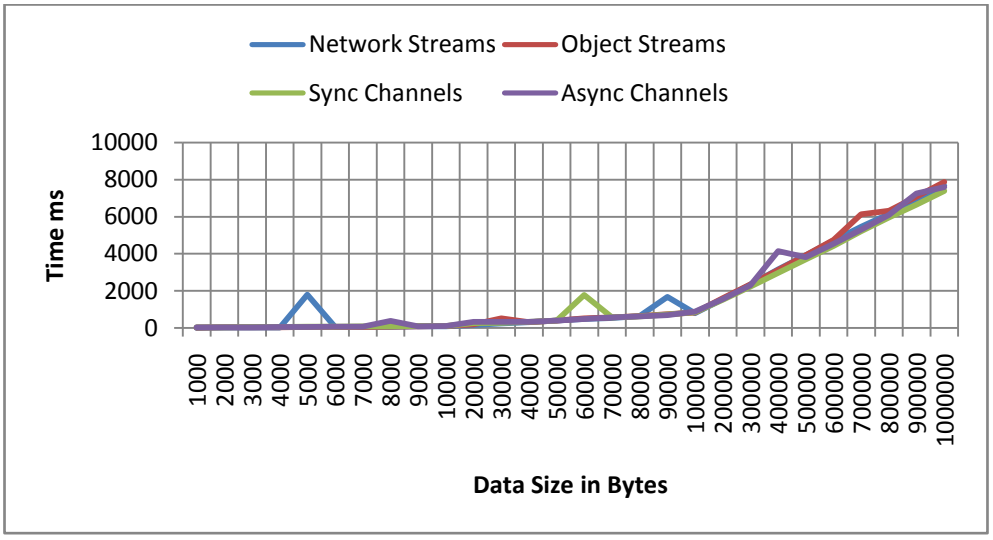


Figure 104: PC Time PDA to PC Roundtrip Data

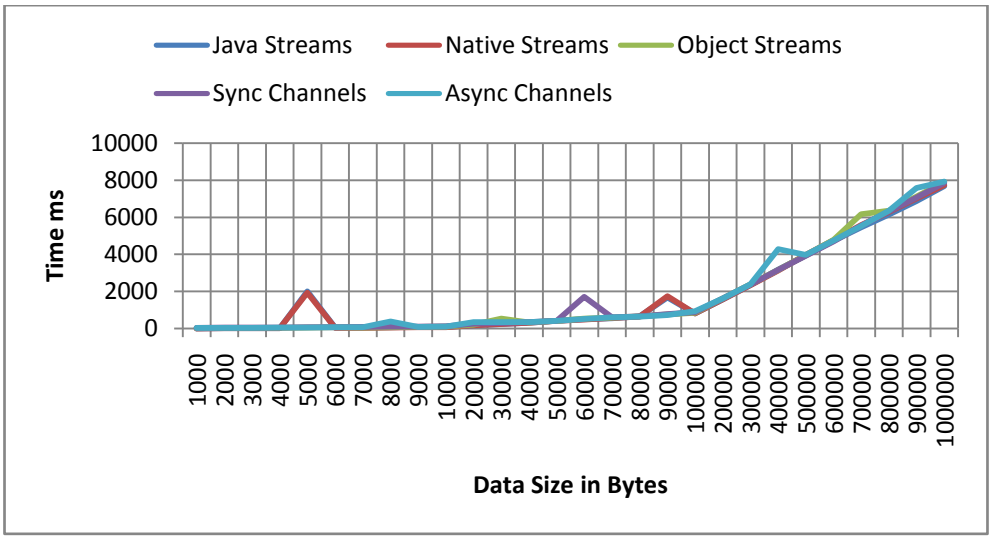


Figure 105: PDA Time PDA to PC Roundtrip Data

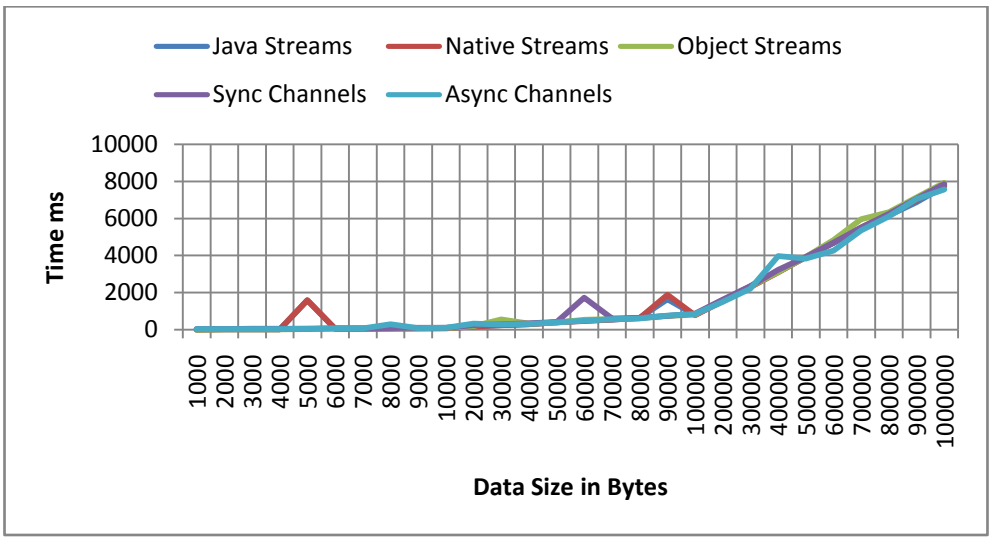


Figure 106: PDA Time PC to PDA Roundtrip Data

D.2.4 New Roundtrip

Figure 107 presents the times recorded on the PC for a roundtrip operation of large data packets from the PC to the PDA using the new JCSP Networking implementation. Figure 108 presents the times recorded on the PC for a roundtrip operation from the PDA to the PC and back using the new JCSP Networking implementation. Figure 109 presents the times recorded on the PDA for a roundtrip from the PDA to the PC, whereas Figure 110 presents the times recorded on the PDA for a roundtrip operation from the PC to the PDA and back using the new JCSP Networking implementation.

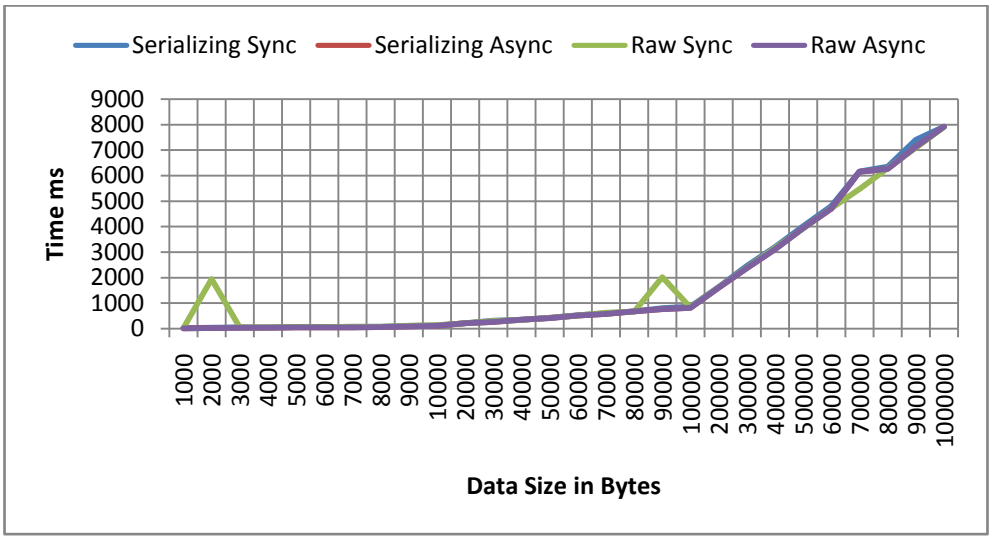


Figure 107: PC Time PC to PDA New JCSP Roundtrip Data

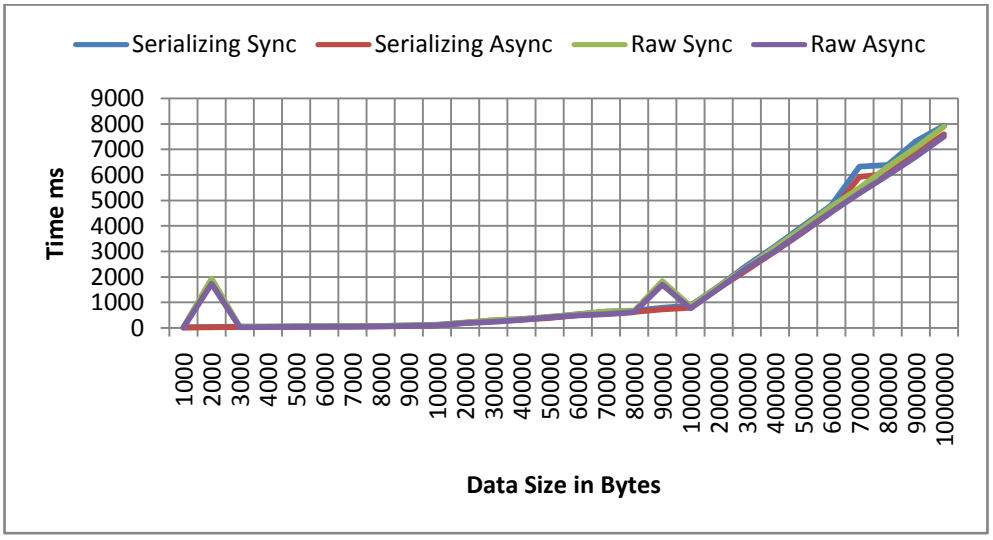


Figure 108: PC Time PDA to PC New JCSP Roundtrip Data

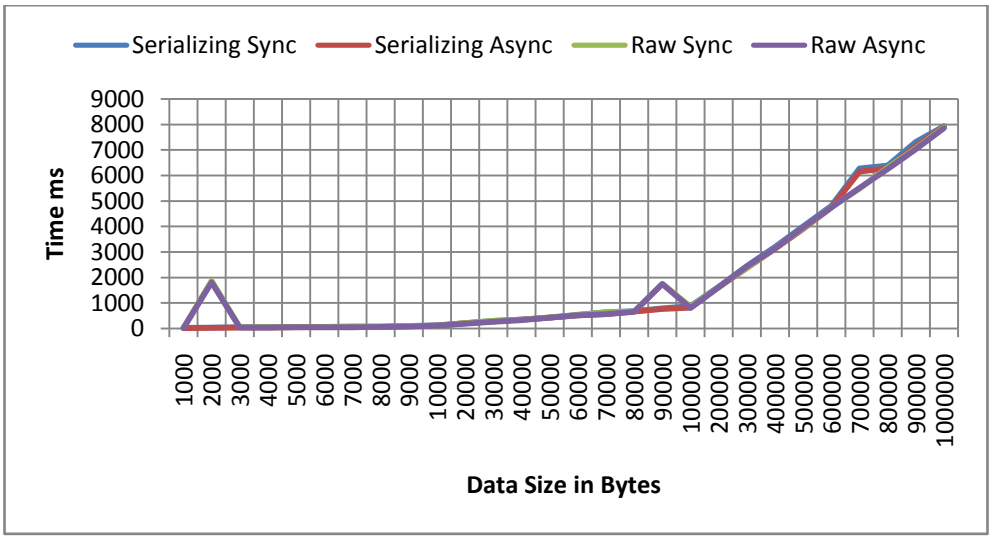


Figure 109: PDA Time PDA to PC New JCSP Roundtrip Data

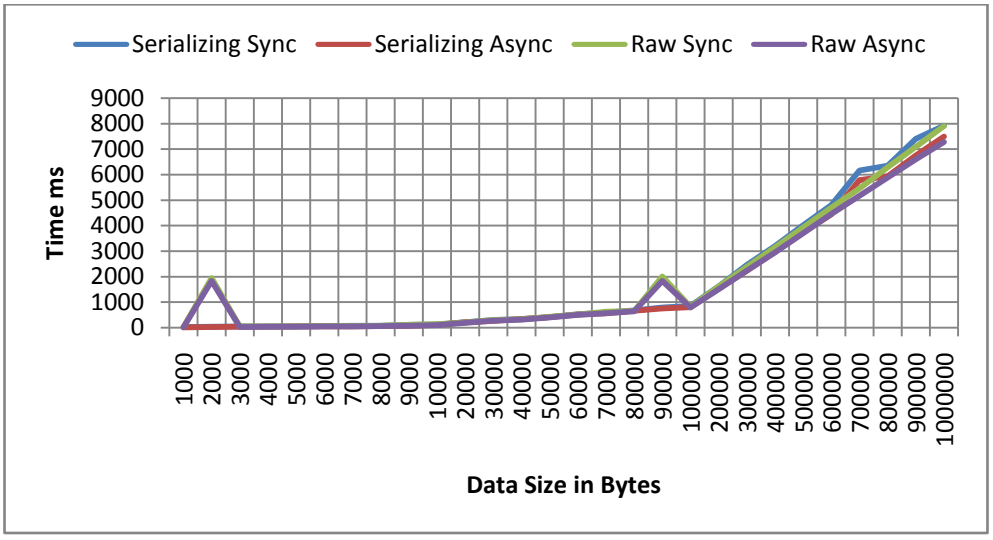


Figure 110: PDA Time PC to PDA New JCSP Roundtrip Data

D.3 Test Object Communication

D.3.1 Sending

D.3.1.1 Object Streams

Figure 111 presents the times recorded for the PC to send the various test objects, whereas Figure 112 presents the recorded times for the PDA to send the test objects.

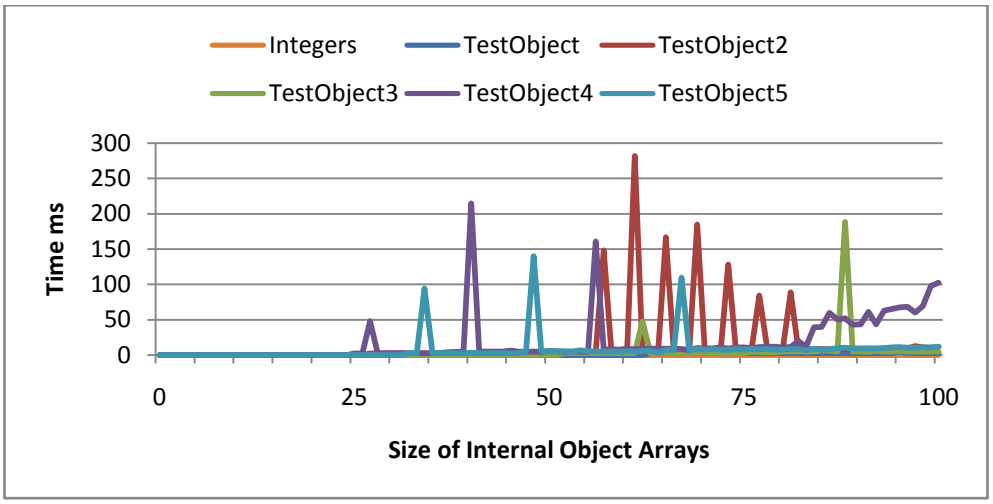


Figure 111: PC Sending TestObject via Object Streams

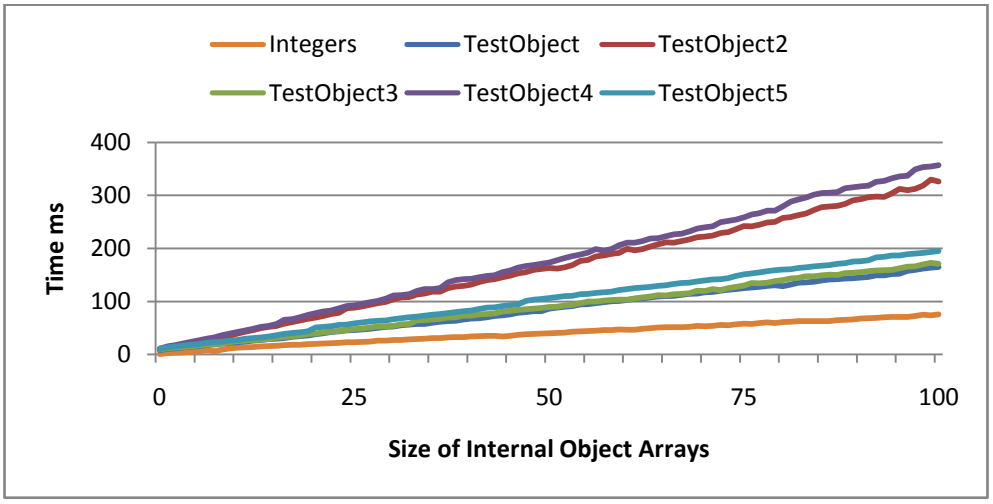


Figure 112: PDA Sending TestObject via Object Streams

D.3.1.2 Networked Channels

Figure 113 presents the expected sending time for the PC sending the various test objects using synchronous channels, and Figure 114 and Figure 115 present the actual recorded results for synchronous and asynchronous channels respectively. Likewise, Figure 116 presents the expected send times for the PDA using synchronous networked channels, with Figure 117 and Figure 118 presenting the actual results for synchronous and asynchronous channels respectively.

