Kent Academic Repository

Schweigler, Mario (2006) ***A Unified Model for Inter- and Intra-processor Concurrency.*** **Doctor of Philosophy (PhD) thesis, University of Kent.**

Downloaded from

https://kar.kent.ac.uk/14444/ The University of Kent's Academic Repository KAR

The version of record is available from

This document version

UNSPECIFIED

DOI for this version

Licence for this version

UNSPECIFIED

Additional information

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal* , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

A UNIFIED MODEL FOR INTER- AND INTRA-PROCESSOR CONCURRENCY

a thesis submitted to

The University of Kent

in the subject of computer science

for the degree

of doctor of philosophy.

By

Mario Schweigler

August 2006

Abstract

Although concurrency is generally perceived to be a 'hard' subject, it can in fact be very simple — provided that the underlying model is simple. The occam-π parallel processing language provides such a simple yet powerful concurrency model that is based on CSP and the π-calculus. This thesis presents pony, the occam-π Network Environment. occam-π and pony provide a new, unified, concurrency model that bridges inter- and intra-processor concurrency. This enables the development of distributed applications in a transparent, dynamic and highly scalable way. The author specified the layout of the pony system as presented in this thesis, and carried out about 90% of the implementation.

This thesis is structured into three main parts, as well as an introduction and an appendix. In the introduction, the need for a unified concurrency model is examined in detail. Thereupon, the pony environment is presented as a solution that provides such a unified model. The first part of this thesis is concerned with the usage of the pony environment for the development of distributed applications. It presents the interface between pony and the user-level code, as well as pony's configuration and a sample application. The second part presents the design and implementation of the pony environment. It explains the internal structure of pony, the implementation of pony's components and public processes, and the integration of pony in the KRoC compiler. The third part evaluates pony's performance and contains the final conclusions. It presents a number of performance tests and concludes with a discussion of the work presented in this thesis, along with an outline of possible future research.

Contents

Abstract ii

List of Figures x

List of Tables xii

List of Algorithms xiii

Acknowledgements xv

Dedication xvii

1 Introduction 1

1.1 Problem and Motivation 1

1.2 The Need for a Unified Concurrency Model 3

1.3 Aspects of Transparency 4

1.3.1 Semantic Transparency 4

1.3.2 Pragmatic Transparency 6

1.4 History 7

1.5 New occam-π Features Relevant for pony 8

1.5.1 Mobile Data 8

1.5.2 Extended Rendezvous 9

1.5.3 Mobile Channel-types 10

1.5.4 Forking 13

1.5.5 Shared Plain Channels 14
1.5.6 Variables of Any Channel-type and Type-descriptors 15
1.5.7 The occam-π C interface 15
1.6 Related Developments 16
1.6.1 Static Approaches 16
1.6.2 Other CSP-based Platforms 17
1.6.3 Channel Mobility in Icarus 18
1.6.4 Other Languages Based on the π-calculus 18
1.7 Other Approaches for Distributed Application Development 19
1.7.1 The Grid 19
1.7.2 CORBA 19
1.7.3 DSM/ Tuple Spaces 20
1.7.4 PVM/MPI 20
1.7.5 Java Isolates 21
1.7.6 Singularity 21
1.8 Thesis Structure 22

I Using pony 24

2 Getting Started 25

2.1 Architecture and Terminology 25
2.1.1 Applications and Nodes 25
2.1.2 Network-channel-types 25
2.1.3 The Application Name Server 26
2.1.4 Network-types 27
2.1.5 Variants of Channel-types and Their Graphical Representation 27
2.2 Running pony on a Node 28
2.2.1 pony-enabled occam-π Programs 28
2.2.2 The pony Library 29

2.2.3 Public pony Processes and Handles 30
2.3 The Startup Mechanism 31
2.3.1 Different Versions of the Startup Process 31
2.3.2 Parameters of the Startup Processes 32
2.3.3 Design Rules 35
2.4 Starting the ANS 36

3 Operation of pony Nodes 37
3.1 Allocating NCT-ends 37
3.1.1 Explicit Allocation 37
3.1.2 Usage of NCTs and Implicit Allocation 40
3.2 Shutting Down Nodes 41
3.3 Configuration 43
3.3.1 The Node-file 44
3.3.2 The ANS-file 45
3.3.3 The ANS-configuration-file 46