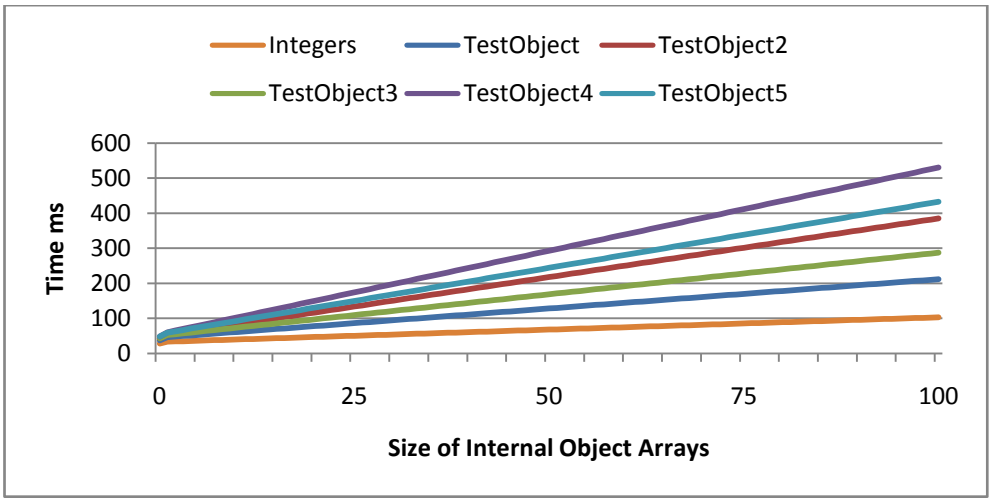


Figure 113: Expected PC Sending TestObject via Synchronous Networked Channels

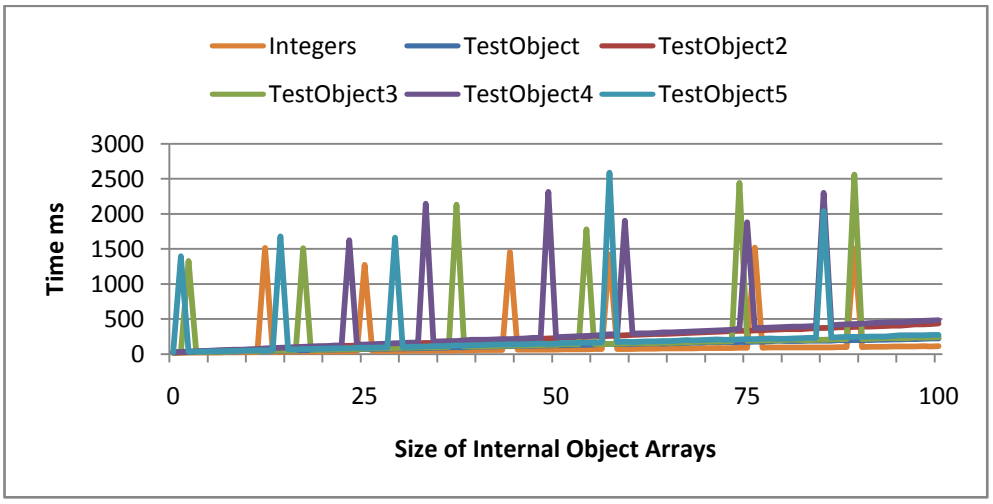


Figure 114: PC Sending TestObject via Synchronous Networked Channels

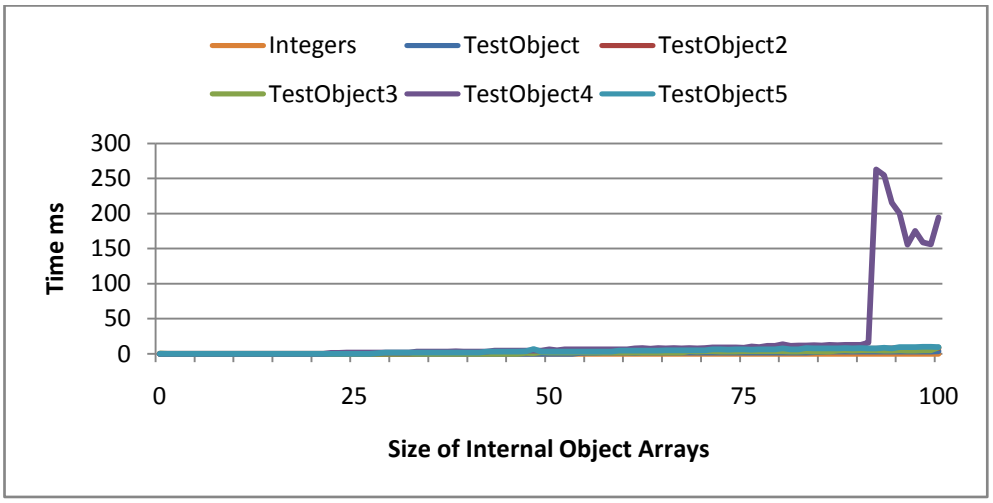


Figure 115: PC Sending TestObject via Asynchronous Networked Channels

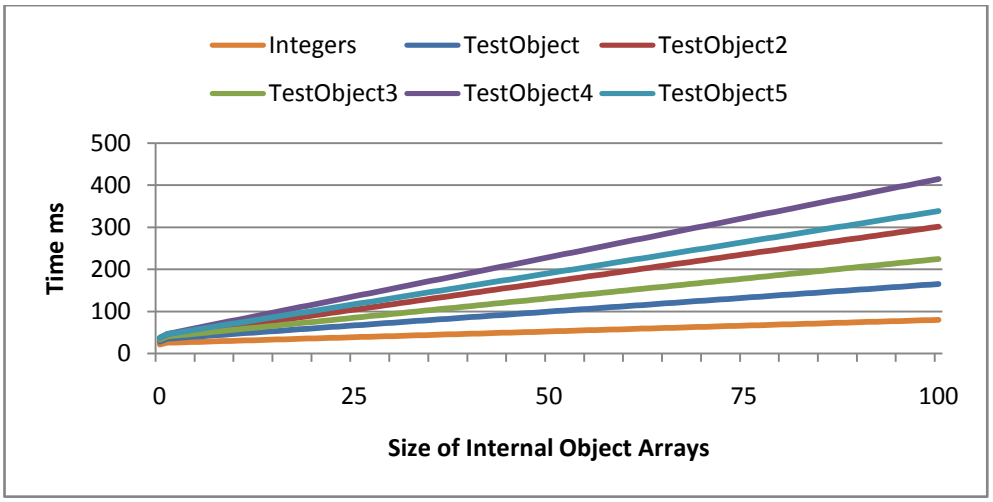


Figure 116: Expected PDA Sending TestObject via Synchronous Networked Channels

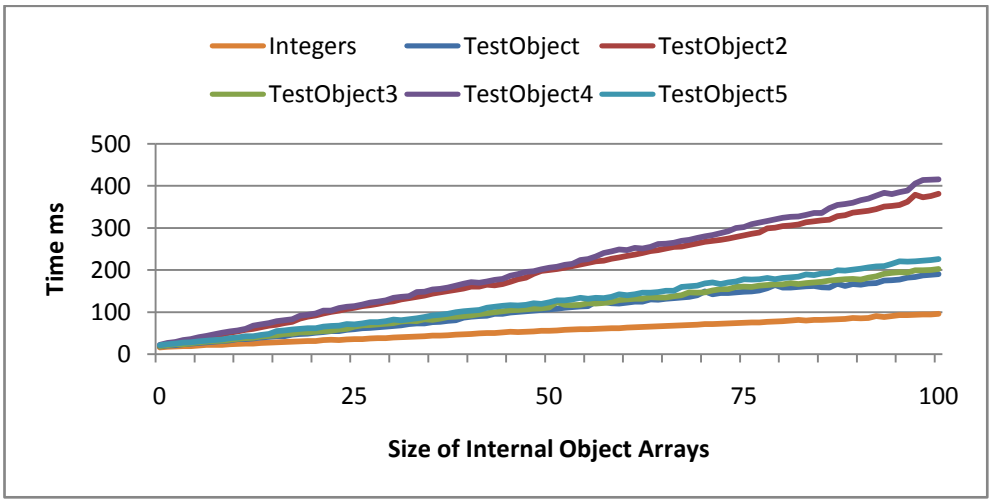


Figure 117: PDA Sending TestObject via Synchronous Networked Channels

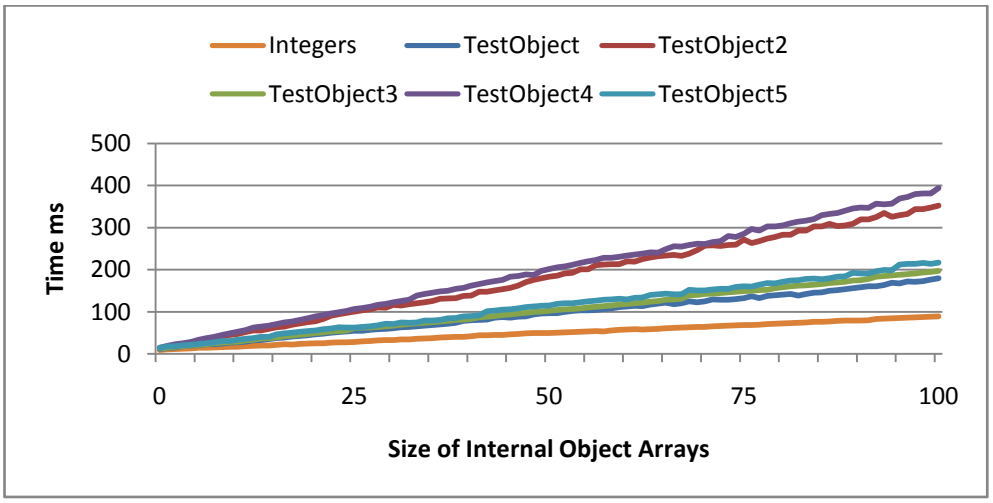


Figure 118: PDA Sending TestObject via Asynchronous Networked Channels

D.3.1.3 New Networked Channels

Figure 119 presents the expected results for the PC to synchronously send the test objects using the new JCSP Networking implementation, with Figure 120 and Figure 121 presenting the actual recorded results for the PC sending via synchronous and asynchronous channels respectively. Figure 122 presents the expected times for the PDA to send the test objects within the new JCSP Networking implementation, and Figure 123 and Figure 124 present the actual results for synchronous and asynchronous channels respectively.

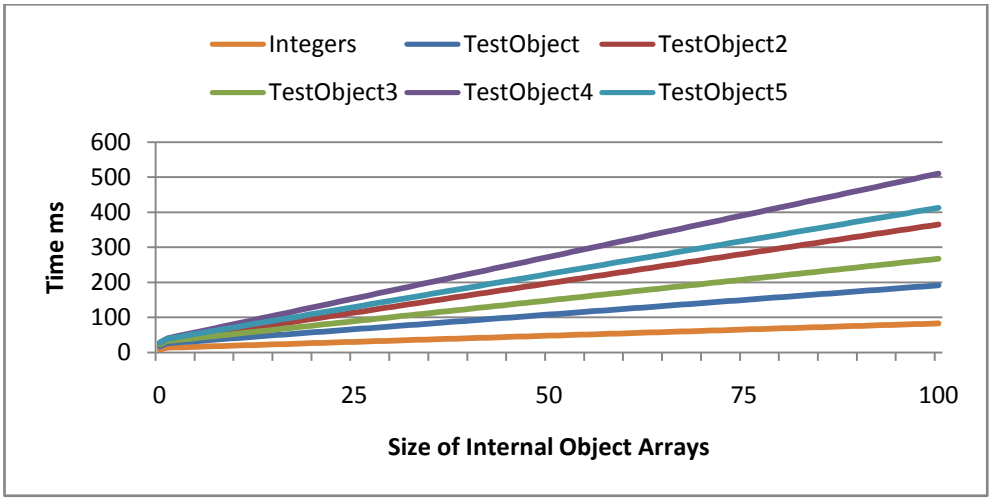


Figure 119: Expected PC Sending TestObject via New Synchronous Networked Channels

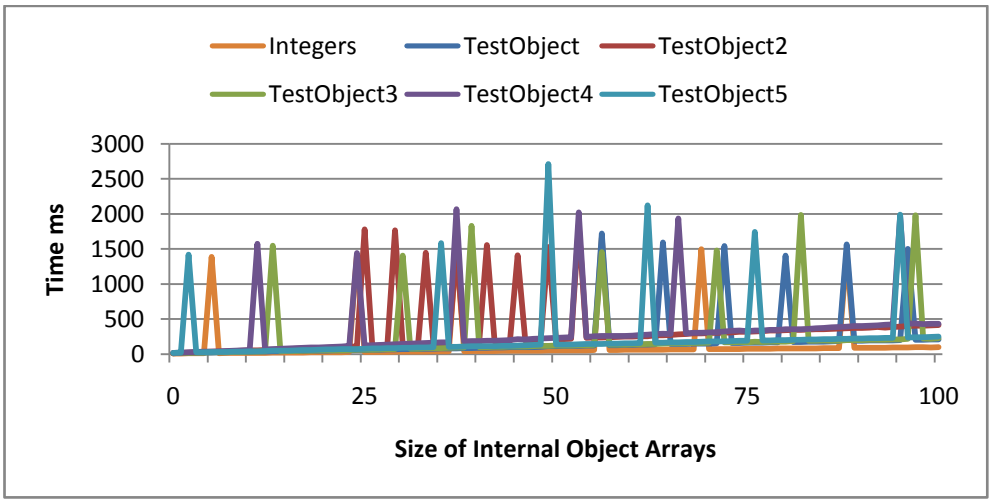


Figure 120: PC Sending TestObject via New Synchronous Networked Channels

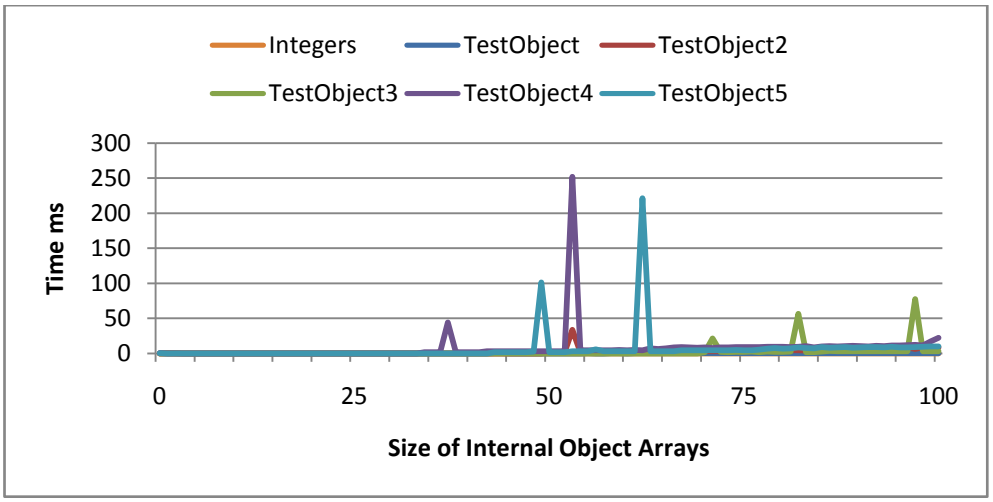


Figure 121: PC Sending TestObject via New Asynchronous Networked Channels

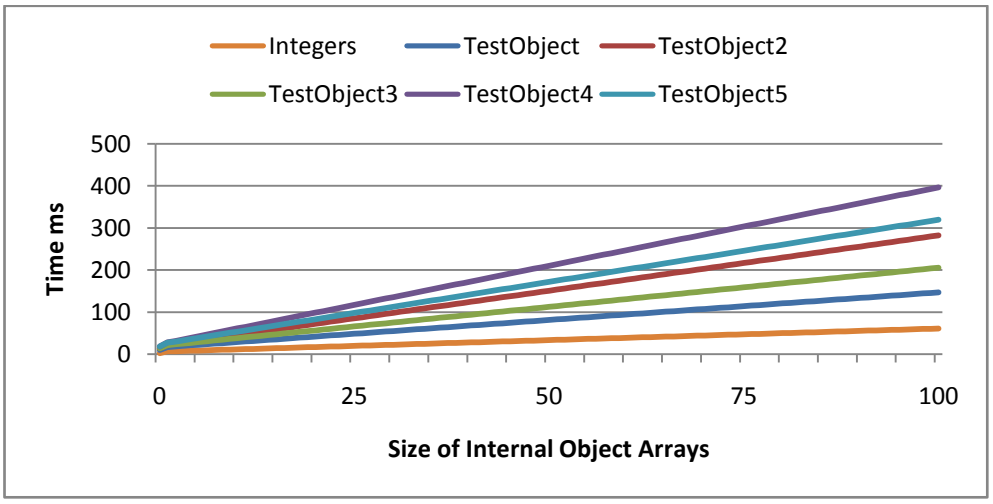


Figure 122: Expected PDA Sending TestObject via New Synchronous Networked Channels

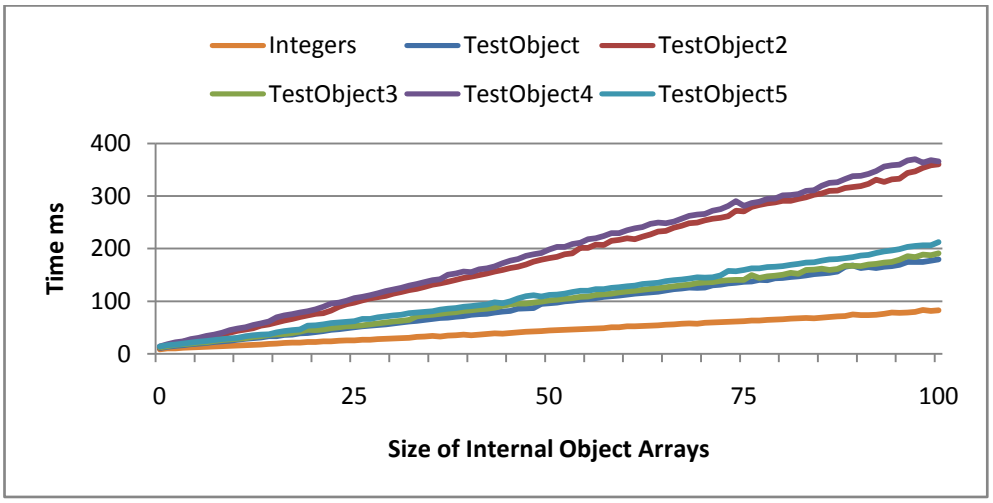


Figure 123: PDA Sending TestObject via New Synchronous Networked Channels

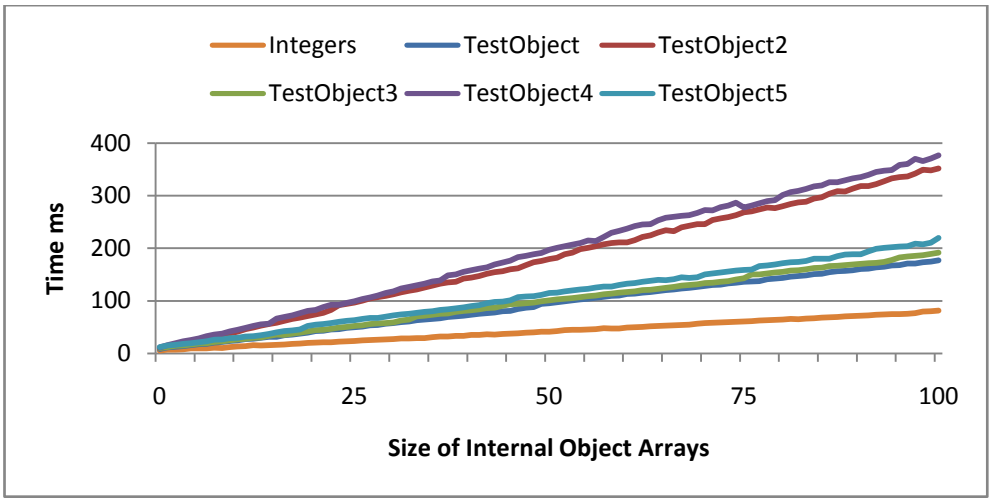


Figure 124: PDA Sending TestObject via New Asynchronous Networked Channels

D.3.2 Receiving

D.3.2.1 Object Streams

Figure 125 presents the results recorded for the PC to receive the test objects utilising object streams, whereas Figure 126 presents the times recorded for the PDA to receive the various test objects via object streams.

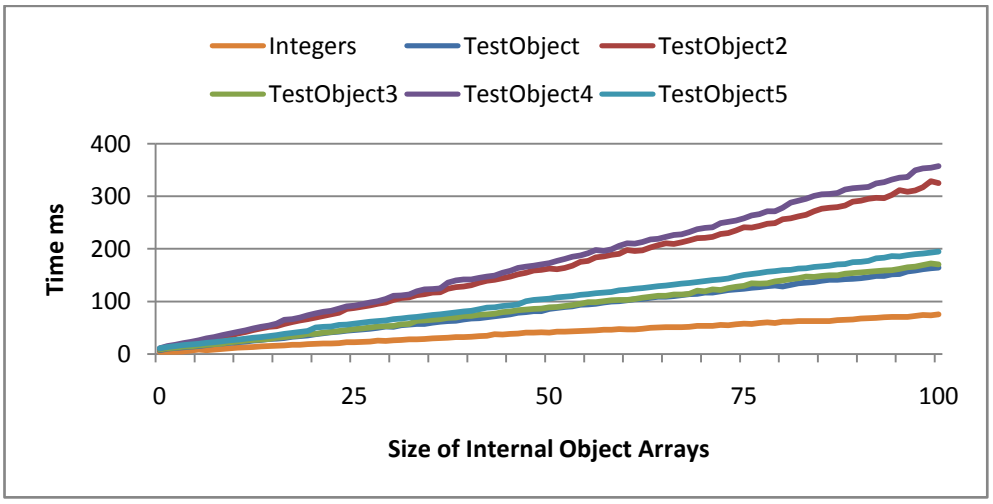


Figure 125: PC Receiving TestObject via Object Streams

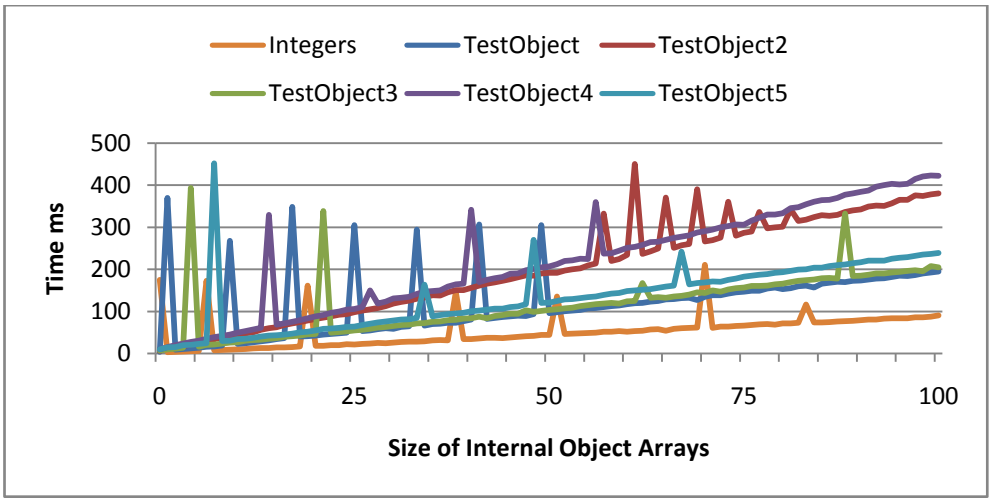


Figure 126: PDA Receiving TestObject via Object Streams

D.3.2.2 Networked Channels

Figure 127 presents the recorded times for the PC to receive the test objects via synchronous networked channels in the original JCSP Networking implementation, and Figure 128 presents the recorded times for the PC to receive the test objects via asynchronous channels. Figure 129 presents the recorded times for the PDA to receive the test objects using synchronous channels, whereas Figure 130 presents the times recorded for the PDA to receive the test objects via asynchronous channels.

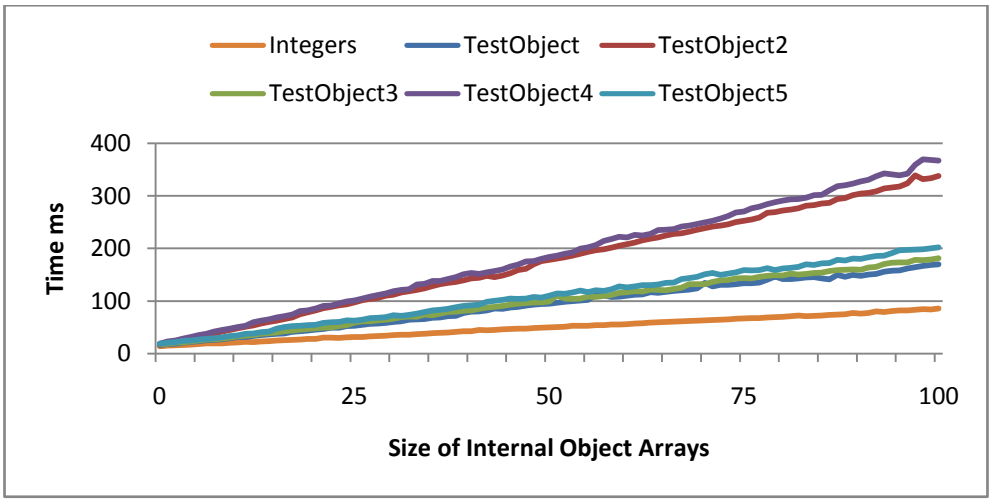


Figure 127: PC Receiving TestObject via Synchronous Networked Channels

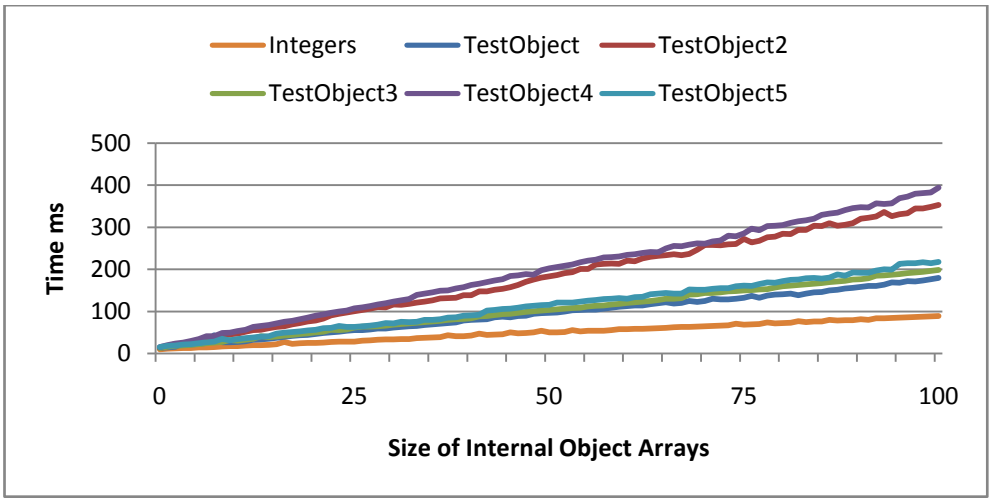


Figure 128: PC Receiving TestObject via Asynchronous Networked Channels

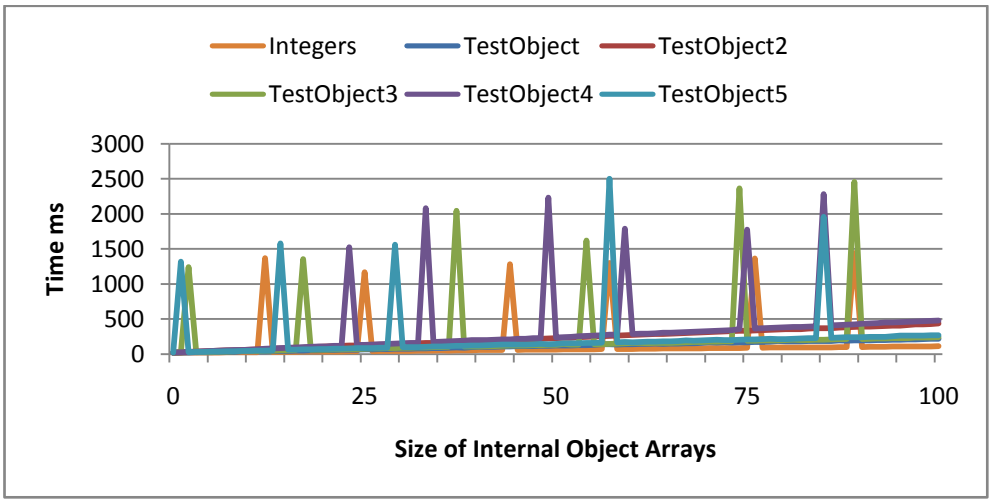


Figure 129: PDA Receiving TestObject via Synchronous Networked Channels

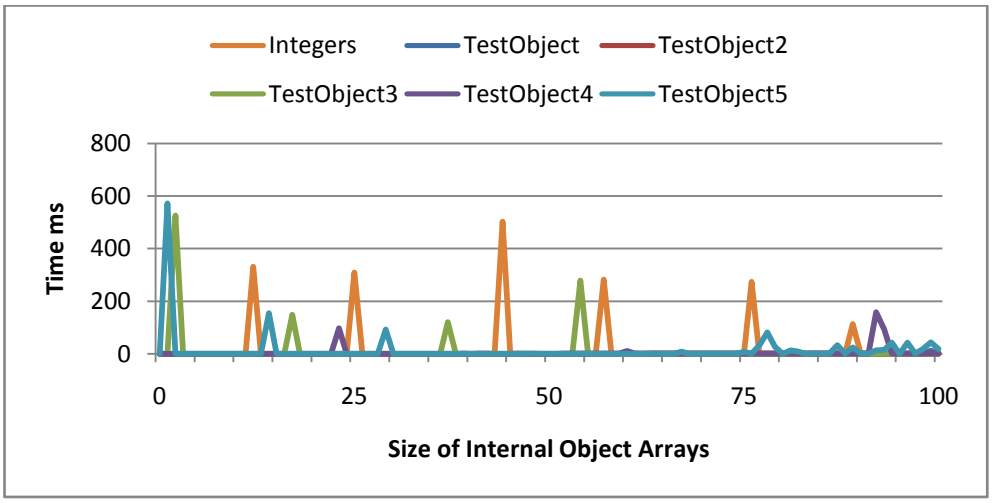


Figure 130: PDA Receiving TestObject via Asynchronous Networked Channels

D.3.2.3 New Networked Channels

Figure 131 presents the recorded times for the PC to receive the test objects via synchronous channels in the new JCSP Networking implementation, and Figure 132 presents the results for the PC to receive the test objects asynchronously. Figure 133 presents the results for the PDA to receive the test objects synchronously within the new JCSP Networking implementation, whereas Figure 134 presents the times recorded for the PDA to receive the test objects using asynchronous channels.

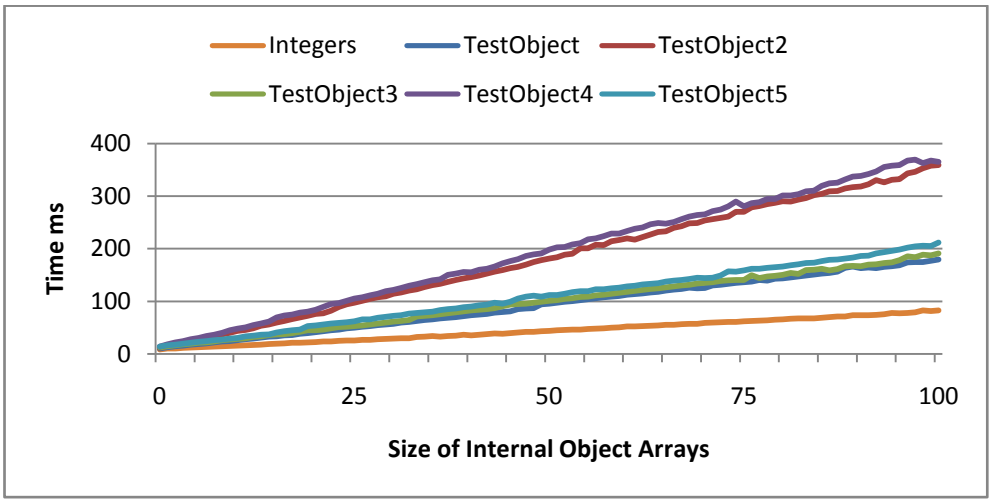


Figure 131: PC Receiving TestObject via Synchronous New Networked Channels

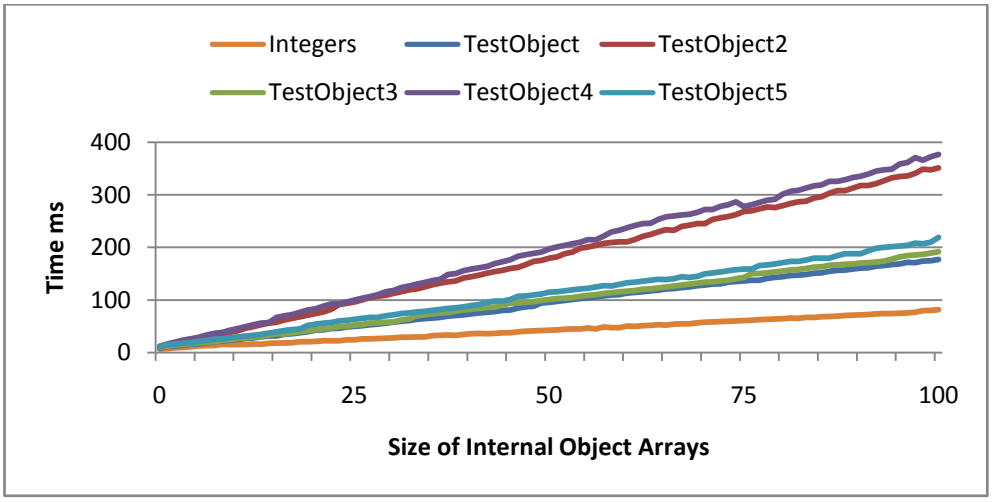


Figure 132: PC Receiving TestObject via Asynchronous New Networked Channels

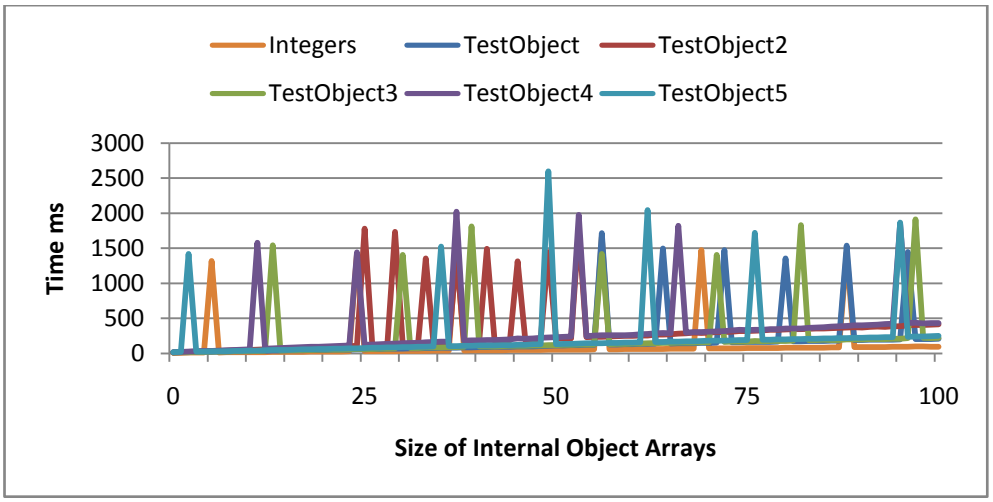


Figure 133: PDA Receiving TestObject via Synchronous New Networked Channels

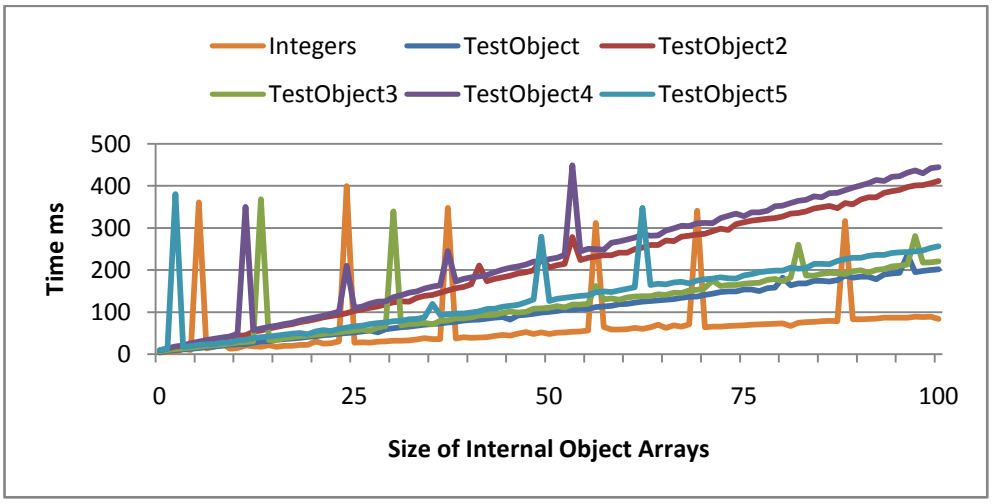


Figure 134: PDA Receiving TestObject via Asynchronous New Networked Channels

D.3.3 Roundtrip

D.3.3.1 Object Streams

Figure 135 presents the times recorded on the PC during a roundtrip operation on the test objects from the PC to the PDA and back, using object streams. Figure 136 presents the times recorded on the PC for a roundtrip from the PDA to PC and back. Figure 137 presents the times recorded on the PDA for a roundtrip operation from the PDA to the PC and back using object streams, and Figure 138 presents the times recorded on the PDA for a roundtrip operation from the PC to the PDA and back with the test objects.

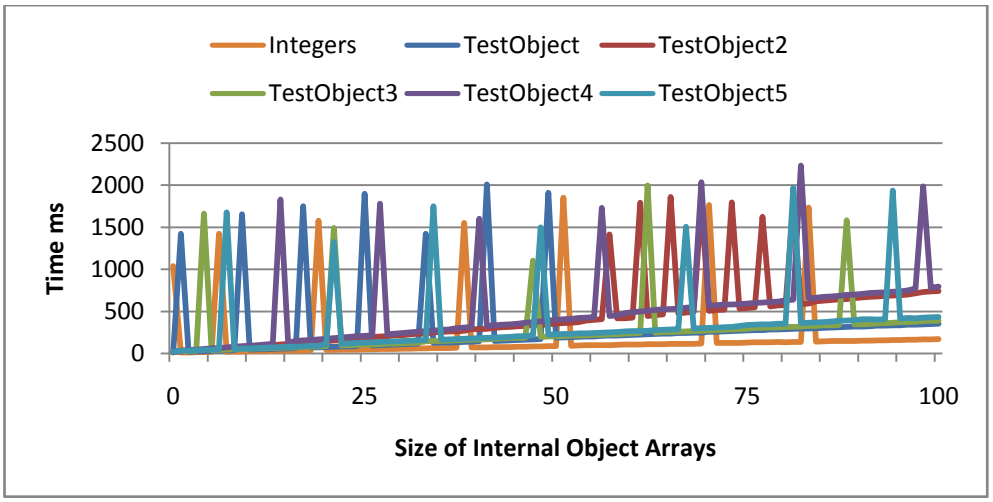


Figure 135: PC Time PC to PDA TestObject Roundtrip via Object Streams

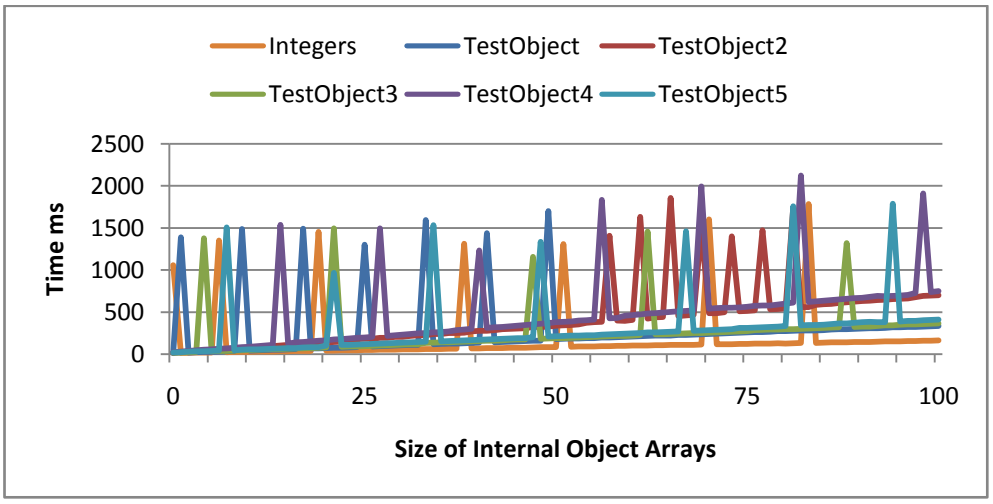


Figure 136: PC Time PDA to PC TestObject Roundtrip via Object Streams

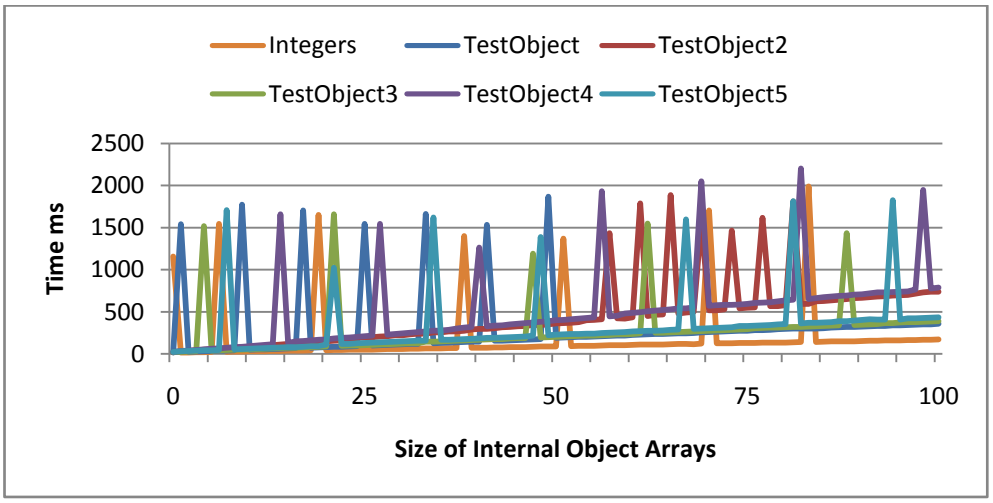


Figure 137: PDA Time PDA to PC TestObject Roundtrip via Object Streams

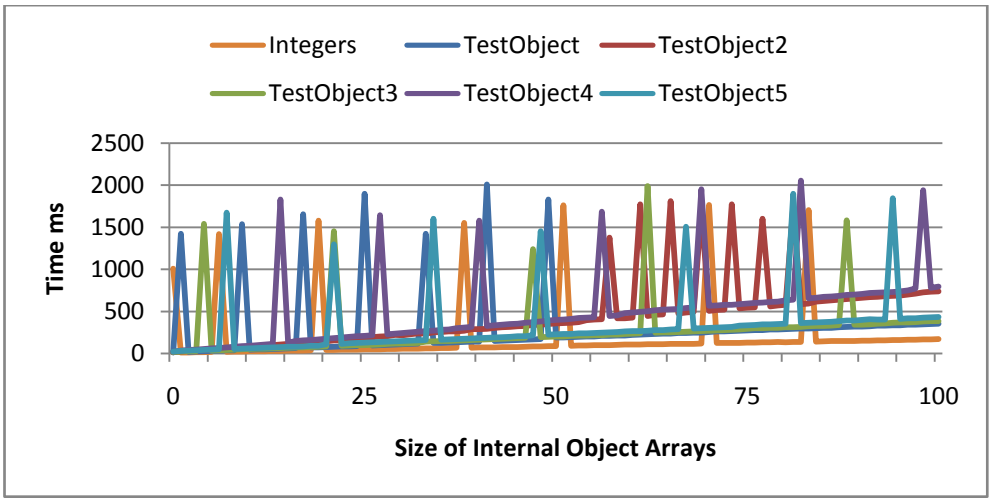


Figure 138: PDA Time PC to PDA TestObject Roundtrip via Object Streams

D.3.3.2 Networked Channels

Figure 139 presents the times recorded on the PC for a synchronous networked channel roundtrip operation with the test objects from the PC to PDA and back, whereas Figure 140 presents the times recorded on the PC for the same operation asynchronously. Figure 141 presents the times recorded for a synchronous networked channel roundtrip operation with the test objects from the PDA to PC and back, with Figure 142 presenting the results for this operation using asynchronous channels.