4 Error-handling and Message-handling 48
4.1 Error-handling 48
4.1.1 Non-transparent Error-handling 48
4.1.2 pony's Error-handling Mechanism 49
4.1.3 Getting Information About NCTs and Remote Nodes 51
4.1.4 Getting and Deleting Error-points 54
4.1.5 Getting Errors That Happened After a Given Error-point . . . 55
4.1.6 Shutting Down the Error-handler 56
4.2 Message-handling 56

5 A Sample Application 59

II Design and Implementation of pony 66

6 Structure of pony 67

6.1 NCTs and CTBs . . . 67
6.2 Internal Components of pony . . . 68
6.2.1 The Individual Components . . . 70
6.2.2 Modular Design of pony . . . 73
6.3 CTBs in pony-enabled Programs . . . 74
6.3.1 The Pointer to the Type-descriptor . . . 75
6.3.2 The Claim/Release Mechanism . . . 77
6.3.3 Shutting Down pony-enabled CTBs . . . 81
6.4 The Main pony Kernel . . . 84
6.4.1 Layout and Startup . . . 84
6.4.2 Explicit Allocation of NCT-ends . . . 84

7 Protocol-conversion 90

7.1 The Protocol-converters . . . 90
7.2 Levels of Communication . . . 91
7.3 Implementation of the Protocol-converters . . . 93
7.4 Cancelling Started ULCs . . . 94
7.5 The Protocol-decoder . . . 95
7.6 The Protocol-encoder . . . 99
7.7 Decoding and Encoding the Various occam-π Protocols . . . 102
7.7.1 Data-item NLCs . . . 104
7.7.2 Channel-type-end NLCs . . . 112
7.8 Decode-handler and Encode-handler . . . 120
7.8.1 Differences Compared With the Protocol-converters . . . 120
7.8.2 CLC-Packets . . . 122

8 Handlers and Managers for CTBs and NCTs **125**
8.1 The CTB-handler 125
8.1.1 The Instant-handler 126
8.1.2 Client-listener and Server-listener 127
8.1.3 Sessions for NCTs 127
8.1.4 Starting a Session 129
8.1.5 Handling Sessions 133
8.1.6 Sending an NCT-end Over Itself 138
8.2 The CTB-manager 141
8.3 The NCT-handler 142
8.4 The NCT-manager 143

9 The Network Drivers **145**
9.1 The TCP/IP Link-handler 145
9.2 The TCP/IP Link-manager 147
9.3 Optimising TCP/IP Network Performance 150

10 Other Implementation Issues **153**
10.1 Implementation of Error-handling 153
10.2 Implementation of Message-handling 156
10.3 The Application Name Server 158
10.4 The Startup Mechanism 160
10.5 The Shutdown Mechanism 162

III Performance of pony, Evaluation and Conclusions **168**

11 Performance Evaluation **169**
11.1 Basic Considerations 169
11.2 Communication Time 170
11.3 Throughput 172

11.4 Network Overhead 174
11.5 CPU Overhead 175
11.6 Application Scalability 179
11.7 Distributed Robust Annealing Case Study 181

12 Conclusions and Future Work **188**
12.1 Evaluation of What Has Been Achieved 188
12.2 Adding Support for New/Future occam-π Features 191
12.2.1 Supporting Mobile Processes 191
12.2.2 Supporting Mobile Barriers 192
12.2.3 Supporting RMoX 192
12.2.4 Supporting Buffered Channels 193
12.2.5 Supporting Behaviour Patterns 193
12.2.6 Supporting the Proposed 'GATE'/'HOLE' Mechanism 193
12.2.7 Supporting New Fault Tolerance Mechanisms 197
12.3 Other Things 197
12.3.1 Adding Support for Networked Plain Channels 197
12.3.2 Adding Support for 'LOCAL' High Performance Channels 198
12.3.3 Supporting Different Architectures 200
12.3.4 Security and Reliability 201
12.3.5 Simplifying the Setup 201

Bibliography **203**

IV Appendices **214**

A Abbreviations and Acronyms **215**

B The Public pony Interface **216**
B.1 Public pony Processes 216

B.1.1 Startup Processes 217
B.1.2 Allocation Processes 237
B.1.3 The Shutdown Process 241
B.1.4 Error-handling Processes 242
B.1.5 The Message-outputters 251
B.2 Public pony Data-types and Constants 259
B.2.1 The Error Record 259
B.2.2 Message-types 260
B.2.3 Network-types 260
B.2.4 Node-types 260
B.2.5 Share-types 260
B.2.6 Results for Startup Processes 261
B.2.7 Results for Allocation Processes 262
B.2.8 Results for Error-handling Processes 263
B.2.9 Error-codes for TCP/IP 264