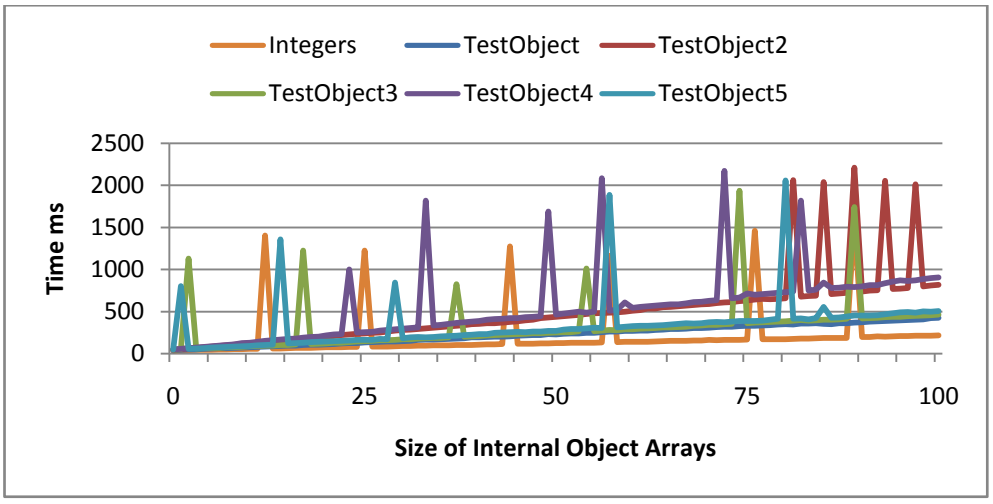


Figure 139: PC Time PC to PDA TestObject Roundtrip via Synchronous Networked Channels

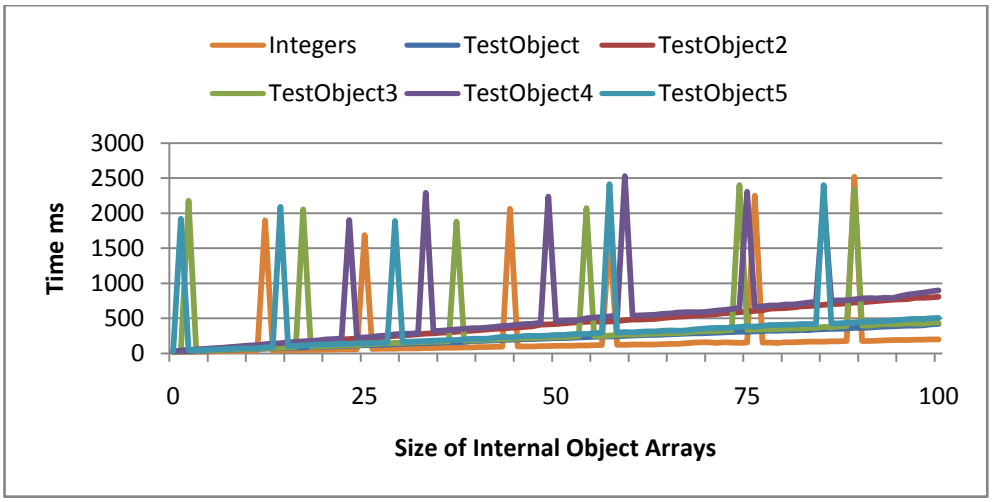


Figure 140: PC Time PC to PDA TestObject Roundtrip via Asynchronous Networked Channels

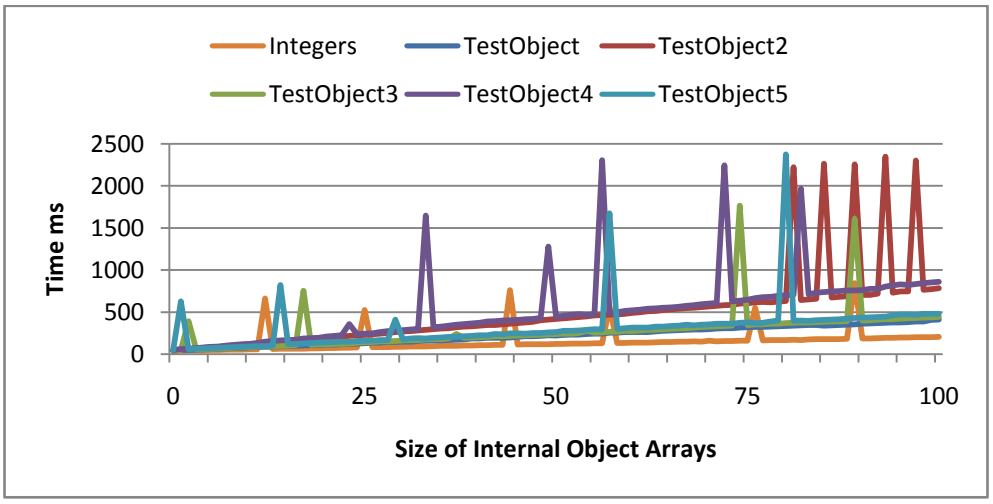


Figure 141: PC Time PDA to PC TestObject Roundtrip via Synchronous Networked Channels

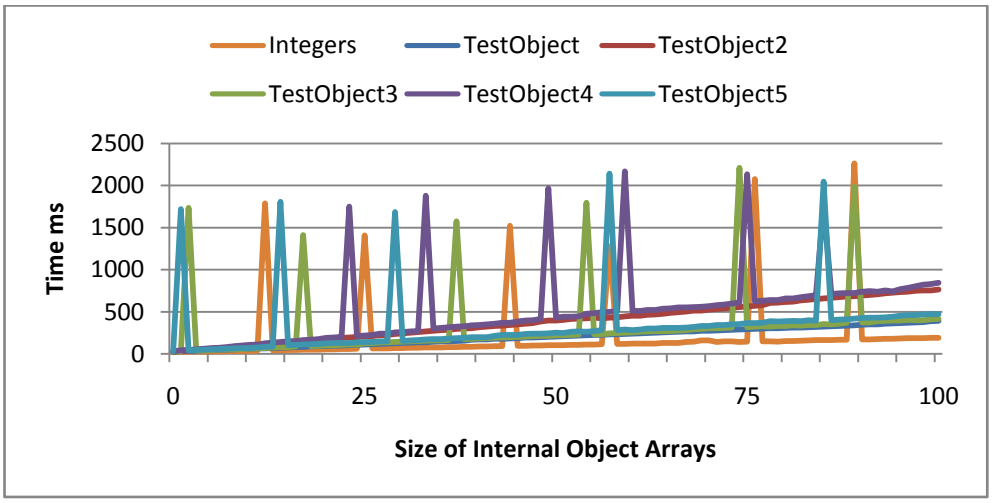


Figure 142: PC Time PDA to PC TestObject Roundtrip via Asynchronous Networked Channels

Figure 143 presents the times recorded on the PDA for a synchronous networked channel roundtrip operation from the PDA to PC and back using the test objects, with Figure 144 presenting the results for this operation performed asynchronously. Figure 145 presents the results recorded on the PDA for a synchronous roundtrip operation from the PC to the PDA and back, and Figure 146 presents the recorded times for this operation performed asynchronously.

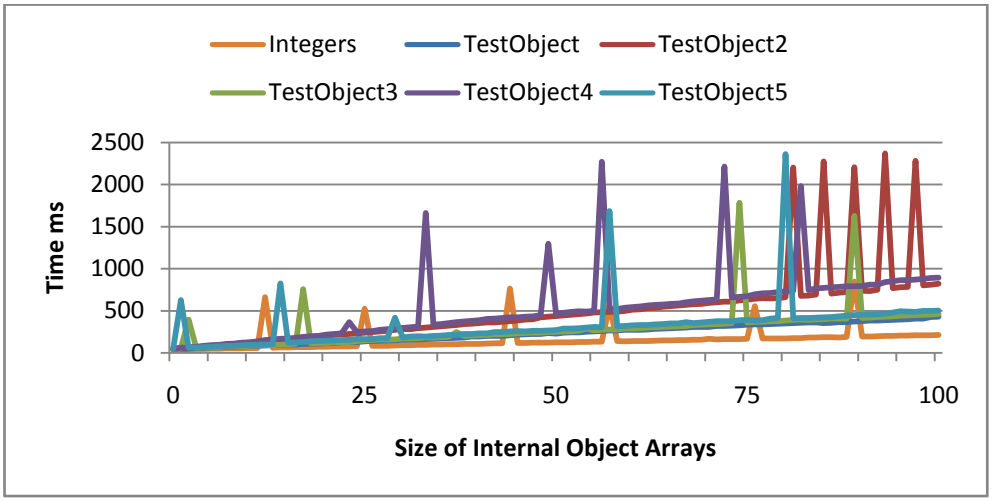


Figure 143: PDA Time PDA to PC TestObject Roundtrip via Synchronous Networked Channels

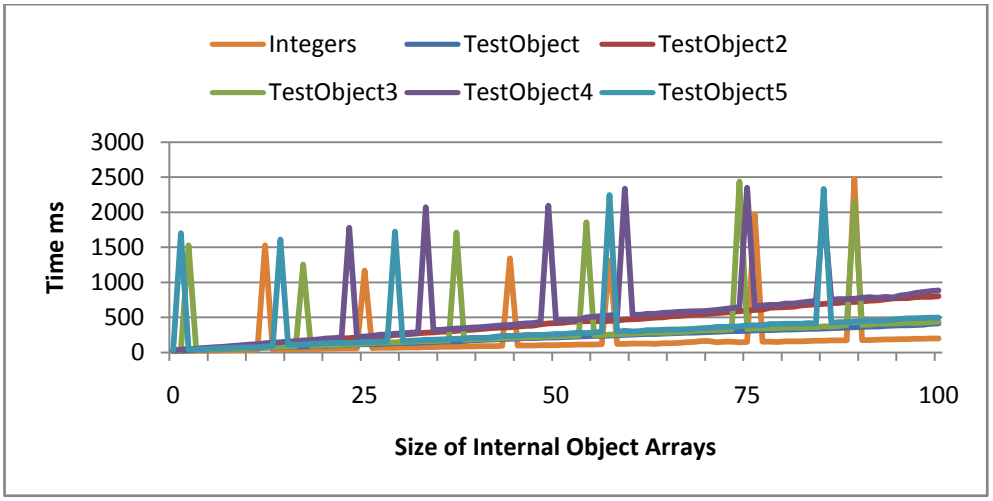


Figure 144: PDA Time PDA to PC TestObject Roundtrip via Asynchronous Networked Channels

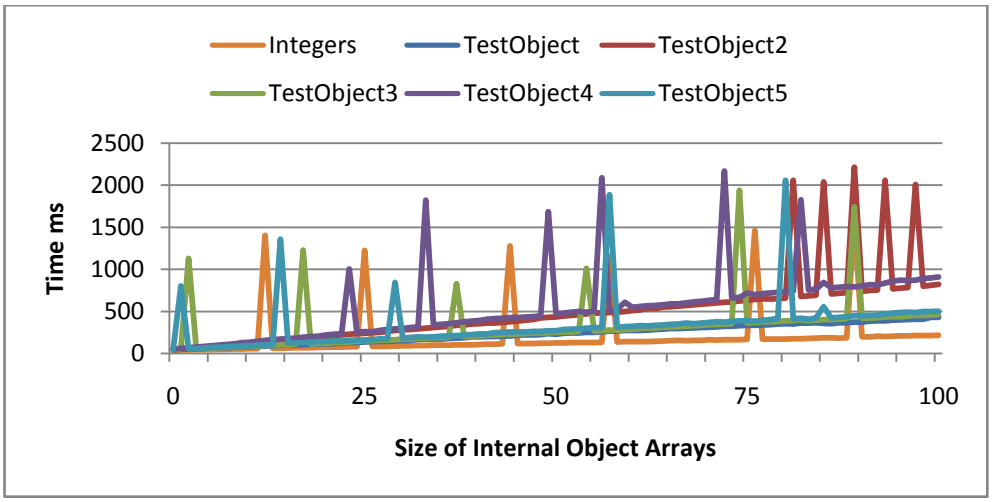


Figure 145: PDA Time PC to PDA TestObject Roundtrip via Synchronous Networked Channels

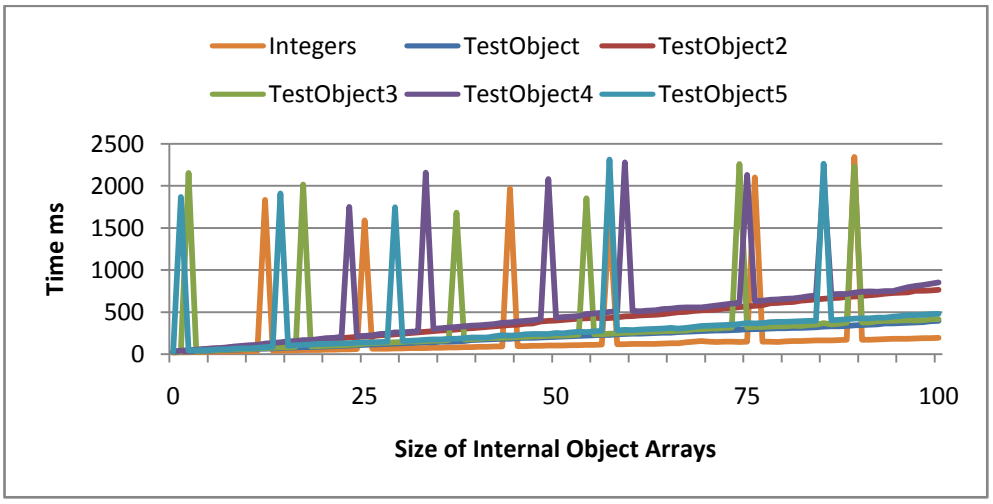


Figure 146: PDA Time PC to PDA TestObject Roundtrip via Asynchronous Networked Channels

D.3.3.3 New Networked Channels

Figure 147 presents the times recorded on the PC to perform a synchronous roundtrip operation, from the PC to the PDA and back, within the new JCSP Networking implementation, using the test objects. Figure 148 presents the results for this operation performed asynchronously. Figure 149 presents the results recorded on the PC for a synchronous roundtrip operation from the PDA to PC and back with the test objects, with Figure 150 providing the asynchronous results.

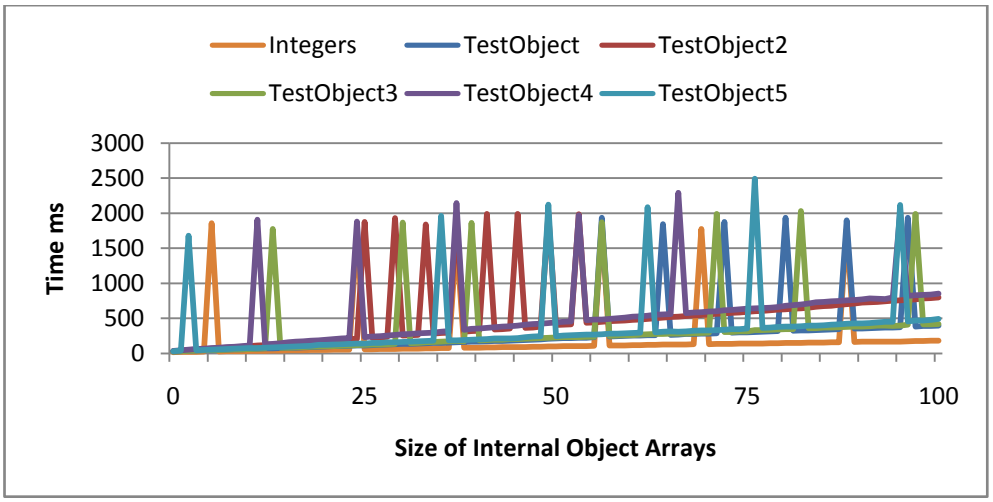


Figure 147: PC Time PC to PDA TestObject Roundtrip via Synchronous New Networked Channels

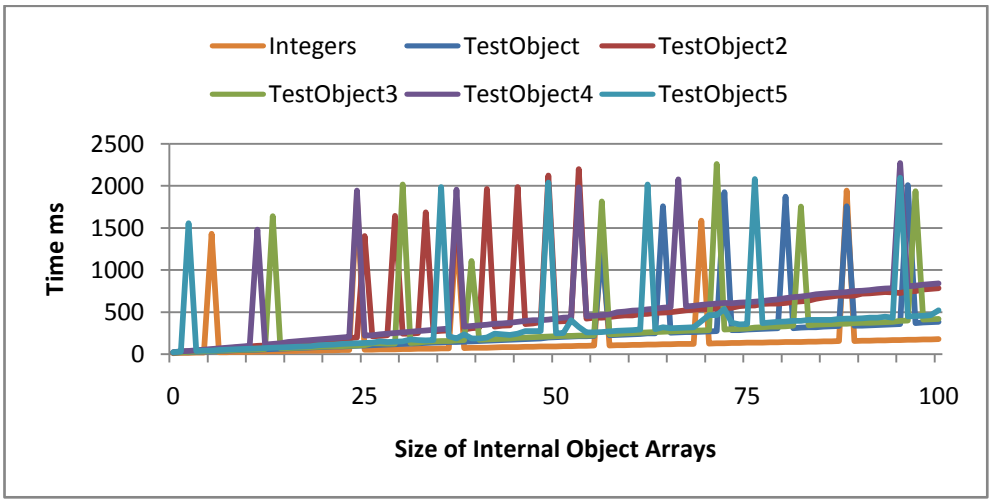


Figure 148: PC Time PC to PDA TestObject Roundtrip via Asynchronous New Networked Channels

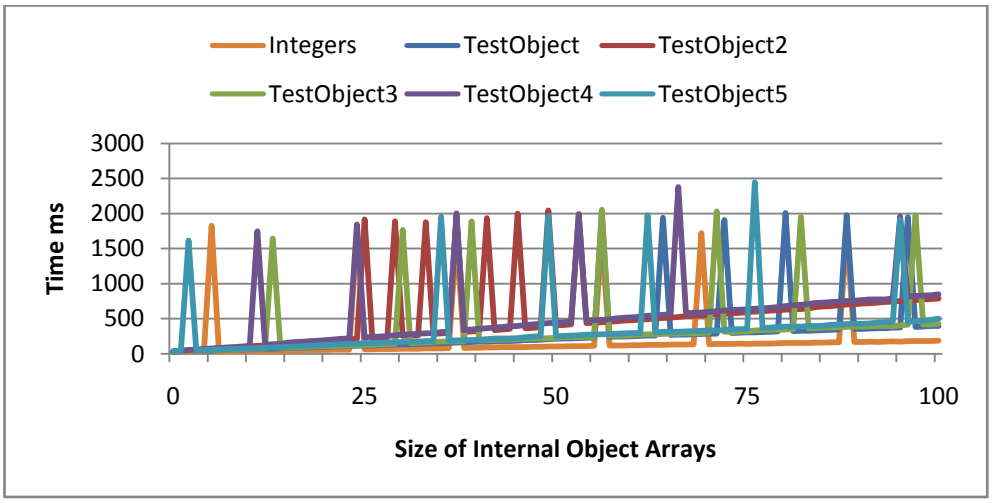


Figure 149: PC Time PDA to PC TestObject Roundtrip via Synchronous New Networked Channels

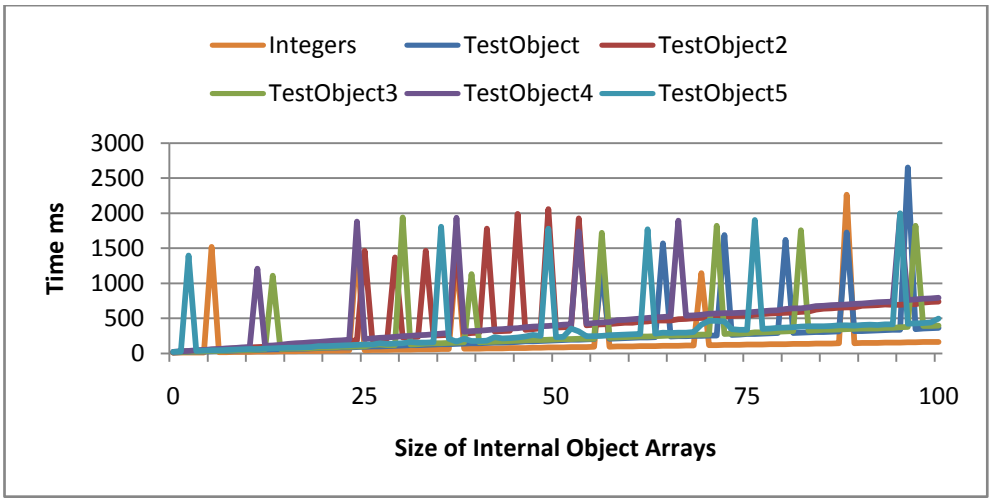


Figure 150: PC Time PDA to PC TestObject Roundtrip via Asynchronous New Networked Channels

Figure 151 presents the results recorded on the PDA for a synchronous roundtrip operation from the PDA to the PC and back, using the new implementation of JCSP Networking and the test objects. Figure 152 provides the asynchronous results for this operation. Figure 153 presents the PDA recorded times for synchronous test object roundtrip operations from the PC to PDA and back using the new JCSP Networking implementation, whereas Figure 154 presents the times recorded for these operations performed asynchronously.