C Different ‘commstime’ Implementations 265

C.1 The Traditional ‘commstime’ Implementation 265
C.2 The Distributed ‘commstime’ Implementation 266
C.2.1 The Channel-type Declaration 266
C.2.2 The ‘prefix’ Node 266
C.2.3 The ‘delta’ Node 267
C.2.4 The ‘succ’ Node 268
C.2.5 The ‘consume’ Node 269

D Own Publications 270

List of Figures

2.1 Channel-type variants 28

5.1 Sample application: Possible dynamic layout 65

6.1 Layout of the pony environment 69

7.1 pony components related to protocol-conversion 90

8.1 pony components related to CTBs and NCTs 126

9.1 pony's network drivers 145

10.1 The error-handler 153
10.2 pony components related to message-handling 157

11.1 The 'commstime' benchmark 171
11.2 Throughput: 100 KB messages, two workers per slave 173
11.3 Throughput: Varying message size, one slave with 50 workers 174
11.4 Throughput: 50 KB messages, one slave, varying number of workers . 175
11.5 CPU overhead: Single byte array of varying size 176
11.6 CPU overhead: Sequential protocol, 1 B arrays 177
11.7 CPU overhead: Sequential protocol, several array sizes 178
11.8 The 'mandelbauer' application: Shared mode 179
11.9 The 'mandelbauer' application: Multiplexing mode 180
11.10 Scalability of a distributed application 181
11.11 Annealing: occam-π versions 184

11.12Annealing: Single machine versions 185
11.13Annealing: Distributed versions 186

List of Tables

6.1 Memory layout of CTBs in a pony-enabled KRoC build 75

7.1 Layout of the first byte of an NLC-descriptor 124

List of Algorithms

3.1 Typical structure of a pony node 42

4.1 An error-handling example 50

5.1 Sample application: Declarations 61
5.2 Sample application: The broker 62
5.3 Sample application: The worker 63
5.4 Sample application: The customer 64

6.1 Claiming/releasing ends of a pony-enabled CTB 80
6.2 Making a CTB networked when necessary 81
6.3 Reference-count check for a pony-enabled CTB 81
6.4 Shutdown of a networked CTB 83

7.1 Pseudo output guard mechanism used in the protocol-decoder 97
7.2 Counterpart to pseudo output guard in the decode-handler 98
7.3 Implementation of ‘cancel-encode’ operation in the decoder 100
7.4 Initiating ‘cancel-encode’ operation in the CTB-handler 101
7.5 Several examples of decoding counted arrays 109
7.6 Making a CTB networked — detailed description 113

8.1 Initiating the closing of a session 135
8.2 Reacting to a ‘close-session’ message 136
8.3 Initiating the suspension of a session 139

8.4 Reacting to a 'suspend-session' message 140

10.1 Deleting an error-point 156

Acknowledgements

The work on this PhD thesis was carried out on a full-time basis in the Computing Laboratory at the University of Kent, largely funded by a Brian Spratt Bursary from the Computing Laboratory.

I would particularly like to thank my supervisor Professor Peter Welch for his guidance and support over the course of my PhD. His occam lectures during my Masters degree inspired me for this great parallel processing language, and CSP-style concurrency in general. This was why I decided to develop a network environment for occam for my MSc dissertation, and why I continued the project for my PhD. Many thanks also go to the members of my Supervisory Panel, David Wood and Gerald Tripp, for their advice and help, and to the Computing Laboratory as a whole for funding my research and providing office space and equipment. Thanks also to EPSRC for funding the TUNA project, whose high performance cluster was used to run the performance tests presented in this thesis.

I would also like to thank the Concurrency Research Group at the Computing Laboratory for creating an excellent academic environment. Over the years, a lot of research was done in the Concurrency Group, much of which provided valuable stimuli for my work. In particular, many new features were incorporated in the occam language, which has now grown into occam-π. This involved many discussions in the research group that also positively influenced the pony project. I would particularly like to acknowledge Fred Barnes for his advice on the KRoC compiler and for implementing the compiler support and various auxiliary functions for pony,

as well as Adam Sampson for implementing pony's protocol-converters and various benchmarks/demos.

Acknowledgements also go to the WoTUG community, whose annual CPA conferences have been a great forum for exchanging ideas and discussing research results and future plans. Many WoTUG members provided valuable feedback for my research. Particularly, I would like to thank Dyke Stiles for porting his Distributed Robust Annealing package to occam-π/pony and running a series of tests, the results of which are presented in this thesis.

Finally, I would like to thank my family and friends for their support during my PhD, especially my wife Karin for enduring the long separation while I was away in England.

Dedication

To my wife Karin.

Chapter 1

Introduction

This thesis is concerned with a new, unified, concurrency model that bridges inter- and intra-processor concurrency. The need for such a unified model is examined in detail. Thereupon, pony[1], the occam-π Network Environment, is presented as a solution that provides such a unified model. This thesis presents the pony environment, both in usage and implementation, and evaluates the achievements to date. The author specified the layout of the pony system as presented in this thesis, and carried out about 90% of the implementation.

1.1 Problem and Motivation

Historically, computing has been sequential. This has several reasons, which may probably be best summed up in a single word: tradition. For decades, the von Neumann architecture [vN93] has moulded our view on what a computer is and how a computer works. The basic paradigm has always been the sequential execution of instructions, and, despite the fact that modern processors have evolved enormously from the first von Neumann based designs, this notion remains virtually unchanged.

[1]The name 'pony' is an anagram of the first letters of [o]ccam, [p]i and [n]etwork; plus a [y] to make it a word that is easy to remember.

The idea of step-by-step execution being the somehow 'natural' way of computing is still 'hard-wired' in the collective mind of the computer science community.

Concurrency has traditionally been seen as an 'advanced' subject. It is taught late (if at all) in computer science curricula, because it is seen as a non-trivial extension of the 'basic' sequential computing. In a way, this is surprising, since the real world around us is highly concurrent. It consists of entities that are communicating with each other; entities that have their own internal lives and that are exchanging information between each other.

Process calculi such as CSP [Hoa85], with their notion of processes and channels, are particularly suited to model the real world. Processes can be used to model real-world entities, since they are self-contained and have their own internal behaviour and clearly defined interface. Processes may be encapsulated into each other, just like real-world entities. Channels are the means of exchanging information between processes. They provide a clear interface between processes without 'hidden' routes. The π-calculus [Mil99] additionally gives us the notion of mobility. It allows us to communicate processes and channels over other channels, which enhances the flexibility of the modelled systems.

There is a programming language available that is based on those formal calculi, but is still easy to understand and to use. This language is occam-π, the new dynamic version of the classical occam[2] [Inm95]. Originally targeted at transputer [Inm88] platforms, it was specifically designed for the efficient execution of fine-grained, highly concurrent programs. Still, most people associate concurrency with the traditional approach of threads, locks and semaphores rather than with the much more intuitive one of a process algebra.

Networking is increasingly important in today's world. Originally a merely academic topic, it has gained significant importance since the 1990s, especially due to

[2] occam is a trademark of ST Microelectronics. The original occam language was based on CSP only; dynamic features from the π-calculus, particularly the notion of channel and process mobility, have been incorporated in occam-π recently.

the advent of the internet as an everyday 'commodity' on the consumer market. The development of large distributed applications is one of the modern challenges in computer science. Infrastructures such as the Grid [FKT02, FKT01, FKNT02] are specifically designed for the distribution of large computational tasks onto decentralised resources.

Distributed applications are typically *designed* to be distributed right from the start — the mechanisms used for distribution must be specifically addressed by the developer. The work within this thesis is targeted towards bringing concurrency and networking together in a *transparent* and dynamic yet efficient way, using the occam-π language as the basis for the development of distributed applications. This is possible because, as stated above, the world is concurrent by nature, which includes networks of computers. A programming language such as occam-π, which by design captures this 'natural' concurrency, is particularly suited as the basis for a unified concurrency model.

1.2 The Need for a Unified Concurrency Model

Concurrency is simple — provided that the underlying model is simple. occam-π offers just that, a concurrency model that is simple to use, yet based on the formal algebras of CSP and the π-calculus. One of the major advantages of occam-π is that it encourages component-based programming. Each occam-π process is such a component, which can communicate with other components. occam-π applications may be highly structured, since a group of processes running in parallel can be encapsulated into a 'higher level' occam-π process, and so on.

This component-based approach is the particular charm of occam-π programming. It allows the development of sub-components independently from each other, as long as the interface for communication between those sub-components is clearly defined. In occam-π, this interface is provided (primarily) by channels; this includes both the

'classical' occam channels and the new dynamic channel-types[3] [BW02]. Once all components of an occam-π application have been developed, they just need to be 'plugged together' via their interfaces.

We want to utilise the advantages of occam-π's concurrency model for the development of distributed applications. In order to do this successfully, it is necessary to extend occam-π in such a way that the distribution of components is transparent to the components' developers. As long as the interface between components (i.e. processes) is clearly defined, the programmer should not need to distinguish whether the process on the 'other side' of the interface is located on the same computer or on the other end of the globe.