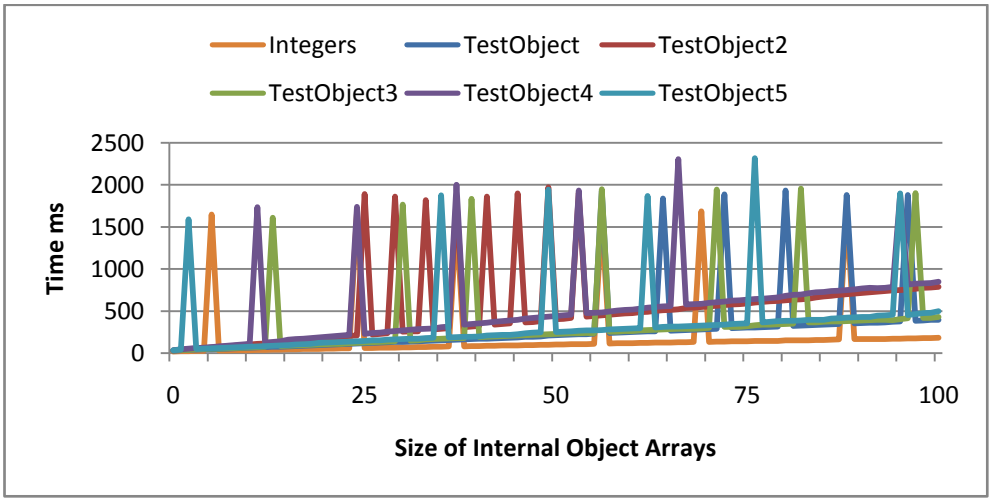


Figure 151: PDA Time PDA to PC TestObject Roundtrip via Synchronous New Networked Channels

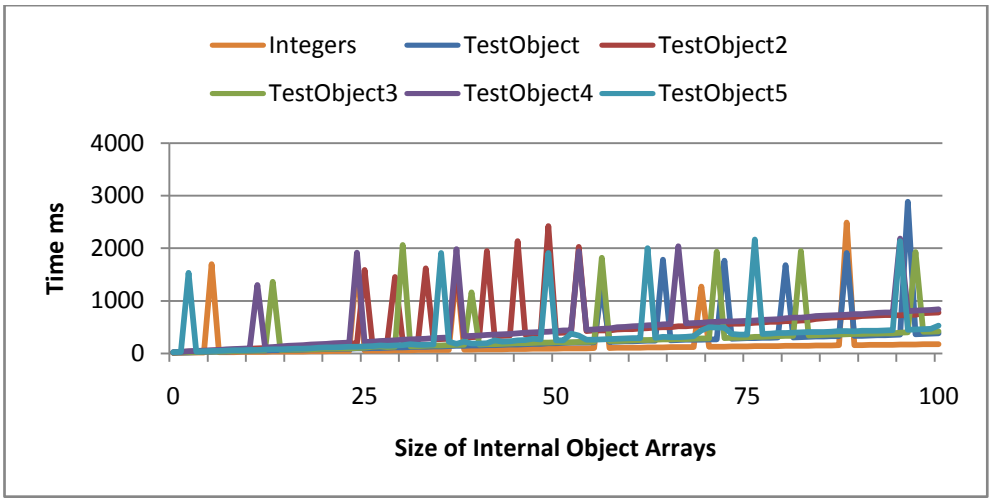


Figure 152: PDA Time PDA to PC TestObject Roundtrip via Asynchronous New Networked Channels

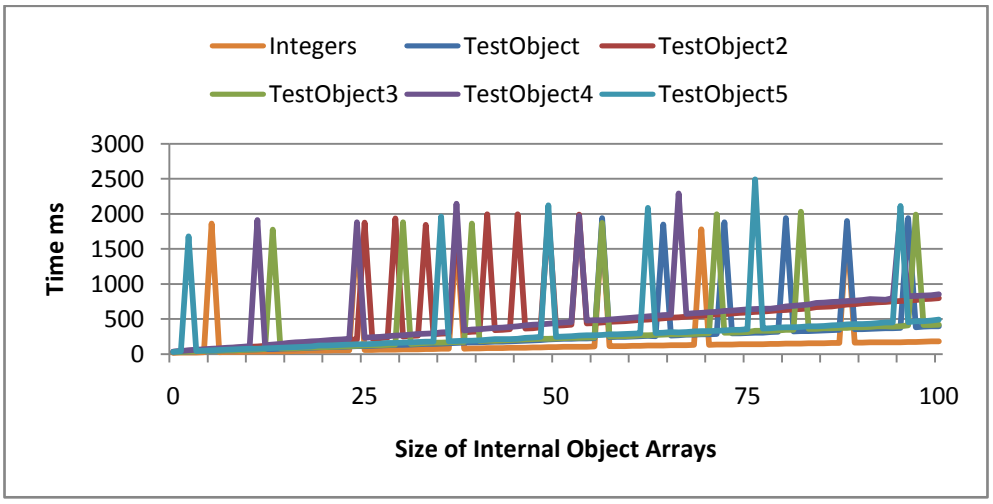


Figure 153: PDA Time PC to PDA TestObject Roundtrip via Synchronous New Networked Channels

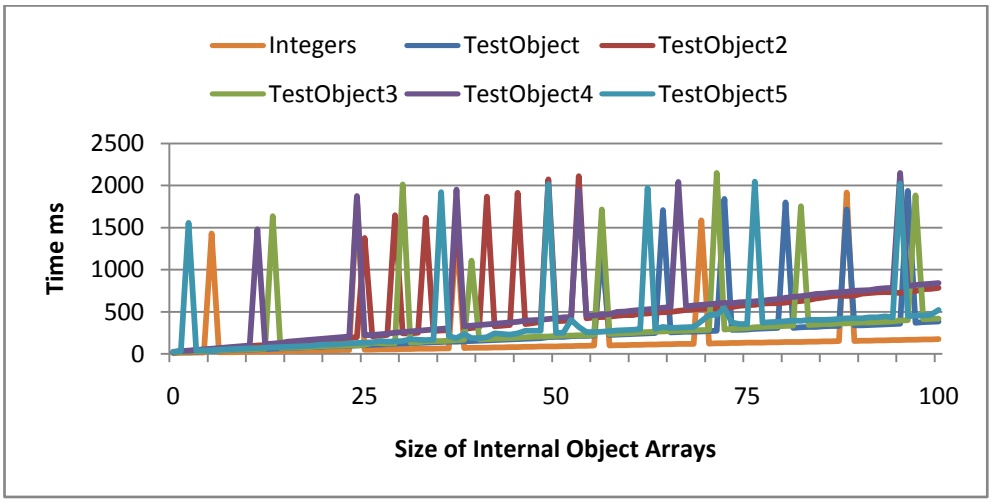


Figure 154: PDA Time PC to PDA TestObject Roundtrip via Asynchronous New Networked Channels

Appendix E Network Protocol Definition

E.1 Channel Messages

Message	Value	Description
SEND	1	A data message sent from a NetChannelOutput to a NetChannelInput. This takes the form (SEND, <destination>, <source>, <size>, <bytes>).
ACK	2	The acknowledgement sent from a NetChannelInput to the sending NetChannelOutput. This takes the form (ACK, <destination>, -1).
REJECT_CHANNEL	8	Sent to a Node to indicate that a previous SEND message was rejected at the receiving Node for some reason. This takes the form (REJECT_CHANNEL, <destination>, -1).
POISON	12	A message sent to indicate that a channel end should poisoned. The message needs to indicate the destination and the strength of the poison. This takes the form (POISON, <destination>, <strength>).
ASYNC_SEND	13	An unacknowledged send message. This is kept in for legacy reasons at present, and will likely be removed in the future. This takes the form (ASYNC_SEND, <destination>, <source>, <size>, <bytes>).

E.2 Barrier Messages

Message	Value	Description
SYNC	5	A synchronisation from a NetBarrier client to a NetBarrier server. This takes the form (SYNC, <destination>, <source>).
RELEASE	6	A message sent from the NetBarrier server end to a NetBarrier client end to indicate that all client ends have synchronised and can now continue. This takes the form (RELEASE, <destination>, -1).
ENROLL	3	Sent from a NetBarrier client end to a NetBarrier server end to indicate that it is joining the set of synchronising NetBarrier ends. This takes the form (ENROLL, <destination>, -1).
RESIGN	4	Sent from a NetBarrier client end to a NetBarrier server end to indicate that it is resigning from the set of synchronising NetBarrier ends. This takes the form (RESIGN, <destination>, -1).
REJECT_BARRIER	7	Sent to a Node to indicate that a previous barrier message was rejected at the receiving Node for some reason. This can be sent to both server and client NetBarrier ends. This takes the form (REJECT_BARRIER, <destination>, -1).

E.3 Connection Messages

Message	Value	Description
OPEN	14	Opens a connection communication from a client connection end to a server connection end. This includes a data message that is the initial request to the connection. This takes the form (OPEN, <destination>, <source>, <size>, <bytes>).
REQUEST	15	A request data message sent from the connection client end to the connection server end. This is basically the same as the OPEN message, but is used when an ongoing communication is occurring. This takes the form (REQUEST, <destination>, <source>, <size>, <bytes>).
REPLY	16	A reply data message sent from the connection server end to the connection client end. This takes the form (REPLY, <destination>, <source>, <size>, <bytes>).
REPLY_AND_CLOSE	17	A reply message sent from the server end to the connection end, and also closes the communication. This takes the form (REPLY_AND_CLOSE, <destination>, <source>, <size>, <bytes>).
REQUEST_ACK	22	An acknowledgement sent from the server connection end to the client end. This message is used to acknowledge both OPEN and REQUEST messages. It takes the form (REQUEST_ACK, <destination>, -1).
REPLY_ACK	23	An acknowledgement sent from the client connection end to the server end. This message is used to acknowledge both REPLY and REPLY_AND_CLOSE messages. It takes the form (REPLY_ACK, <destination>, -1).
ASYNC_OPEN	18	An unacknowledged open connection message. As any connection interaction takes the form request-reply, there is no risk of infinite buffer increasing as the NetConnectionClient must call reply before performing another ASYNC_OPEN. This takes the form (ASYNC_OPEN, <destination>, <source>, <size>, <bytes>).
ASYNC_REQUEST	19	An unacknowledged request message. This takes the form (ASYNC_REQUEST, <destination>, <source>, <size>, <bytes>).
ASYNC_REPLY	20	An unacknowledged reply message. This takes the form (ASYNC_REPLY, <destination>, <source>, <size>, <bytes>).
ASYNC_REPLY_AND_CLOSE	21	An unacknowledged reply message that closes the connection. (ASYNC_REPLY_AND_CLOSE, <destination>, <source>, <size>, <bytes>).
REJECT_CONNECTION	24	Sent to a Node to indicate that a previous connection message was rejected. This can be sent from both the client and server end. It takes the form (REJECT_CONNECTION, <destination>, -1).

E.4 Miscellaneous Messages

Message	Value	Description
LINK_LOST	9	Sent to an event component to indicate that the Link it was operating on has failed for some reason. All event component types use this message in some form. A NetChannelOutput is sent LINK_LOST to indicate that the Link to the NetChannelInput has failed. The NetBarrier client end is sent this message to indicate that the connection to the server end has gone, and therefore the NetBarrier fails. The NetBarrier server end receives this message when one of its client ends has become disconnected, and the server end should act accordingly. A connection end receives this message if the other end of the connection has gone. A client end is broken, whereas the server end may continue after disassociating itself from the client.