1.3 Aspects of Transparency

pony, the occam-π Network Environment, extends occam-π in such a transparent way. There are two aspects of transparency that are important: *semantic* transparency and *pragmatic* transparency.

1.3.1 Semantic Transparency

occam was originally developed to be executed on transputers. The transputer was a microprocessor with a built-in micro-coded scheduler, allowing the parallel execution of occam processes. occam channels were either emulated within a single transputer if their ends were held by processes on the same transputer ('soft channels'), or implemented using the transputer's links. Each transputer had four links by which it could be connected to other transputers ('hard channels'). The late T9000 transputer [Inm93], which was the last transputer being developed and which went out of production shortly after having been introduced by Inmos, additionally offered a *Virtual*

[3] Channel-types are bundles of channels. The ends of channel-types are mobile and may be communicated between processes.

Channel Processor (VCP) [MTW93] which allowed many logical occam channels to be multiplexed over the same physical link.

This approach allowed the simple construction of networks of transputers offering large computing power, despite the (comparatively) low processing capabilities of a single transputer. The great advantage of this approach was that the programmer of an occam process did not have to care whether a specific channel was a soft or a hard channel. This distinction was transparent from the programmer's point of view — the semantics of channel communication was identical for all occam channels.

After the decline of the transputer, the 'occam For All' [Poo96] project successfully saved the occam language from early retirement. Although occam was originally targeted at transputers, the aim was to bring the benefits of its powerful concurrency model to a wide range of other platforms. This was achieved by developing KRoC, the Kent Retargetable occam Compiler [WW96]. KRoC essentially consists of two parts. The front-end compiles occam code into Extended Transputer Code (ETC) [Poo98], which is an extension of the original transputer bytecode.

The back-end of KRoC is a *translator* that translates ETC into the native code of the target platform, and links it with a runtime kernel that provides a transputer-style scheduler. This allows the execution of an entire occam program in a single OS-level process, without the need to involve OS-level thread scheduling. Therefore, occam programs compiled with KRoC are highly efficient, offering context-switch times in the range of nanoseconds. In this way, a KRoC program, together with the linked-in runtime kernel, provides the same functionality as an occam program running on a single transputer; the runtime kernel emulating the transputer's support for process scheduling and soft channels. What had been lost, however, was the support for hard channels, since without transputers there were no transputer links anymore.

The pony environment re-creates the notion of semantic transparency from the old transputer days. pony enables the easy distribution of an occam-π application across several processors — or back to a single processor — without the need to change the application's components.

With the constant development of KRoC, occam has been developed into occam-π, which offers many new, dynamic, features [BW01, BW02, Bar03]. pony takes into account and exploits this development. In the classical occam of the transputer days, channels were the basic communication primitive, and semantic transparency existed between soft and hard channels. pony's basic communication primitive are occam-π's new channel-types, and there is semantic transparency between non-networked channel-types and *network-channel-types (NCTs)*. This transparency includes the new dynamic features of occam-π.

All occam-π PROTOCOLs can be communicated over NCTs. Mobile semantics (cf. Section 1.5.1) are preserved as well, both when mobile data [BW01] is communicated over NCTs, and when ends of (networked or non-networked) channel-types are communicated over other channel-types. The semantics is always the same, and the developer of an occam-π process does not have to care whether a given channel-type is networked or not. Some of pony's general routing mechanisms are similar to the Virtual Channel Processor of the T9000 transputer; however, routing in pony is dynamic, rather than static as on the transputer.

1.3.2 Pragmatic Transparency

When achieving semantic transparency, we do not want to pay for it with bad performance. For instance, a system that uses sockets for every single communication, including local communication, would still be semantically transparent — since the developer would not have to distinguish between networked and non-networked communication — but it would be hugely inefficient. Here the other important aspect becomes relevant, namely pragmatic transparency. This essentially means that the infrastructure that is needed for network communication is set up *automatically* by the pony environment when necessary. Due to pony's dynamic routing, it is used if and only if needed.

Local communication over channel-types is implemented in the traditional occam-π way, involving access to the channel-word only. In this way, the pony environment preserves one of the key advantages of occam-π and KRoC, namely high performance and lightweight, fine-grained concurrency. Only when the two ends of an NCT are not located on the same node of a distributed application, communication between them goes through the infrastructure provided by pony. But also for this case, high performance was one of the key aspects during pony's development; the network communication mechanisms in pony are specifically designed to reduce network latency.

This pragmatic transparency approach, together with a simple setup and configuration mechanism, makes the pony environment very dynamic and highly scalable. The topology of a distributed application written in occam-π and pony is constructed at runtime and can be altered by adding or removing nodes when needed or when they become available.

1.4 History

The development of pony and its predecessors has gone through a number of stages. Originally, it started as an undergraduate student project in 2001 [Goo01]. In autumn 2001, the first major version was released as part of an MSc dissertation under the name 'Distributed occam Protocol' [Sch01]. This version was implemented fully in occam and offered a certain degree of transparency. Due to the limitations of the occam language at that time, it was far from being fully semantically transparent, however.

Since then, the pony project has continued as part of this PhD thesis work[4] [SBW03, Sch04]. During this time, the occam language was extended significantly[5], adding many dynamic features, some of which are discussed in Section 1.5. This

[4] partly under the provisional name 'KRoC.net'
[5] and renamed to 'occam-π'

affected the pony project two-fold. Firstly, the new dynamic features in occam-π enabled the pony environment to be implemented in a semantically and pragmatically transparent way; being implemented almost entirely in occam-π, with a small part implemented in C, as well as some compiler-level support built-in directly in KRoC. Secondly, features such as the new dynamic channel-types were themselves incorporated in the pony environment. As the development of occam-π progresses in the future, the pony environment will also have to be extended to accommodate support for new developments; foremost for mobile processes [BW04] and mobile barriers [WB05].

Another ongoing project is RMoX, the occam operating system [BJV03]. Integrating the pony environment into RMoX will be another important aspect of the future development of pony. Since RMoX is implemented in occam, an RMoX-integrated pony environment would be able to utilise RMoX's native network drivers directly rather than going through an underlying operating system. This could further enhance network performance compared to versions of occam-π and pony that are running on top of an 'ordinary' OS.

1.5 New occam-π Features Relevant for pony

This section contains a (very condensed) overview of some recent extensions to occam-π that are relevant to pony in one way or another. For a comprehensive reference, the reader is referred to the KRoC homepage [WMBW06].

1.5.1 Mobile Data

Mobile data-items differ from traditional static occam data-items in their semantics of assignment and communication. While static data is *copied* when it is assigned to another variable or communicated via a channel, mobile data is *moved*. This means

that after the assignment or communication, the variable that held the data-item before is now *undefined*.

If we need an actual copy of a mobile data-item, we can *clone* it, and then continue using the clone. The original variable will remain defined and unchanged, and the clone will contain the same data as the original variable at the time of the cloning. If we change the value of the original variable or of the clone later, the other variable will not be affected by this. That is, neither moving nor cloning introduce aliasing.

When mobile data-items are communicated across the network via the pony infrastructure, they are physically copied but retain their mobile semantics, i.e. the source variable is undefined after the communication. This behaviour guarantees semantic transparency.

A special subgroup of mobiles are dynamic mobile arrays. These are mobile as defined above, and differ from ordinary occam-π arrays insofar as their size is not known at compile-time and they are allocated dynamically at runtime.

The key advantage of mobile data is that large data-items that take up a lot of memory do not need to be copied anymore. This saves resources and time. But also apart from this, there are applications where mobile data offers a 'natural' way of modelling certain scenarios. For instance, if a data structure models the access to a particular resource which only one process is allowed to use at a time, it makes sense to implement this data structure as a mobile — which gives us this property by definition.

1.5.2 Extended Rendezvous

The extended rendezvous allows to 'intercept' an occam-π channel communication. Contrary to normal input operations, the process writing to the channel is not released from the write operation until explicitly specified by the inputting process. This allows for instance to read from a channel, perform some operation on the data that

was input, pass the data on to another process, wait for the other process to take the data, and only then to release the original writing process.

The key point is that for the original writer, the handshake semantics between itself and the final reader is preserved even though a process in the middle has intercepted the data. This allows to add further processes to an application without changing the semantics of the existing processes.

The syntax for the extended rendezvous is as follows[6]:

```
chan ?? x    -- Extended input from channel ‘chan’ into ‘x’
  ...  Stuff to be done while writing process remains suspended
  ...  Stuff to be done after writing process has been released
```

The first indented process is called the *during-process*; the second one is called the *after-process*. The latter may be omitted, in which case it is treated like ‘SKIP’. The during-process may not use the ‘extended’ channel since this would result in deadlock. The after-process is intended for situations where the extended rendezvous is used in an ‘ALT’ or ‘CASE’ input. For stand-alone extended rendezvous inputs, the after-process is meaningless, since it may be done just as well after the extended rendezvous has finished.

pony uses the extended rendezvous mechanism to ‘stretch’ communications over the network while preserving the original handshake semantics between sender and receiver, giving us semantic transparency. This means that their codings do not need to be changed. Application processes do not need to detect and take account of whether the external channels they are using are networked.