Appendix F SPIN Model of New JCSP Network Architecture

```
/* Define the possible states of the channel.
   These are set constants. */

#define INACTIVE   0
#define OK_INPUT 1
#define OK_OUTPUT 2
#define DESTROYED 3
#define BROKEN 4
#define POISONED 5

/* Define return values from a call on a channel.
   These are set constants. */

#define OK 1
#define EXCEPTION 0

/* Define number of of input channels, number outputs to inputs
   and buffer size of the channels. */

#define NUMBER_INPUTS 1
#define NUMBER_OUTPUTS 1
#define TOTAL_OUTPUTS 1
#define BUFFER_SIZE 1

/* Protocol definition */

mtype = {
  SEND,    /* A standard send message to a ChannelInput */
  ACK,     /* An acknowledgement for a SEND */
  REJECT_CHANNEL,    /* Rejection of a channel message */
  LINK_LOST,    /* Link to Node lost */
  POISON};      /* Poison message */

/* ********** TYPE DEFINITIONS ********** */

/* The channel data state object.  This retains information
about a channel to allow operation. */

typedef CHANNEL_DATA
{
  byte vcn;    /* The Virtual Channel Number of the channel.
                  Used to uniquelly identify the channel in
                  the Node. */
  byte state = INACTIVE;    /* The current state of the channel.
                               Initially the channel is set to
                               INACTIVE. */
  chan toChannel;    /* The channel connecting the Link level with
                        the Net Channel. The Net Channel reads
                        from this channel, whereas the Link writes
                        to it. */
};
```

```
/* The input channel interface.  We bundle this together to make
   things easier. */

typedef INPUT_CHANNEL_INTERFACE
{
  /* Channel written to when a read operation occurs */
  chan read = [0] of { bool };
  /* Channel written to when an extended read operation is begun */
  chan startRead = [0] of { bool };
  /* Channel written to when an extended read operation is
     completed */
  chan endRead = [0] of { bool };
  /* Channel written to when the channel is poisoned by the
     application */
  chan poison = [0] of { bool };
  /* Channel written to when the channel is destroyed by the
     application */
  chan destroy = [0] of { bool };
  /* Channel read from to simulate the return from the method
     call */
  chan callReturn = [0] of { bit };
};

/* The output channel interface.  We bundle this together to make
   things easier */

typedef OUTPUT_CHANNEL_INTERFACE
{
  /* Channel written to when a write operation occurs */
  chan write = [0] of { bool };
  /* Channel written to when the channel is poisoned by
     the application */
  chan poison = [0] of { bool };
  /* Channel written to when the channel is destroyed by
     the application */
  chan destroy = [0] of { bool };
  /* Channel read from to simulate the return from the method
     call */
  chan callReturn = [0] of { bit };
};

/* The channels declared for a Node */

typedef CHANNEL_ARRAY
{
  CHANNEL_DATA channels[TOTAL_OUTPUTS];
};

/* ********** GLOBALS ********** */

/* Global flag used to signal link failure to the Application
   processes.  We use this to get round the problem if a Link fails
   while a channel input is committed in a read operation. We
   essentially know that this will cause the application process to
   break, as it has no one to read from.  However, this should not
   be seen as a fault of the architecture, as the net channels are
   any-2-one, so another connection might later be established.
   The flag is false and is set to true when the Network fails, and
   the input channels can safely die. */

byte linkLost = false;

/* The channels for each Node.  The writing Node has one, and the
   reading Node also has one */

CHANNEL_ARRAY chans[2];
```

```
/* ********** PROCESSES ********** */

/* A network connection, simulated by a process to allow network
   failure and network buffering to occur. */

proctype Network(chan in0; chan in1; chan out0; chan out1)
{
  /* Network reads in message type and two attributes. */
  mtype type;
  byte attr1;
  byte attr2;
  /* Either read in a message and output it, or break
     the connection */
  /* Valid end state for network.  Waiting to stream message */
end_network:
  do
  /* Stream message */
  :: atomic
     {
       in0 ? type, attr1, attr2 -> out1 ! type, attr1, attr2
     }
  /* Stream message */
  :: atomic
     {
       in1 ? type, attr1, attr2 -> out0 ! type, attr1, attr2
     }
  /* Non deterministically choose to break link.  Connection
     down */
  :: atomic
     {
       linkLost = true ->
       /* Send connected Nodes the LINK_LOST message */
       out0 ! LINK_LOST(-1, -1);
       out1 ! LINK_LOST(-1, -1);
       /* End network process */
     }
     break;
  od
}

/* A NetChannelOutput */

proctype NetChannelOutput(OUTPUT_CHANNEL_INTERFACE interface;
                          chan toLinkTx; chan ackChannel;
                          CHANNEL_DATA data; byte remoteVCN)
{
  /* The response from the Link */
  mtype response;

  /* Valid end state for output channel.  Channel is awaiting a
     method call */
end_nco:
  do
  /* Write operation */
  :: interface.write ? _ ->
      /* Check the channel state first, and return exception if
         necessary */
    if
    :: atomic
       {
         (data.state == DESTROYED) ->
           interface.callReturn ! EXCEPTION
       }
    :: atomic
       {
         (data.state == BROKEN) ->
           interface.callReturn ! EXCEPTION
       }
```

```
:: atomic
   {
     (data.state == POISONED) ->
       interface.callReturn ! EXCEPTION
   }
/* Otherwise channel in OK state.  Continue write operation */
:: else ->
     if
     /* Check if there are any residual messages.  The
        channel may have been rejected, or the Link might
        have failed */
     :: atomic
        {
          (nempty(ackChannel)) ->
            /* There is a residual message.  Process it. */
            ackChannel ? response;
            if
            /* A previous send was rejected.  Break channel */
            :: (response == REJECT_CHANNEL) ->
                 data.state = BROKEN;
                 interface.callReturn ! EXCEPTION
            /* The network has gone down.  Break channel */
            :: (response == LINK_LOST) ->
                 data.state = BROKEN;
                 interface.callReturn ! EXCEPTION
            /* Poison received.  Poison channel */
            :: (response == POISON) ->
                 data.state = POISONED;
                 interface.callReturn ! EXCEPTION
            /* Otherwise an unexpected message has been
               received.  Fail */
            :: else -> assert(false)
            fi
        }
     /* There are no residual messages.  Continue write
        operation. */
     :: else ->
          /* Send message to the Link.  Destination at
             receiving Node is remoteVCN and local
             sender is data.vcn */
          toLinkTx ! SEND(remoteVCN, data.vcn);
          /* Now wait for a response */
          atomic
          {
            ackChannel ? response;
            /* Now process the message */
            if
            /* SEND was rejected.  Break channel */
            :: (response == REJECT_CHANNEL) ->
                 data.state = BROKEN;
                 interface.callReturn ! EXCEPTION
            /* Network has gone down.  Return exception */
            :: (response == LINK_LOST) ->
                 data.state = BROKEN;
                 interface.callReturn ! EXCEPTION
            /* Channel has been poisoned. */
            :: (response == POISON) ->
                 data.state = POISONED;
                 interface.callReturn ! EXCEPTION
            /* ACK received. */
            :: (response == ACK) ->
                 interface.callReturn ! OK
            /* Unexpected message.  Fail */
            :: else -> assert(false)
            fi
          }
     fi;
fi; /* End of write operation */
```

```
  /* The application has poisoned the channel */
  :: interface.poison ? _ ->
       /* First check the status of the channel.  If the channel is
          BROKEN, DESTROYED, or POISONED we simply return */
       if
       :: atomic
          {
            (data.state == DESTROYED) ->
              interface.callReturn ! OK
          }
       :: atomic
          {
            (data.state == BROKEN) ->
              interface.callReturn ! OK
          }
       :: atomic
          {
            (data.state == POISONED) ->
              interface.callReturn ! OK
          }
       /* Otherwise channel in OK state.  Continue with poisoning */
       :: else ->
            /* Check for pending messages.  This could mean the
               channel is broken, in which case poisoning can
               be ignored */
            if
            :: atomic
               {
                 (nempty(ackChannel)) ->
                   /* There is a residual message. Process it. */
                   ackChannel ? response;
                   if
                   /* A previous send was rejected. Break channel,
                      and return OK */
                   :: (response == REJECT_CHANNEL) ->
                        data.state = BROKEN;
                        interface.callReturn ! OK
                   /* The network has gone down.  Break channel and
                      return OK. */
                   :: (response == LINK_LOST) ->
                        data.state = BROKEN;
                        interface.callReturn ! OK
                   /* Poison received.  Poison channel as this was
                      going to happen anyway, and then return */
                   :: (response == POISON) ->
                        data.state = POISONED;
                        interface.callReturn ! OK
                   /* Otherwise an unexpected message has been
                      received.  Fail */
                   :: else -> assert(false)
                   fi
               }
            /* There are no residual messages.  Continue poison
               operation. */
            :: atomic
               {
                  empty(ackChannel) ->
                    /* Set state to POISONED, send poison to input
                       end, and return */
                    data.state = POISONED;
                    toLinkTx ! POISON(remoteVCN, -1);
                    interface.callReturn ! OK
               }
            fi;
       fi; /* End of poison operation */
```

```
  /* Channel is being destroyed by the application */
  :: interface.destroy ? _ ->
       /* First check the status of the channel.  We may not need to
          do anything */
       if
       :: atomic
          {
            (data.state == DESTROYED) ->
              interface.callReturn ! OK
          }
       :: atomic
          {
            (data.state == BROKEN) ->
              data.state = DESTROYED;
              interface.callReturn ! OK
          }
       :: atomic
          {
            (data.state == POISONED) ->
              data.state = DESTROYED;
              interface.callReturn ! OK
          }
       /* Otherwise we need to set the state to destroyed, remove
          any pending messages, and return OK */
       :: else ->
            atomic
            {
              data.state = DESTROYED;
              do
              :: (nempty(ackChannel)) -> ackChannel ? response
              :: (empty(ackChannel)) -> break
              od;
              interface.callReturn ! OK
            }
       fi; /* End of destroy operation */
  od
}

/* A networked input channel */

proctype NetChannelInput(INPUT_CHANNEL_INTERFACE interface;
                         chan fromLink; CHANNEL_DATA data)
{
  /* Variables used to store incoming message */
  mtype type;
  byte returnIdx;
  chan toLink;

  /* Flag to indicate if the channel is in an extended read state.
     In the implementation we are actually testing for nullity of
     the last read message, but this flag serves the same purpose */
  bool extended = false;
  /* Valid end state for channel input.  Channel waiting for
     application call */
end_nci:
  do
  /* Read operation */
  :: interface.read ? _ ->
      /* First check state of channel and act accordingly */
      if
      :: atomic
         {
           (data.state == DESTROYED) ->
             interface.callReturn ! EXCEPTION
         }
      :: atomic
         {
           (data.state == POISONED) ->
             interface.callReturn ! EXCEPTION
         }
```

```
      :: atomic
         {
           (data.state == BROKEN) ->
             interface.callReturn ! EXCEPTION
         }
      :: atomic
         {
           extended -> interface.callReturn ! EXCEPTION
         }
      /* Otherwise continue read operation */
      :: else ->
           if
           /* Read in next message */
           :: atomic
              {
                 fromLink ? type, returnIdx, toLink ->
                   /* Check message type and act accordingly */
                   if
                   /* SEND received.  Complete read operation */
                   :: (type == SEND) ->
                        /* Send back ACK message */
                        toLink ! ACK(returnIdx, -1);
                        /* Return OK */
                        interface.callReturn ! OK;
                   /* Poison received.  Poison the channel */
                   :: (type == POISON) ->
                        /* Change state to poisoned */
                        data.state = POISONED;
                        /* Send poison to any incoming messages */
                        do
                        :: (nempty(fromLink)) ->
                             /* There is an incoming message.
                                Process it */
                             fromLink ? type, returnIdx, toLink;
                             if
                             :: (type == SEND) ->
                                  toLink ! POISON(returnIdx, -1)
                             :: else -> skip
                             fi;
                        :: (empty(fromLink)) ->
                             break /* No more pending messages */
                        od;
                        /* Return EXCEPTION */
                        interface.callReturn ! EXCEPTION;
                   /* Otherwise an unknown message has been
                      received.  Fail */
                   :: else -> assert(false)
                   fi;
              }
            /* All incoming Links are gone.  Read cannot complete,
               so set state to broken and return */
           :: atomic
              {
                linkLost ->
                  data.state = BROKEN;
                  interface.callReturn ! EXCEPTION
              }
           fi;
      fi; /* End of read operation */
/* Start extended read operation */
:: interface.startRead ? _ ->
      /* First check the state of the channel and act
         accordingly */
      if
      :: atomic
         {
           (data.state == DESTROYED) ->
             interface.callReturn ! EXCEPTION
         }
```

```
      :: atomic
         {
           (data.state == POISONED) ->
             interface.callReturn ! EXCEPTION
         }
      :: atomic
         {
           (data.state == BROKEN) ->
             interface.callReturn ! EXCEPTION
         }
      :: atomic
         {
           extended -> interface.callReturn ! EXCEPTION
         }
      /* Otherwise continue extended read operation */
      :: else ->
           if
           /* Read in next message */
           :: atomic
              {
                fromLink ? type, returnIdx, toLink ->
                  /* Check message type and act accordingly */
                  if
                  /* SEND received.  Complete read operation */
                  :: (type == SEND) ->
                       /* Set extended to true and return */
                       extended = true;
                       /* Return OK */
                       interface.callReturn ! OK;
                  /* Poison received.  Poison the channel */
                  :: (type == POISON) ->
                       /* Change state to poisoned */
                       data.state = POISONED;
                       /* Send poison to any incoming messages */
                       do
                       :: (nempty(fromLink)) ->
                            /* There is an incoming message.
                               Process it */
                            fromLink ? type, returnIdx, toLink;
                            if
                            :: (type == SEND) ->
                                 toLink ! POISON(returnIdx, -1)
                            :: else -> skip
                            fi;
                       :: (empty(fromLink)) ->
                            break /* No more pending messages */
                       od;
                       /* Return EXCEPTION */
                       interface.callReturn ! EXCEPTION;
                       /* Otherwise an unknown message has been
                          received.  Fail */
                  :: else -> assert(false)
                  fi;
              }
           /* All incoming Links are gone.  Extended read cannot
              complete, so set state to broken and return */
           :: atomic
              {
                linkLost ->
                  data.state = BROKEN;
                  interface.callReturn ! EXCEPTION
              }
           fi;
      fi; /* End of start read operation */
```

```
  /* End extended read operation */
  :: interface.endRead ? _ ->
       /* First check state of channel. */
       if
       :: atomic
          {
            (data.state == DESTROYED) ->
              interface.callReturn ! EXCEPTION
          }
       :: atomic
          {
            (data.state == POISONED) ->
              interface.callReturn ! EXCEPTION
          }
       :: atomic
          {
            (data.state == BROKEN) ->
              interface.callReturn ! EXCEPTION
          }
       :: atomic
          {
            (!extended) ->
              interface.callReturn ! EXCEPTION
          }
       /* Otherwise send acknowledgement message */
       :: atomic
          {
            else ->
              extended = false;
              toLink ! ACK(returnIdx, -1);
              interface.callReturn ! OK
          }
       fi; /* End of end extended read operation */
  /* Poison channel operation */
  :: interface.poison ? _ ->
       /* First check the state of the channel.  Nothing may need
          to be done */
       if
       :: atomic
          {
            (data.state == DESTROYED) -> interface.callReturn ! OK
          }
       :: atomic
          {
            (data.state == POISONED) -> interface.callReturn ! OK
          }
       :: atomic
          {
            (data.state == BROKEN) -> interface.callReturn ! OK
          }
       /* Otherwise continue poison operation */
       :: else ->
            atomic
            {
              /* If the channel is extended, the previous message
                 needs to be acked by poison */
              if
              :: extended -> toLink ! POISON(returnIdx, -1)
              :: else -> skip
              fi;
              /* Set state to poisoned and process any pending
                 messages */
              /* Change state to poisoned */
              data.state = POISONED;
              /* Send poison to any incoming messages */
```

```
              do
              :: (nempty(fromLink)) ->
                   /* There is an incoming message.  Process it */
                   fromLink ? type, returnIdx, toLink;
                   if
                   :: (type == SEND) ->
                        toLink ! POISON(returnIdx, -1)
                   :: else -> skip
                   fi;
              :: (empty(fromLink)) ->
                   break /* No more pending messages */
              od;
              /* Return OK */
              interface.callReturn ! OK;
            }
       fi; /* End of poison operation */
  /* Destroy channel operation operation */
  :: interface.destroy ? _ ->
       /* First check the state of the channel.  Nothing may need to
          be done */
       if
       :: atomic
          {
            (data.state == DESTROYED) -> interface.callReturn ! OK
          }
       :: atomic
          {
            (data.state == POISONED) ->
              data.state = DESTROYED;
              interface.callReturn ! OK
          }
       :: atomic
          {
            (data.state == BROKEN) ->
              data.state = DESTROYED;
              interface.callReturn ! OK
          }
       /* Otherwise continue destroy operation */
       :: else ->
            atomic
            {
              /* If the channel is extended, the previous message
                 needs to be rejected */
              if
              :: extended -> toLink ! REJECT_CHANNEL(returnIdx, -1)
              :: else -> skip
              fi;
              /* Set state to destroyed and process any pending
                 messages */
              /* Change state to destroyed */
              data.state = DESTROYED;
              /* Send rejection to any incoming messages */
              do
              :: (nempty(fromLink)) ->
                   /* There is an incoming message.  Process it */
                   fromLink ? type, returnIdx, toLink;
                   if
                   :: (type == SEND) ->
                        toLink ! REJECT_CHANNEL(returnIdx, -1)
                   :: else -> skip
                   fi;
              :: (empty(fromLink)) ->
                   break /* No more pending messages */
              od;
              /* Return OK */
              interface.callReturn ! OK;
            }
       fi; /* End of destroy operation */
  od
}
```

```
/* The TX process of the Link */

proctype LinkTx(chan input; chan txStream)
{
  mtype type;
  byte attr1;
  byte attr2;

  /* The point of this process is to forward whatever message it
     receives onto the network *EXCEPT* when the network has gone
     down.  We simulate this with the linkFailed flag, so the LinkTx
     must also check this flag when it tries to send */
  /* Valid end state for LinkTx.  Waiting for message to forward */
end_ltx1:
  do
  :: input ? type, attr1, attr2 ->
       if
       /* Network still up, we can send */
       :: txStream ! type, attr1, attr2
       /* Network is down. We now accept any incoming messages, but
          do not forward them onto the stream. The sender should be
          informed by the LinkRx.  Valid end state, waiting for
          message to black hole */
       :: linkLost ->
end_ltx2:
            do
            :: input ? type, attr1, attr2
            od
       fi
  od
}

/* The RX process of the Link */

proctype LinkRx(chan toTxProcess; chan rxStream; bit nodeNumber)
{
  /* Attributes read in with incoming message */
  byte attr1;
  byte attr2;

  /* This process reads an incoming message from the message and
     processes it.  Generally the message is forwarded onto the
     correct destination, although erroneous behaviour must be dealt
     with */
  /* Valid end state.  Waiting for input from the network */
end_lrx:
  do
  /* SEND received.  */
  :: atomic
     {
       rxStream ? SEND(attr1, attr2) ->
         /* First check if the message is going to a valid
            channel */
         if
         /* Destination channel is outside range.  Reject message */
         :: (attr1 > TOTAL_OUTPUTS) ->
              toTxProcess ! REJECT_CHANNEL(attr2, -1)
         :: else ->
              /* Message is for a valid channel.  Check channel
                 state and deal with accordingly */
              if
              :: (chans[nodeNumber].channels[attr1].state
                                              == OK_INPUT) ->
                   /* Channel is OK to receive messages.  Forward
                      the message onto the channel process */
                   chans[nodeNumber].channels[attr1].toChannel
                                    ! SEND(attr2, toTxProcess)
```

```
            :: (chans[nodeNumber].channels[attr1].state
                                          == POISONED) ->
                 /* Channel has been poisoned.  Propagate the
                    poison back to the writer */
                 toTxProcess ! POISON(attr2, 0)
            :: (chans[nodeNumber].channels[attr1].state
                                         == DESTROYED) ->
                 /* Channel has been destroyed.  Reject the
                    message */
                 toTxProcess ! REJECT_CHANNEL(attr2, 0)
            :: (chans[nodeNumber].channels[attr1].state
                                           == BROKEN) ->
                 /* Channel is broken.  This should only happen
                    during Link failure, but be safe and reject */
                 toTxProcess ! REJECT_CHANNEL(attr2, 0)
            :: else ->
                 /* Channel is in some other state.  This could be
                    a channel trying to send to an output or some
                    other problem.  We reject the message in this
                    instance and continue */
                 toTxProcess ! REJECT_CHANNEL(attr2, 0)
            fi
       fi
   }
/* Acknowledgement operation */
:: atomic
   {
     rxStream ? ACK(attr1, attr2) ->
       /* First check if the message is going to a valid
          channel */
       if
       /* Destination channel is outside range.  Ignore message */
       :: (attr1 > TOTAL_OUTPUTS) -> skip
       :: else ->
            /* Message is for a valid channel.  Check channel
               state and deal with accordingly */
            if
            :: (chans[nodeNumber].channels[attr1].state
                                         == OK_OUTPUT) ->
                 /* ACK being sent to an output channel.  Forward
                    the message onto the channel process */
                 chans[nodeNumber].channels[attr1].toChannel ! ACK
                 /* In all other cases, we drop the message. The
                    message has been sent to a channel that was
                    not in a state to accept it. */
            :: else -> skip
            fi
       fi
   }
/* Reject channel message received */
:: atomic
   {
     rxStream ? REJECT_CHANNEL(attr1, attr2) ->
       /* First check if the message is going to a valid
          channel */
       if
       /* Destination channel is outside range.  No point in
          rejecting (we could end up with a continuous cycle of
          rejects). Simply ignore the message. */
       :: (attr1 > TOTAL_OUTPUTS) -> skip
       :: else ->
            /* Message is for a valid Channel.  Check channel
               state and deal with accordingly */
            if
            :: (chans[nodeNumber].channels[attr1].state
                                         == OK_OUTPUT) ->
                 /* Channel can accept the reject message.  Pass
                    onto the channel process */
                 chans[nodeNumber].channels[attr1].toChannel
                                          ! REJECT_CHANNEL
```

```
              /* In all other cases ignore the message.  The channel
                 is in no state to receive it */
              :: else -> skip
              fi
        fi
    }
  /* Poison message received */
  :: atomic
    {
      rxStream ? POISON(attr1, attr2) ->
        /* First check if the message is going to a valid
           channel */
        if
        /* Destination channel is outside range.  No point in
           rejecting (we could end up with a continuous cycle of
           rejects). Simply ignore the message. */
        :: (attr1 > TOTAL_OUTPUTS) -> skip
        :: else ->
             /* Message is for a valid Channel.  Check channel
                state and deal with accordingly */
             if
             :: (chans[nodeNumber].channels[attr1].state
                                          == OK_OUTPUT) ->
                  /* Channel is an output.  Simply send POISON to
                     it. */
                  chans[nodeNumber].channels[attr1].toChannel
                                                  ! POISON
             :: (chans[nodeNumber].channels[attr1].state
                                           == OK_INPUT) ->
                  /* Channel is an input.  Simply send POISON to
                     it */
                  chans[nodeNumber].channels[attr1].toChannel
                                        ! POISON(attr1, attr2)
             /* In all other cases we ignore the poison. Either the
                channel is poisoned, and in the Model nothing else
                needs to be done (in the implementation we increase
                the poison strength if necessary), or it is
                destroyed or broken, which is considered to be
                greater than poison */
             :: else -> skip
             fi
        fi
    }
  /* Link lost received */
  :: rxStream ? LINK_LOST(attr1, attr2) ->
       atomic
       {
         /* Inform all output ends */
         byte idx = 0;
         do
         :: (idx < TOTAL_OUTPUTS) ->
              if
              :: (chans[nodeNumber].channels[idx].state
                                         == OK_OUTPUT) ->
                   chans[nodeNumber].channels[idx].toChannel
                                                 ! LINK_LOST
              :: else -> skip
              fi;
              idx = idx + 1;
         :: else -> break
         od;
       }
       break;
  od;
}
```

```
/* The complete Link process */

proctype Link(chan toLinkTx; chan toNetwork; chan fromNetwork;
              bit nodeNumber)
{
  atomic
  {
    run LinkRx(toLinkTx, fromNetwork, nodeNumber);
    run LinkTx(toLinkTx, toNetwork);
  }
}

/* A receiving application process */

proctype Receiver(INPUT_CHANNEL_INTERFACE chanIn)
{
  /* Response from the method call */
  bit response;
  /* Non deterministically choose an operation to perform on the
     channel */
  /* Valid end state for the process */
end_receiver:
  do
  :: atomic
     {
       if
       :: chanIn.read ! true -> chanIn.callReturn ? response
       :: chanIn.startRead ! true -> chanIn.callReturn ? response
       :: chanIn.endRead ! true -> chanIn.callReturn ? response
       :: chanIn.poison ! true -> chanIn.callReturn ? response
       :: chanIn.destroy ! true -> chanIn.callReturn ? response
       fi;
       if
       :: (response == EXCEPTION) -> goto end_receiverStop
       :: else -> skip
       fi
     }
  Od;
end_receiverStop:
  skip
}

/* A sending application process */

proctype Sender(OUTPUT_CHANNEL_INTERFACE chanOut)
{
  /* Response from the method call */
  bit response;
  /* Non deterministically choose an operation to perform on
     the channel */
  /* Valid end state for the process */
end_sender:
  do
  :: atomic
     {
       if
       :: chanOut.write ! true -> chanOut.callReturn ? response
       :: chanOut.poison ! true -> chanOut.callReturn ? response
       :: chanOut.destroy ! true -> chanOut.callReturn ? response
       fi;
       if
       :: (response == EXCEPTION) -> goto end_senderStop
       :: else -> skip
       fi
     }
  od;
end_senderStop:
    skip
}
```

```
/* Inputting Node process */

proctype InputNode(chan toNetwork; chan fromNetwork)
{
  atomic
  {
    chan toLinkTx = [0] of { mtype, byte, byte };
    /* Set up the Link */
    run Link(toLinkTx, toNetwork, fromNetwork, 0);
    /* Set up the channel and receiving processes */
    INPUT_CHANNEL_INTERFACE inputInterface[NUMBER_INPUTS];
    chan toInput[NUMBER_INPUTS] =
      [BUFFER_SIZE] of { mtype, byte, chan };
    byte idx = 0;
    do
    :: (idx < NUMBER_INPUTS) ->
         chans[0].channels[idx].vcn = idx;
         chans[0].channels[idx].state = OK_INPUT;
         chans[0].channels[idx].toChannel = toInput[idx];
         run NetChannelInput(inputInterface[idx], toInput[idx],
                             chans[0].channels[idx]);
         run Receiver(inputInterface[idx]);
         idx = idx + 1;
    :: else -> break;
    od;
    /* Set any other channels to INACTIVE. */
    do
    :: (idx < TOTAL_OUTPUTS) ->
         chans[0].channels[idx].state = INACTIVE;
         idx = idx + 1
    :: else -> break
    Od
  }
}

/* Outputting Node process */

proctype OutputNode(chan toNetwork; chan fromNetwork)
{
  atomic
  {
    chan toLinkTx = [0] of { mtype, byte, byte };
    /* Set up the Link */
    run Link(toLinkTx, toNetwork, fromNetwork, 1);
    /* Set up the channel and receiving processes */
    OUTPUT_CHANNEL_INTERFACE outputInterface[TOTAL_OUTPUTS];
    chan toOutput[TOTAL_OUTPUTS] = [BUFFER_SIZE] of { mtype };
    byte idx = 0, count = 0;
    do
    :: (idx < TOTAL_OUTPUTS) ->
         do
         :: (count < NUMBER_INPUTS) ->
              chans[1].channels[idx].vcn = idx;
              chans[1].channels[idx].state = OK_OUTPUT;
              chans[1].channels[idx].toChannel = toOutput[idx];
              run NetChannelOutput(outputInterface[idx], toLinkTx,
                                   toOutput[idx],
                                   chans[1].channels[idx], count);
              run Sender(outputInterface[idx]);
              idx = idx + 1;
              count = count + 1;
         :: else ->
              count = 0;
              break;
         od;
    :: else -> break;
    Od
  }
}
```

```
/* initialisation */

init
{
  atomic
  {
    chan fromNode[2] = [0] of { mtype, byte, byte };
    chan toNode[2] = [0] of { mtype, byte, byte };
    run Network(fromNode[0], fromNode[1], toNode[0], toNode[1]);
    run OutputNode(fromNode[0], toNode[0]);
    run InputNode(fromNode[1], toNode[1]);
  }
}
```

Appendix G Channel Mobility Models

The state model for channels in the new JCSP architecture is presented in Figure 41. For all the models presented, channel states which would cause an exception if the channel were migrated are ignored (*DESTROYED, POISONED* and *BROKEN)*.

G.1 One-to-One Networked Channel

This model for channel mobility is based on the work of Muller [145]. As a channel must only have one complement, a networked channel input must be claimed by the output end. The one-to-one channel model requires presented by Muller distinguishes between remotely and locally connected channels for optimisation reasons. JCSP cannot provide optimisation from this standpoint, although a locally connected networked channel does connect directly to the complement end. Refining the proposed model in this context provides the state model presented in Figure 155.

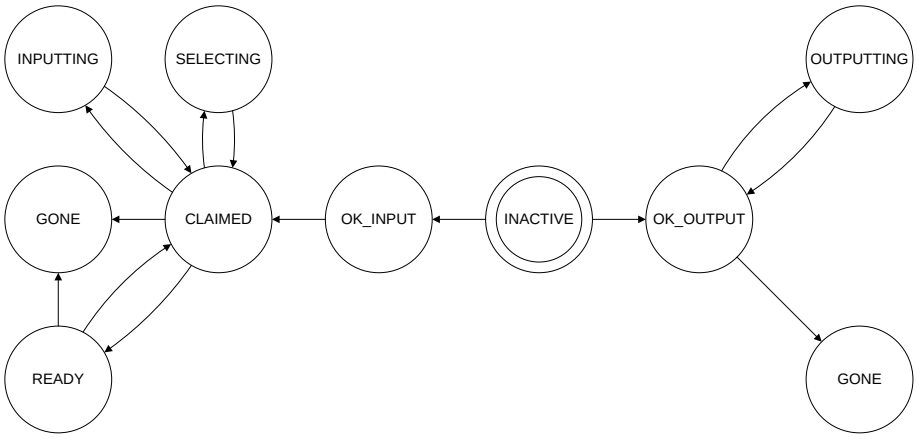


Figure 155: One-to-One Networked Channel Mobility Model State Diagram

There are six new states introduced:

- *CLAIMED* – an input channel that has an output channel associated with it.

- *SELECTING* – the input channel is currently being used in a guarded command.
- *INPUTTING* – the input channel is waiting for input to arrive.
- *OUTPUTTING* – the output channel has sent a message to the input channel and is awaiting acknowledgement.
- *READY* – the input channel has data ready to be read.
- *GONE* – the channel end has been moved to another node.

The basic operations for moving a channel end are shown in the sequence diagram Figure 156. The channel connects C to A. The first operation shows the message transfer if the input end moves from A to B, and the second operation shows the message transfer if the output end moves from C to B.

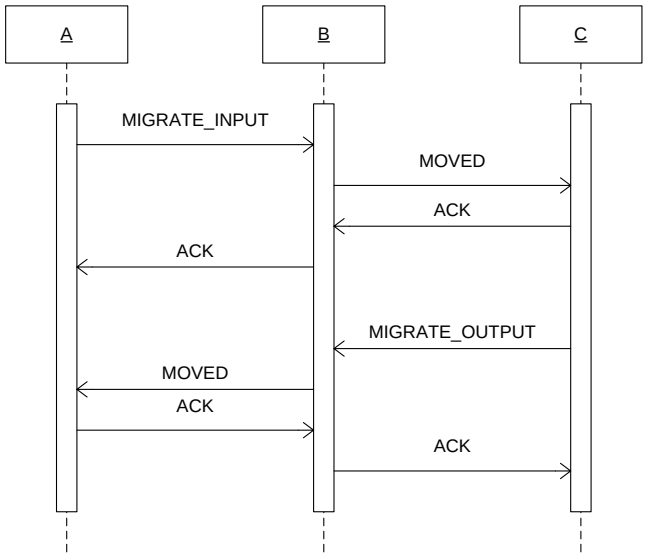


Figure 156: Sequence Diagram for One-to-One Networked Channel Mobility Model

There are three new channel messages introduced:

- *MIGRATE_INPUT* – signals that an input channel has been moved. This message must contain the address of the companion output port.
- *MIGRATE_OUTPUT* – signals that an output channel has been moved. This message must contain the address of the companion input port.
- *MOVED* – the message sent by the receiving node of a mobile channel end to the companion ports location. This message must contain the new address of the complement channel end.

Figure 156 does not show the occurrence when an input channel moves when data is waiting to be read (it is in the *READY* state). In this situation, the output end of the channel must resend the message to the new channel location. The other approach would be to include it within the data segment of the *MIGRATE_INPUT* message. This would require the data segment to contain both the address and the data, and have a method to flag that data is also contained. The simpler approach is therefore to have the output end resend the message.