1.5.3 Mobile Channel-types

As already discussed in the above sections, channel-types are mobile bundles of channels. They have two ends, called ‘client-end’ and ‘server-end’ for convenience. These

[6]Lines starting with ‘...’ denote parts of the code that have been *folded*. This notation, which is used by origami and other folding editors, will be used throughout this thesis.

names are merely distinctions, however; channel-types may be used for any topology necessary, not just for client/server architectures.

A channel-type may contain many channels of different protocols and of different directions. They are declared from the server-end point of view:

```
CHAN TYPE FOO
  MOBILE RECORD
    CHAN INT req?:
    CHAN INT reply!:
:
```

In the above example, the server-end of the channel-type 'FOO' holds the reading-end of the request channel and the writing-end of the reply channel. The client-end of 'FOO' holds exactly the opposite ends of the channels.

Channel-types are allocated in pairs of one server-end and one client-end variable. Either or both of these variables may be shared. The following example allocates a 'FOO' channel-type with a shared client-end and an unshared server-end:

```
SHARED FOO! foo.cli:    -- Shared client-end
FOO? foo.svr:           -- Unshared server-end
SEQ
  foo.cli, foo.svr := MOBILE FOO
  ...  Use 'foo.cli' and 'foo.svr'
```

Shared channel-type-ends must be claimed before they can be used for communication over their channels. When a shared end is assigned to another variable, or sent over a channel, it gets auto-cloned, and then the clone is assigned resp. communicated. Since internally, channel-type-end variables are pointers to a memory structure that holds the channel-type, the clone is just another pointer to the same memory structure (i.e. to the same channel-type). In this way, after the assignment or communication, both the original variable and the target variable will logically contain the shared end. This

introduces aliasing, but it is *explicitly* declared and controlled by language structures and rules. In particular, as mentioned above, a shared end must be claimed before it can be used. When an unshared end is assigned or communicated, the original variable will be undefined afterwards — as usual for mobile variables.

The mobility of channel-type-ends has the same advantages as the mobility of data discussed in Section 1.5.1. In particular, it allows to model the access to resources that are 'mobile' in the real world too.

The channels within the channel-type would be used in the following way:

```
-- Process using shared client-end
INT n:
SEQ
  CLAIM foo.cli
    SEQ
      foo.cli[req] ! 47
      foo.cli[reply] ? n
  ...  Use 'n'

-- Process using unshared server-end
INT n:
SEQ
  foo.svr[req] ? n
  foo.svr[reply] ! n * 5
```

Channel-types provide the basic communication paradigm for the pony environment. As discussed above, the network-channel-types that are implemented by the pony infrastructure are semantically and pragmatically transparent to occam-π's non-networked channel-types.

1.5.4 Forking

In classical occam, processes could only be started by calling them directly. Therefore, the topology of an occam network of processes was determined at compile-time and could not be altered. occam-π offers the possibility to fork off processes dynamically at runtime when needed. This approach is similar to the well-known mechanisms of forking off OS-level processes, for instance in Unix-derived systems, although syntax and handling are much simpler in occam-π:

```
FORK proc (...)
```

This example forks off the process '`proc`' and then continues in parallel with it. It is also possible to include forking calls in a special forking block:

```
FORKING
  SEQ
    ...  Do stuff
    FORK proc (...)
    ...  Do other stuff
```

The '`FORKING`' block will not terminate until all processes that have been forked off within the forking block have terminated themselves — similarly to a normal '`PAR`' construct.

Note that forking can always be replaced by '`PAR`' constructs. In case of a '`FORK`' in a loop, the replacement '`PAR`' construct would need to be recursive. For instance, the following example:

```
WHILE TRUE
  SEQ
    ...  Acquire 'chan.type.svr'
    FORK worker (..., chan.type.svr, ...)
```

could be replaced by

```
REC PROC rec.proc (...)
  SEQ
    ...  Acquire ‘chan.type.svr’
    PAR
      worker (..., chan.type.svr, ...)
      rec.proc (...)
:
rec.proc (...)
```

However, using ‘PAR’ constructs instead of forks may make the code significantly more complex and add extra overheads. Especially in an infinite loop, the resources taken by the recursively entered ‘PAR’ constructs are never freed, whereas a forked process simply terminates.