G.2 Centralised Server

The centralised server approach to channel mobility is the approach currently taken in pony [120, 130]. It has also been discussed as a communication method for agent based systems. To achieve mobility, each channel end is allocated an identifier by the server, and this identifier is used to check the current location of the channel end whenever it is not found at its current location. The updated state model only requires one new state – *GONE* – to indicate that the channel end is no longer at that location. The state diagram is presented in

Figure 157: Centralised Server Mobility Model State Diagram

The operation of moving an input channel end and the output end subsequently trying to send to the original location, and thereby requiring resolution of the new location with the server is presented in Figure 158, with the channel in question connecting C to A. Output end mobility is trivial in that in only involves sending the identifier of the input channel end so that the current address can be resolved with the server.

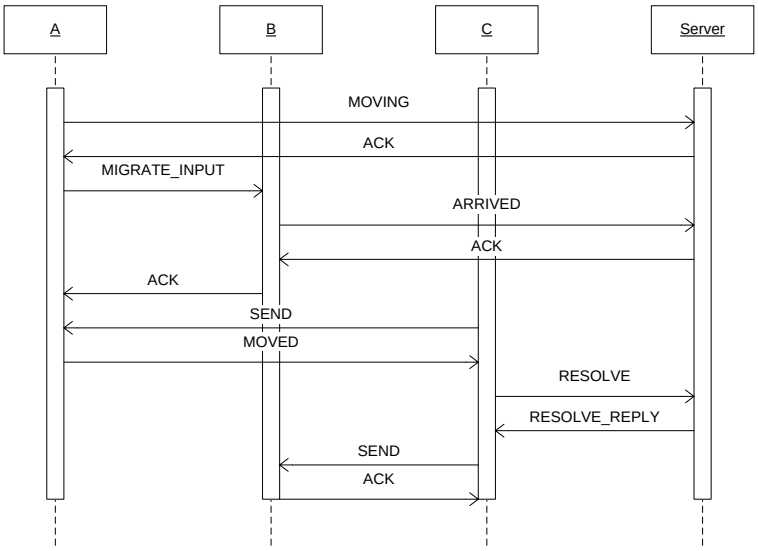


Figure 158: Sequence Diagram for Centralised Server Mobility Model

There are seven new messages introduced:

- *MOVING* – is sent from the input channel end location to the Server to indicate that a channel is about to move. After receiving this message, the Server changes the state of the channel and buffers any resolution messages for this channel ID.
- *MIGRATE_INPUT* – signals that an input end has moved. This need only contain the ID of the channel relevant to the Server.
- *MIGRATE_OUTPUT (not shown)* – signals that an output end has moved. This need only contain the ID of the input end of the channel relevant to the Server.
- *ARRIVED* – sent from the receiver of an input channel end to the Server to indicate the new location.
- *MOVED* – sent to an output end instead of an acknowledgement to indicate that the input channel end has moved.
- *RESOLVE* – sent to the Server to acquire the current location of the input channel end. This contains the ID of input channel to resolve.
- *RESOLVE_REPLY* – the reply from the RESOLVE. This contains the current address of the input channel.

G.3 Message Box

The message box is another model that is commonly used in mobile agent frameworks. With this approach, messages are always sent and retrieved from a single location. As the message box is fixed, there is no need to add new states to the channel model itself as when the channel end is moved the state *DESTROYED* can be used to signify that the channel can no longer be used. The message box itself does require a state model, but this only consists of two states: *ENABLED* when the message box is enabled in a guard, and *DISABLED* for when the message box is not enabled in a guard.

Figure 159 presents the sequence of messages that can occur within this mobility model. This diagram represents a mobile input end and subsequent request and response messages. The diagram also illustrates how a channel is checked to see if a value is ready in the message box. Mobile output is not shown as it only requires the address of the message box to be sent.

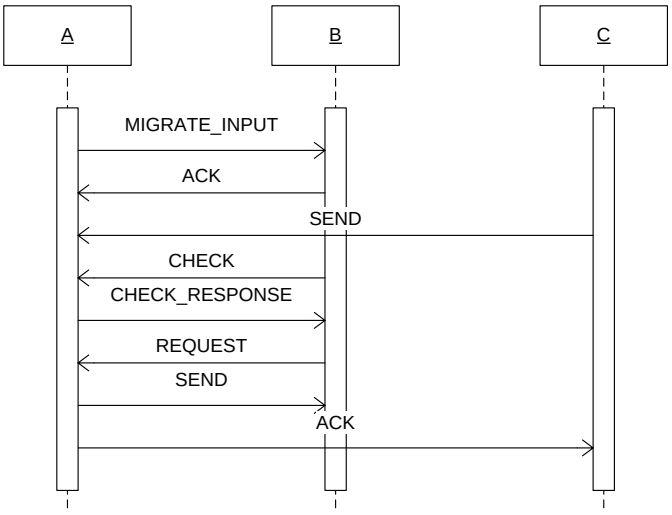


Figure 159: Sequence Diagram for Message Box Mobility Model

There are five new messages incorporated into this model:

- *MIGRATE_INPUT* – signals that an input end has moved. This message contains the address of the message box to allow messages to be requested.
- *MIGRATE_OUTPUT* – signals that an output end has moved. This message contains the address of the message box where messages are to be sent.

- *CHECK* – sent by the input end to check if any messages are waiting in the message box. This message contains the current location of the input channel end to respond to.
- *CHECK_RESPONSE* – sent from the message box to the input end in response to the *CHECK*. There is an immediate response, and also a delayed one. If a message arrives after a *CHECK* but prior to a *REQUEST* then a message is sent to the input channel end and buffered. If the input channel end moves or requests prior to the response being utilised, the message can be silently dropped. If it does not move, then the locally buffered response can be retrieved instead of sending a new *CHECK*.
- *REQUEST* – requests the next available message from the message box. This message contains the current location of the input end.

G.4 Message Box Server

The message box server model places all message boxes on a centralised server, thus removing the weakness associated with distributed message boxes. This model does not add much in comparison to the normal message box approach. The only extra messages required involve creation and destruction of the message box with the server. As such, Figure 159 serves to illustrate the sequence of messages, with requests and sends being directed to a Server instead of a particular node.

G.5 Chain

The chain model of mobility utilises forwarding addresses to allow messages to reach their intended destination. A migrating input end must inform the previous location of the new channel input address. Output end mobility only requires that the address of the previous link in the chain is taken to allow connection to that link. Thus there are really multiple chains of different length that eventually connect at the original output location. As the normal buffering and reply technique for network messages is used, acknowledgements will flow back in the direction that the original message travelled.

The state diagram for this model is presented in Figure 160.

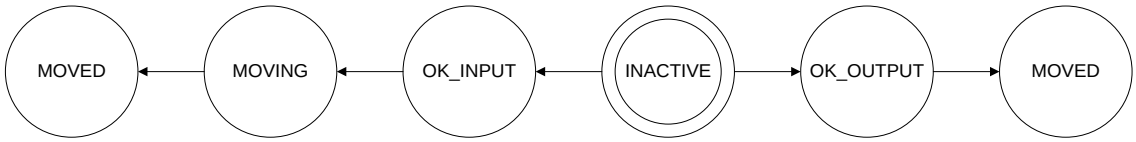


Figure 160: Chain Mobility Model State Diagram

There are two new states introduced:

- *MOVING* – indicates that the input channel end is in the process of moving. In this state the node where the input channel end is located must buffer messages until the channel has arrived at its new location.
- *MOVED* – indicates that the channel end has arrived at a new location. Any incoming message is forwarded to the next link in the chain and buffered to allow the response to likewise travel back down the chain to the origin channel end.

Reconfiguration of the architecture to allow the channel to move is fairly trivial. The sequence of required messages is illustrated in Figure 161. There are only three new messages introduced:

- *MIGRATE_INPUT* – indicates that an input channel end has moved. This message contains the previous input address to allow the chain to be expanded.
- *MIGRATE_OUTPUT* – indicates that an output channel end has moved. This message contains the previous output address to allow the chain to be expanded.
- *ARRIVED* – indicates that the input end has arrived at a new location. This message contains the new input channel end location to allow channel to forward on messages accordingly.

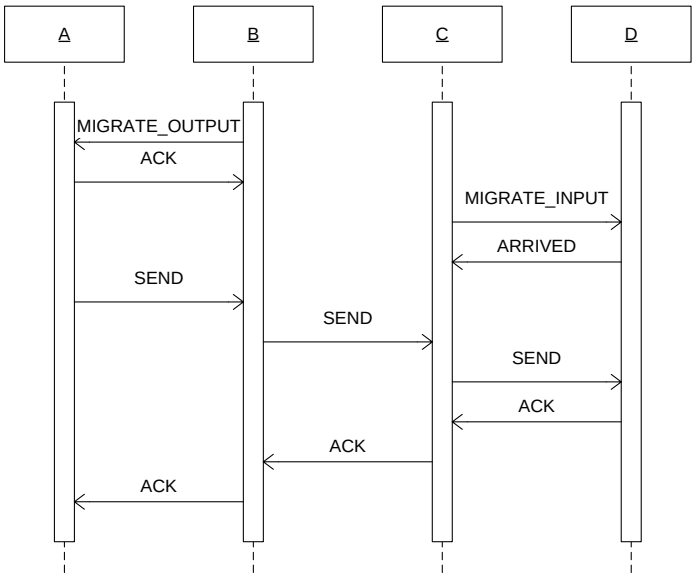


Figure 161: Sequence Diagram for Chain Mobility Model

G.6 Reconfiguring Chain

The reconfiguring chain model attempts to overcome the major limitations of the normal chain model by allowing the chain to shorten itself by checking if any other link in the chain is directly accessible instead of the immediately previous link. As a channel has two separate mobile ends, there are essentially two separate chains that can shorten. If the shared output view is taken for networked channels, then there are multiple such reconfiguring output chains. This does mean that although the chain may be shortcut at the output end, all previous links in the chain must remain in case another mobile output end cannot use the new direct connection. The previous output ends could be shut down by sending poison down the redundant input sub-chain.

The state diagram for this model of channel mobility is the same as the one for the normal chain. The sequence of messages occurring during a migration operation is presented in Figure 162. C is the original location of the channel input end.

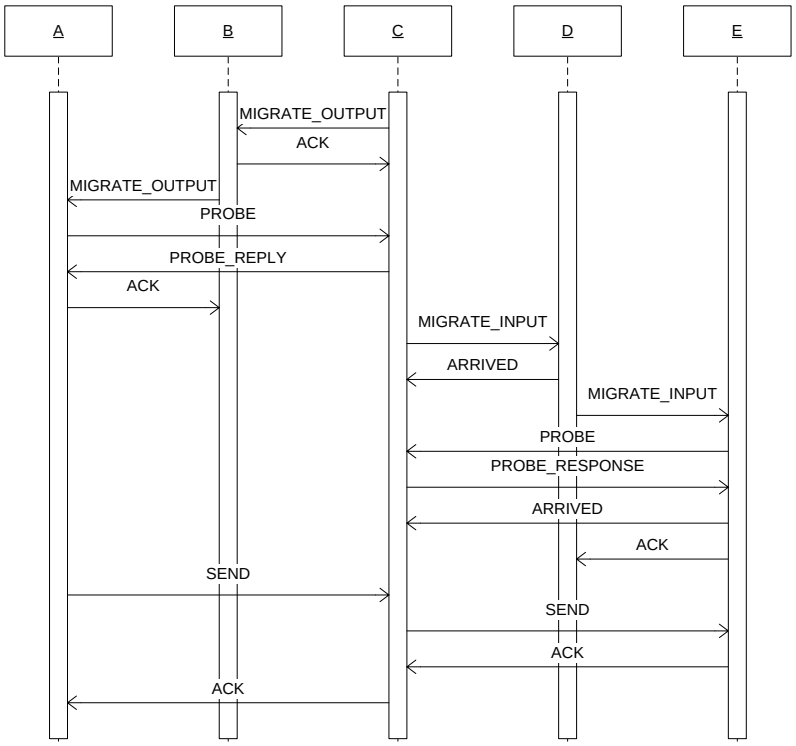


Figure 162: Sequence Diagram for Reconfiguring Chain Mobility Model

There are five new messages:

- *MIGRATE_OUTPUT* – indicates that a channel out end has moved. This message must contain all previous locations in the chain to allow reconnection.
- *MIGRATE_INPUT* – indicates that a channel input end has moved. This message must contain all previous locations in the chain to allow reconnection.
- *ARRIVED* – sent from the new location of an input channel end to a previous location in the chain. This might be to the immediately previous link, or it may be further down the chain.
- *PROBE* – sent from the new location of a channel end to one of the previous locations. This is used to shorten the chain whenever possible. In some cases, the *PROBE* will never be sent as the relevant link will not be reachable.
- *PROBE_RESPONSE* – sent in reply to the *PROBE* message. This is used to indicate that a direct path between the previous location and the new one does exist.

PROBE and *PROBE_RESPONSE* may occur multiple times depending on the number of previous locations that are in the chain. A migrated input end may receive an *ACK* message or an *ARRIVED* message as acknowledgement depending on whether the link is to be used, or whether a shortcut has been created.

G.7 Mobile IP Model

The Mobile IP model [146] utilises agents within each domain to allow messages to be routed to the correct destination. In effect, there is a reconfiguring chain between domain servers which utilise message boxes. This model is sufficiently more complex than the previous models to require numerous scenarios of mobility. These are presented in sequence diagram form.

The scenario is best illustrated using a small domain tree. This is presented in Figure 163. *G* is the global domain, and *A* and *B* are two sub-domains with no means of direct contact. *A* has two members – A_1 and A_2 – and *B* has two members – B_1 and B_2.

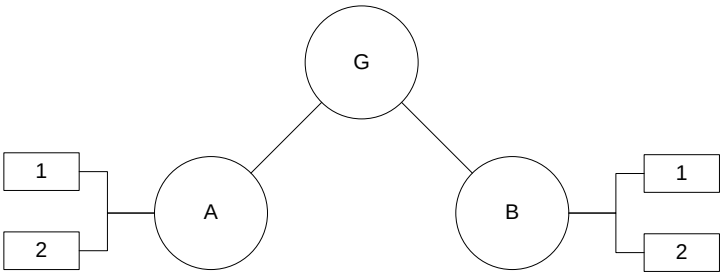


Figure 163: Simple Domain Tree

The scenario to be illustrated originally has a channel connecting A_1 to A_2 ($mobile_1$), and one connecting B_1 to B_2 ($mobile_2$). The input channel end of $mobile_1$ is moved from node A_2 to node B_2. Then the output end of $mobile_1$ is moved from A_1 to B_1. For $mobile_2$ the output channel end is moved first from node B_1 to A_1 and then the input end from B_2 to A_2.

Each agent in the domain maintains a lookup table of where to redirect messages to. Initially, there are two channels connecting A_1 and B_1 (one in each direction) and likewise two channels connecting A_2 and B_2. How these were created is

inconsequential. The three separate lookup tables are merged into one and presented in Table 14.

From this table it becomes possible to define the paths of all the channels in the system.

- *mobile*$_1$ – $A_1 \rightarrow$ A #01 $\rightarrow A_2$ (although these nodes are directly linked)
- *mobile*$_2$ – $B_1 \rightarrow$ B #01 $\rightarrow B_2$ (although these nodes are directly linked)
- A_1B_1 – $A_1 \rightarrow$ A #04 $\rightarrow$ G #03 $\rightarrow$ B #02 $\rightarrow B_1$
- A_2B_2 – $A_2 \rightarrow$ A #05 $\rightarrow$ G #04 $\rightarrow$ B #03 $\rightarrow B_2$
- B_1A_1 – $B_1 \rightarrow$ B #04 $\rightarrow$ G #01 $\rightarrow$ A #02 $\rightarrow A_1$
- B_2A_2 – $B_2 \rightarrow$ B #05 $\rightarrow$ G #02 $\rightarrow$ A #03 $\rightarrow A_2$

Table 14: Initial Channel Destination Table

DOMAIN	CHANNEL ID	DESTINATION	PREVIOUS
A	#01	A_2	-
A	#02	A_1	G #01
A	#03	A_2	G #02
A	#04	G #03	-
A	#05	G #04	-
B	#01	B_2	-
B	#02	B_1	G #03
B	#03	B_2	G #04
B	#04	G #01	-
B	#05	G #02	-
G	#01	A #02	B #04
G	#02	A #03	B #05
G	#03	B #02	A #04
G	#04	B #03	A #05

G.7.1 Sending a New Input Channel End

To send the input end of *mobile*$_1$, a MIGRATE_INPUT message is sent via A_2B_2. The message must have the ID of the channel relevant to the domain (i.e. A #01), the previous location if relevant, and the normal source and destination (destination being A #05, source is the VCN of A_2B_2 on node *2* – represented by *a*). Thus, the message takes the form:

MIGRATE_INPUT | A #05 | *a* | A #01 | -1

When the agent for domain A receives this message, it first examines the channel ID to determine which channel it is moving. As the channel has no previous location, the agent knows that the channel is unknown at the destination. It buffers the message in the normal buffer for channel A #05, and recreates the message accordingly to tunnel it through to the agent for domain G:

MIGRATE_INPUT | G #04 | A #05 | -1 | A #01

When the agent at node G receives this message, it too examines the channel ID. As this time the value is -1, it knows that the channel is new in this domain context, and creates a new entry:

G	#05	-	A #01

As the agent does not know the destination of this channel yet, the destination field is left blank. The agent now sends an *ARRIVED* signal back to the agent for domain A with the new destination:

ARRIVED | A #01 | G #05

The first attribute is the destination of the message, and the second is the new ID destination of the channel on the domain agent for G. The agent for A now updates this field in its table:

A	#01	G #05	-

The agent for domain G now modifies the migration message:

MIGRATE_INPUT | B #03 | G #04 | -1 | G #05

When the agent for domain B receives this message, it checks the channel ID and finds it to be -1. Therefore the channel is new in this context and a new entry is created accordingly.

B	#06	-	G #05

An arrived message is generated, and sent back to the agent for domain G.

ARRIVED | G #05 | B #06

The agent for B now modifies the migration message and sends it to node B_2 on, using the relevant VCN – represented by *b*:

MIGRATE_INPUT | *b* | B #02 | -1 | B #05

When node B_2 receives this message, it creates a new channel and sends the *ARRIVAL* message back to the agent for the domain. When the receiving process reads the message, it is given the newly created channel, and the ACK is sent back down the path. The new channel path is now:

$mobile_1$ – A_1 → A #01 → G #05 → B #06 → B_2

G.7.2 Sending the Complement Output End

To send the output end of $mobile_1$, a *MIGRATE_OUTPUT* message is sent via A_1B_1. As with the *MIGRATE_INPUT* channel, a channel ID must be sent relevant to the domain. However, this time the ID is where the output channel end is pointing – which is A #01. The previous location is not filled in, as it is not required for shortening the connection. Let the VCN of A_1B_1 on node A_1 be represented by *c*. Thus, the message created for the send is:

MIGRATE_OUTPUT | A #04 | *c* | A #01 | -1

When the agent for domain *A* receives this message, it retrieves the destination of the message (G #03) and the current destination of the migrating channel (G #05). As these destinations are on the same domain – *G* – the agent determines the chain is shortening and it does not have to update its table. It buffers the sent message for future acknowledgement, and creates a new *MIGRATE_OUTPUT* message, using the next destination link of the migrating output. As there is no previous location, there is no need to send an *ARRIVED* message.

MIGRATE_OUTPUT | G #03 | A #04 | G #05 | -1

When the agent at domain *G* receives this message, it too extracts the destination (B #02) and the destination of the migrating end (B #06). As these are on the same node, and there is no previous location, there is no table updates required. The message is buffered, and a new *MIGRATE_OUTPUT* generated:

MIGRATE_OUTPUT | B #02 | G #03 | B #06 | -1

When the agent of domain *B* receives this message, it retrieves the destination (B_1) and the destination of the migrating end (B_2). As both of these destinations are within this domain, the domain specific ID is used for the final *MIGRATE_OUTPUT* message to B_1. Thus the message sent to B_1 (let the destination VCN be *d*) is:

MIGRATE_OUTPUT | *d* | B #02 | B #06 | -1

The updated channel path for $mobile_1$ is:

$mobile_1$ – B_1 ⟶ B #06 ⟶ B_2

When the new channel is first used, B_1 can connect directly to B_2.

G.7.3 Sending a New Output End

To send the output end of $mobile_2$, a new *MIGRATE_OUTPUT* is sent via B_1A_1. The message has the channel ID currently connected to:

MIGRATE_OUTPUT | B #04 | *e* | B #01 | -1

e is the VCN of B_1A_1 on node B_1. When the agent for domain *B* receives this message, it retrieves the destination of the message (G #01) and the destination of the migrating end (B_2). As the destination of the migrating end is within this domain, whereas the destination of the message is not, then the agent checks the previous destination of the channel to see if the channel has previously come from the destination domain. As it has not, the agent knows that the channel is new within *G*, and creates a new message with -1 as the channel ID at the destination domain, and B #01 as the previous destination.

MIGRATE_OUTPUT | G #01 | B #04 | -1 | B #01

When the agent for domain *G* receives this message, it extracts the next destination (A #02) and the migrating end's destination (-1). As the migrating end has -1 as its destination, the agent knows the channel is new in this context, and thus creates a new entry in its table, with the destination from previous destination attribute:

G	#06	B #01	-

The agent for domain *G* then sends an *ARRIVED* message to the agent on domain *B*:

ARRIVED | B #01 | G #06

The agent for domain *B* uses this message to fill the previous location entry for channel B #01:

B	#01	B_2	G #06

The agent then creates a new *MIGRATE_OUTPUT* message:

MIGRATE_OUTPUT | A #02 | G #01 | -1 | G #06

When the agent for domain *A* receives this message, it extracts the next destination (A_1) and the destination of the migrating end (-1). As the migrating end has -1 as its destination, the agent at *A* knows this is a new channel in this context. A new entry in the table is created, and an *ARRIVED* message sent to the agent for domain *G*, updating the entry for G #06:

A	#06	G #06	-

ARRIVED | G #06 | A #06

The final *MIGRATE_OUTPUT* is sent to A_1, with the location of the relevant channel ID:

MIGRATE_OUTPUT | *f* | A #02 | A #06 | -1

The updated channel path for $mobile_2$ is:

$mobile_2$ – A_1 → A #06 → G #06 → B #01 → B_2

G.7.4 Sending the Complement Input End

Sending the input end of $mobile_2$ requires a *MIGRATE_INPUT* message sent via B_2A_2. This message requires the relevant channel ID within the domain (B #01). The previous location is not relevant initially:

MIGRATE_INPUT | B #05 | *g* | B #01 | -1

When the agent for domain *B* receives this message, it extracts the previous destination from the table for the channel ID (G #06). As the channel has a previous location, the agent checks the destination (G #02) and discovers them to be the same. Thus, the agent determines that the table must be updated so that B #01 points towards G #06 instead of from. As the previous destination attribute is -1, it is determined that there is no new previous destination to be set for the entry:

B	#01	G #06	-

The agent then creates a new *MIGRATE_INPUT* message and sends it to the agent for domain *G*:

MIGRATE_INPUT | G #02 | B #05 | G #06 | B #01

When the agent for domain *G* receives this message, it extracts the channel ID, and from this the previous destination of the channel (A #06). As the channel has a previous location, the agent checks the destination (A #03) and as they are on the same node, determines that the channel must be redirected. As the previous destination attribute has a value, the previous destination value for the entry is set to the current destination of the channel.

G	#06	A #06	B #01

An *ARRIVED* message is generated and sent to the agent for domain *B* allowing the agent to forward messages onto the new destination. The agent then creates a new *MIGRATE_INPUT* message to send to the agent for domain *A*:

MIGRATE_INPUT | A #03 | G #02 | A #06 | G #06

When the agent for domain *A* receives this message, it retrieves the relevant previous destination of the channel which is empty. Therefore, the channel is unknown at the destination. The agent checks the destination, and finds it to be for domain *A* (A_2). Therefore, the table is updated so that A #06 points towards A_2. As the previous destination has a value, the previous destination for A #06 is set to the current destination (G #06):

A	#06	A_2	G #06

The agent informs the agent for domain *G* that the channel has arrived, and then sends the final *MIGRATE_INPUT* message to A_2:

MIGRATE_INPUT | *g* | A #03 | -1 | A #06

g is the VCN of B_2A_2 on node A_2. The new path of $mobile_2$ is:

$mobile_2$ – $A_1 \rightarrow$ A #06 $\rightarrow A_2$

G.7.5 Protocol Messages

From these examples, we can determine that three new messages are required:

- *MIGRATE_INPUT* – sent when an input channel migrates. This contains two addresses or ID constructs, identifying the channel and its previous location.
- *MIGRATE_OUTPUT* – sent when an output channel migrates. This contains two addresses or ID constructs, indentifying the channel and its previous location.
- *ARRIVED* – sent to indicate a channel end has arrived at its destination. This contains the old address and new address of the channel.

Appendix H Numbers and Mobile Numbers Processes

The standard versions of these processes are taken from the JCSP release, and where originally created by P. D. Austin.

H.1 IdentityInt

H.1.1 Normal

```
public class IdentityInt implements CSProcess
{
    private ChannelInputInt in;
    private ChannelOutputInt out;

    public IdentityInt(ChannelInputInt in, ChannelOutputInt out)
    {
        this.in = in;
        this.out = out;
    }

    public void run()
    {
        while (true)
            out.write(in.read());
    }
}
```

H.1.2 Mobile

```
public class MobileIdentityInt implements CSProcess, Serializable
{
    private static final int READING = 0;
    private static final int WRITING = 1;
    private transient AltingChannelInputInt input;
    private transient AltingChannelOutputInt output;
    private transient AltingBarrier migrate;
    private int state = READING;
    private int x;

    public void init(AltingChannelInputInt input,
                     AltingChannelOutputInt output,
                     AltingBarrier migrate)
    {
        this.input = input;
        this.output = output;
        this.migrate = migrate;
    }
```

```
    public MobileIdentityInt(AltingChannelInputInt input,
                             AltingChannelOutputInt output,
                             AltingBarrier migrate)
    {
        this.input = input;
        this.output = output;
        this.migrate = migrate;
    }

    private void writeObject(ObjectOutputStream out)
    throws IOException
    {
        out.writeInt(state);
        out.writeInt(x);
    }

    private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
    {
        this.state = in.readInt();
        this.x = in.readInt();
    }

    public void run()
    {
        Guard[] guards = {migrate, input, output};
        Alternative alt = new Alternative(guards);
        boolean running = true;
        while (running)
        {
            switch (state)
            {
            case READING:
            {
                // migrate and input
                boolean[] active = {true, true, false};
                int selected = alt.priSelect(active);
                switch (selected)
                {
                case 0: // migrate
                    running = false;
                    break;
                case 1: // input
                    x = input.read();
                    state = WRITING;
                    break;
                }
            }
            break;
            case WRITING:
            {
                // migrate and output
                boolean[] active = {true, false, true};
                int selected = alt.priSelect(active);
                switch (selected)
                {
                case 0: // migrate
                    running = false;
                    break;
                case 2: // output
                    output.write(x);
                    state = READING;
                    break;
                }
            }
            break;
            }
        }
    }
}
```

H.2 PrefixInt

H.2.1 Normal

```
public class PrefixInt implements CSProcess
{
    private ChannelInputInt in;
    private ChannelOutputInt out;
    private int n;

    public PrefixInt(int n, ChannelInputInt in,
                     ChannelOutputInt out)
    {
        this.in = in;
        this.out = out;
        this.n = n;
    }

    public void run()
    {
        out.write(n);
        new IdentityInt(in, out).run();
    }
}
```

H.2.2 Mobile

```
public class MobilePrefixInt implements CSProcess, Serializable
{
    private static final int WRITING = 0;
    private static final int IDENTITY = 1;
    private int state = WRITING;
    private int prefix = 0;
    private MobileIdentityInt identity;
    private transient AltingChannelInputInt input;
    private transient AltingChannelOutputInt output;
    private transient AltingBarrier migrate;

    public void init(AltingChannelInputInt input,
                     AltingChannelOutputInt output,
                     AltingBarrier migrate)
    {
        this.input = input;
        this.output = output;
        this.migrate = migrate;
        this.identity.init(input, output, migrate);
    }

    public MobilePrefixInt(int prefix, AltingChannelInputInt input,
                           AltingChannelOutputInt output,
                           AltingBarrier migrate)
    {
        this.prefix = prefix;
        this.input = input;
        this.output = output;
        this.migrate = migrate;
        this.identity =
            new MobileIdentityInt(input, output, migrate);
    }

    private void writeObject(ObjectOutputStream out)
    throws IOException
    {
        out.writeInt(state);
        out.writeInt(prefix);
        out.writeObject(identity);
    }
```

```
    private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
    {
        this.state = in.readInt();
        this.prefix = in.readInt();
        this.identity = (MobileIdentityInt)in.readObject();
    }

    public void run()
    {
        boolean running = true;
        Guard[] guards = {migrate, output};
        Alternative alt = new Alternative(guards);
        while (running)
        {
            switch (state)
            {
            case WRITING:
                int selected = alt.priSelect();
                switch (selected)
                {
                case 0:
                    running = false;
                    break;
                case 1:
                    output.write(prefix);
                    state = IDENTITY;
                    identity.run();
                    break;
                }
                break;
            case IDENTITY:
                identity.run();
                running = false;
            }
        }
    }
}
```

H.3 SuccessorInt

H.3.1 Normal

```
public class SuccessorInt implements CSProcess
{
    private ChannelInputInt in;
    private ChannelOutputInt out;

    public SuccessorInt(ChannelInputInt in, ChannelOutputInt out)
    {
        this.in = in;
        this.out = out;
    }

    public void run()
    {
        while (true)
            out.write(in.read() + 1);
    }
}
```

H.3.2 Mobile

```
public class MobileSuccessorInt implements CSProcess, Serializable
{
    private static final int READING = 0;
    private static final int WRITING = 1;
    private int lastRead = 0;
    private int state = READING;
    private transient AltingChannelInputInt input;
    private transient AltingChannelOutputInt output;
    private transient AltingBarrier migrate;

    public void init(AltingChannelInputInt input,
                     AltingChannelOutputInt output,
                     AltingBarrier migrate)
    {
        this.input = input;
        this.output = output;
        this.migrate = migrate;
    }

    public MobileSuccessorInt(AltingChannelInputInt input,
                              AltingChannelOutputInt output,
                              AltingBarrier migrate)
    {
        this.input = input;
        this.output = output;
        this.migrate = migrate;
    }

    private void writeObject(ObjectOutputStream out)
    throws IOException
    {
        out.writeInt(state);
        out.writeInt(lastRead);
    }

    private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
    {
        state = in.readInt();
        lastRead = in.readInt();
    }

    public void run()
    {
        boolean running = true;
        Guard[] guards = {migrate, input, output};
        Alternative alt = new Alternative(guards);
        while (running)
        {
            switch (state)
            {
            case READING:
            {
                boolean[] active = {true, true, false};
                int selected = alt.priSelect(active);
                switch (selected)
                {
                case 0:
                   running = false;
                   break;
                case 1:
                   lastRead = input.read();
                   state = WRITING;
                   break;
                }
                break;
            }
```

```
            case WRITING:
            {
                boolean[] active = {true, false, true};
                int selected = alt.priSelect(active);
                switch (selected)
                {
                case 0:
                    running = false;
                    break;
                case 2:
                    int toOutput = lastRead + 1;
                    output.write(toOutput);
                    state = READING;
                    break;
                }
                break;
            }
            }
        }
    }
}
```

H.4 ProcessWriteInt

H.4.1 Normal

```
public class ProcessWriteInt implements CSProcess
{
    public int value;
    private ChannelOutputInt out;

    public ProcessWriteInt(ChannelOutputInt out)
    {
        this.out = out;
    }

    public void run()
    {
        out.write(value);
    }
}
```

H.4.2 Mobile

```
public class MobileProcessWriteInt
implements CSProcess, Serializable
{
    private static final int WRITING = 0;
    private static final int FINISHED = 1;
    private transient AltingChannelOutputInt output;
    private transient AltingBarrier migrate;
    private transient AltingBarrier finished;
    public int value = 0;
    private int state = WRITING;

    public void init(AltingChannelOutputInt out,
                     AltingBarrier migrate,
                     AltingBarrier finished)
    {
        this.output = out;
        this.migrate = migrate;
        this.migrate.resign();
        this.finished = finished;
    }
```

```
    public MobileProcessWriteInt(AltingChannelOutputInt out,
                                 AltingBarrier migrate,
                                 AltingBarrier finished)
    {
        this.output = out;
        this.migrate = migrate;
        this.migrate.resign();
        this.finished = finished;
    }

    private void writeObject(ObjectOutputStream out)
    throws IOException
    {
        out.writeInt(state);
        out.writeInt(value);
    }

    private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
    {
        this.state = in.readInt();
        this.value = in.readInt();
    }

    public void run()
    {
        Guard[] guards = {migrate, output, finished};
        Alternative alt = new Alternative(guards);
        boolean running = true;
        while (running)
        {
            switch (state)
            {
            case WRITING:
            {
                boolean[] active = {true, true, false};
                int selected = alt.priSelect(active);
                switch (selected)
                {
                case 0:
                    running = false;
                    break;
                case 1:
                    output.write(value);
                    state = FINISHED;
                    break;
                }
            }
            break;
            case FINISHED:
            {
                boolean[] active = {true, false, true};
                int selected = alt.priSelect(active);
                switch (selected)
                {
                case 0:
                    running = false;
                    break;
                case 2:
                    running = false;
                    state = WRITING;
                    break;
                }
            }
            break;
            }
        }
    }
}
```

H.5 Delta2Int

H.5.1 Normal

```
public class Delta2Int implements CSProcess
{
    private ChannelInputInt  in;
    private ChannelOutputInt out0;
    private ChannelOutputInt out1;

    public Delta2Int(ChannelInputInt in, ChannelOutputInt out0,
                     ChannelOutputInt out1)
    {
        this.in   = in;
        this.out0 = out0;
        this.out1 = out1;
    }

    public void run()
    {
        ProcessWriteInt[] parWrite = {new ProcessWriteInt(out0),
                                      new ProcessWriteInt(out1)};
        Parallel par = new Parallel(parWrite);
        while (true)
        {
            int value = in.read();
            parWrite[0].value = value;
            parWrite[1].value = value;
            par.run();
        }
    }
}
```

H.5.2 Mobile and CheckFinished

```
public class MobileDelta2Int implements CSProcess, Serializable
{
    private static final int READING = 0;
    private static final int WRITING = 1;
    private transient AltingChannelInputInt input;
    private transient AltingChannelOutputInt out0;
    private transient AltingChannelOutputInt out1;
    private transient AltingBarrier migrate;
    private int lastRead = 0;
    private int state = READING;
    private MobileProcessWriteInt[] procs;

    public void init(AltingChannelInputInt input,
                     AltingChannelOutputInt out0,
                     AltingChannelOutputInt out1,
                     AltingBarrier migrate)
    {
        this.input = input;
        this.out0 = out0;
        this.out1 = out1;
        this.migrate = migrate;
    }
```

```
    public MobileDelta2Int(AltingChannelInputInt input,
                           AltingChannelOutputInt out0,
                           AltingChannelOutputInt out1,
                           AltingBarrier migrate)
    {
        this.migrate = migrate;
        this.input = input;
        this.out0 = out0;
        this.out1 = out1;
    }

    public void writeObject(ObjectOutputStream out)
    throws IOException
    {
        out.writeInt(state);
        out.writeInt(lastRead);
        out.writeObject(procs);
    }

    public void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
    {
        this.state = in.readInt();
        this.lastRead = in.readInt();
        this.procs = (MobileProcessWriteInt[])in.readObject();
    }

    public void run()
    {
        AltingBarrier[] barriers = migrate.expand(2);
        AltingBarrier[] finished = AltingBarrier.create(3);
        if (procs == null)
        {
            procs = new MobileProcessWriteInt[2];
            procs[0] =
                new MobileProcessWriteInt(out0, barriers[0],
                                          finished[0]);
            procs[1] =
                new MobileProcessWriteInt(out1, barriers[1],
                                          finished[1]);
        }
        else
        {
            procs[0].init(out0, barriers[0], finished[0]);
            procs[1].init(out1, barriers[1], finished[1]);
        }
        Guard[] guards = {migrate, input};
        Alternative alt = new Alternative(guards);
        CheckFinished check =
            new CheckFinished(migrate, finished[2]);
        CSProcess[] processes = {procs[0], procs[1], check};
        Parallel par = new Parallel(processes);
        boolean running = true;
        while (running)
        {
            switch (state)
            {
            case READING:
            {
                int selected = alt.priSelect();
                switch (selected)
                {
                case 0:
                    running = false;
                    break;
                case 1:
                    lastRead = input.read();
                    state = WRITING;
                    break;
                }
```

```
                break;
            }
            case WRITING:
            {
                procs[0].value = lastRead;
                procs[1].value = lastRead;
                barriers[0].enroll();
                barriers[1].enroll();
                par.run();
                if (!check.isFinished)
                {
                    running = false;
                    break;
                }
                else
                {
                    state = READING;
                    barriers[0].resign();
                    barriers[1].resign();
                }
            }
            }
        }
        par.releaseAllThreads();
    }
}

public class CheckFinished implements CSProcess
{
    private AltingBarrier migrate;
    private AltingBarrier finished;
    public boolean isFinished = false;

    public CheckFinished(AltingBarrier migrate,
                         AltingBarrier finished)
    {
        this.migrate = migrate;
        this.finished = finished;
    }

    public void run()
    {
        Guard[] guards = {migrate, finished};
        Alternative alt = new Alternative(guards);
        isFinished = false;
        int selected = alt.priSelect();
        if (selected != 0)
            isFinished = true;
    }
}
```

H.6 NumbersInt

H.6.1 Normal

```
public class NumbersInt implements CSProcess
{
    private ChannelOutputInt out;

    public NumbersInt(ChannelOutputInt out)
    {
        this.out = out;
    }
```

```
    public void run()
    {
        One2OneChannelInt a = ChannelInt.createOne2One();
        One2OneChannelInt b = ChannelInt.createOne2One();
        One2OneChannelInt c = ChannelInt.createOne2One();
        new Parallel(new CSProcess[]
                     {
                         new Delta2Int(a.in(), b.out(), out),
                         new SuccessorInt(b.in(), c.out()),
                         new PrefixInt(0, c.in(), a.out())
                     }).run();
    }
}
```

H.6.2 Mobile

```
public class MobileNumbersInt implements CSProcess, Serializable
{
    private transient AltingBarrier migrate;
    private transient AltingBarrier innerMigrate;
    private boolean localMigrate = false;
    private transient AltingChannelOutputInt output;
    private MobileSuccessorInt succ;
    private MobilePrefixInt pre;
    private MobileDelta2Int delta;

    public void init(AltingChannelOutputInt output)
    {
        this.output = output;
        AltingBarrier[] bars = AltingBarrier.create(2);
        this.migrate = bars[0];
        this.innerMigrate = bars[1];
    }

    public void init(AltingChannelOutputInt output,
                     AltingBarrier barrier)
    {
        this.output = output;
        this.migrate = barrier;
    }

    public MobileNumbersInt(AltingChannelOutputInt output,
                            AltingBarrier migrate)
    {
        this.output = output;
        this.migrate = migrate;
    }

    public MobileNumbersInt(AltingChannelOutputInt output)
    {
        this.output = output;
        AltingBarrier[] bars = AltingBarrier.create(2);
        this.migrate = bars[0];
        this.innerMigrate = bars[1];
    }

    private void writeObject(ObjectOutputStream out)
    throws IOException
    {
        if (innerMigrate != null)
            innerMigrate.sync();
        out.writeObject(succ);
        out.writeObject(pre);
        out.writeObject(delta);
    }
```

```
    private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
    {
        this.succ = (MobileSuccessorInt)in.readObject();
        this.pre = (MobilePrefixInt)in.readObject();
        this.delta = (MobileDelta2Int)in.readObject();
    }

    public void run()
    {
        AltingBarrier[] barriers = migrate.expand(2);
        One2OneChannelSymmetricInt a =
            Channel.one2oneSymmetricInt();
        One2OneChannelSymmetricInt b =
            Channel.one2oneSymmetricInt();
        One2OneChannelSymmetricInt c =
            Channel.one2oneSymmetricInt();
        if (succ == null)
        {
            succ = new MobileSuccessorInt(a.in(), b.out(),
                                          barriers[0]);
            pre = new MobilePrefixInt(0, c.in(), a.out(),
                                      barriers[1]);
            delta = new MobileDelta2Int(b.in(), output, c.out(),
                                        migrate);
        }
        else
        {
            this.succ.init(a.in(), b.out(), barriers[0]);
            this.pre.init(c.in(), a.out(), barriers[1]);
            this.delta.init(b.in(), output, c.out(), migrate);
        }
        CSProcess[] processes = {succ, pre, delta};
        Parallel par = new Parallel(processes);
        par.run();
        par.releaseAllThreads();
    }
}
```

Appendix I Published Work

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

K. Chalmers and S. Clayton, "CSP for .NET Based on JCSP," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), Communicating Process Architectures 2006, pp. 59-76, IOS Press, Amsterdam, 2006.

K. Chalmers, J. Kerridge, and I. Romdhani, "Performance Evaluation of JCSP Micro Edition: JCSPme," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), Communicating Process Architectures 2006, pp. 31-40, IOS Press, Amsterdam, 2006.

J. Kerridge and K. Chalmers, "Ubiquitous Access to Site Specific Services," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), Communicating Process Architectures 2006, pp. 41-58, IOS Press, Amsterdam, 2006.

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.

K. Chalmers, I. Romdhani, and J. Kerridge, "Mobile Processes and Mobile Channels," in D. Tanier (Ed.), Encyclopedia of Mobile Computing and Commerce (EMCC) Volume 2, Idea Group Reference, 2007.

K. Chalmers, J. Kerridge, and I. Romdhani, "A Critique of JCSP Networking," in P. H. Welch, S. Stepney, F. A. C. Polack, F. R. M. Barnes, A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson (Eds.), Communicating Process Architectures 2008, pp. 271-291, IOS Press, Amsterdam, 2008.

J. Kerridge, J.-O. Haschke, and K. Chalmers, "Mobile Agents and Processes using Communicating Process Architectures," in P. H. Welch, S. Stepney, F. A. C. Polack, F. R. M. Barnes, A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson (Eds.), Communicating Process Architectures 2008, pp. 397-409, IOS Press, Amsterdam, 2008.