The support for forking in occam-π allowed pony to be implemented in a pragmatically transparent way. Sub-components of pony are only started when they are needed – which may be never, depending on the needs of the particular application.

1.5.5 Shared Plain Channels

There is now support for declaring shared ‘plain’ occam-π channels as in the following example:

```
PROC output.something (SHARED CHAN BYTE out!)
  CLAIM out!
    ...  Use ‘out!’ for output
:
```

```
PROC example (CHAN BYTE key?, SHARED CHAN BYTE scr!, CHAN BYTE err!)
  SEQ
    FORK output.something (scr!)
    FORK output.something (scr!)
:
```

Shared plain channels are implemented by the compiler as anonymous channel-types with only one channel inside. pony supports shared plain channels for instance in its message-handling mechanism.

1.5.6 Variables of Any Channel-type and Type-descriptors

occam-π offers a new special type called 'MOBILE.CHAN'. Variables of that type can hold any channel-type. pony uses this feature for its allocation processes. These processes take variables of any given channel-type as parameters and allocate them as ends of NCTs.

In order to do this, another recent feature of occam-π is exploited, namely type-descriptors for channel-types. These are generated automatically for every channel-type defined in an occam-π program, and describe the structure and protocols within that channel-type. For 'MOBILE.CHAN' parameters, KRoC adds another, hidden, parameter that points to the type-descriptor of the argument. Using this approach, pony's allocation processes can allocate any given channel-type-end variable, even though the parameters of the allocation processes are of type 'MOBILE.CHAN' (with the actual type being unknown at compile-time).

1.5.7 The occam-π C interface

The C interface (CIF) [Bar05] allows processes written in C to run in parallel with occam-π processes and communicate with them. Calling external C processes from occam is a fairly old feature. However, the calling occam process always had to wait for the C process to terminate before it could continue itself. What is new about the

C interface is that now occam-π and C processes can run in parallel (which includes support for forking off C processes) and communicate with each other.

pony uses the occam-π C interface for its protocol-converters (cf. Section 7.1). These are implemented in C and interact with the rest of the system via the mechanisms provided by the C interface.

1.6 Related Developments

1.6.1 Static Approaches

In the old transputer days, there was semantic transparency between hard and soft channels. Since occam itself was a purely static language, also the layout of process networks had to be determined statically at configuration time, using the 'PLACED PAR' mechanism.

There have been projects to target multi-processor platforms other than transputers. An example is Vella's work on porting occam to Networks of Workstations (NoWs) [Vel98, VW99] which uses an interface to communicate with the network, called 'ocnet'. This interface is a layer between the occam kernel on the one hand, and the operating system and the networking hardware on the other. Like the original networks of transputers, this system was static and the layout of the network had to be configured in advance.

Another example is MESH [BDv99], which is completely hardware-dependent. MESH is a messaging and scheduling system that enables access to network hardware (e.g. ethernet cards) and provides user-level scheduling in Linux. The original version of CCSP [Moo99], on which the current occam-π kernel is based, could access the messaging capabilities of MESH. But again, this was hardware-dependent and static.

Lately, Barnes' 'Application Link Layer' [Bar05] was developed. This is essentially a static core version of the protocol-converters used in pony, which are described in Chapter 7 of this thesis.

1.6.2 Other CSP-based Platforms

There are other platforms based on CSP, in particular JCSP (*CSP for Java*) [Wel99] and C++CSP (*CSP for C++*) [BW03]. These are libraries on top of their host languages that offer occam-style concurrency mechanisms. This includes:

- processes, channels
- networks of processes
- read, write, '`ALT`' constructs
- channels with shared ends (similar to the shared channel-type-ends in occam-π, cf. Section 1.5.3)

It is possible to move channel-ends and processes over channels (since they are objects), similar to the new mobility features in occam-π. Everything is 'naturally' mobile, since it is object references that are passed around. However, this incurs the risk of aliasing, whereas mobility in occam-π is explicitly aliasing-free (cf. Section 1.5.1).

The *JCSP Network Edition* (JCSP.net) [WAF02, WV02] is the network extension of JCSP. Some of the concepts in pony and JCSP.net are similar. JCSP.net allows the stretching of channels over a networking fabric (e.g. TCP/IP) and the dynamic creation of distributed networks of processes. The basic paradigm are network-channels rather than NCTs as in pony. JCSP.net uses a Channel Name Server which allows processes to find network-channels by a pre-defined name. pony uses a similar concept with its Application Name Server; see below.

JCSP.net is not fully transparent, however. When a Java object is communicated over a JCSP channel locally, its object reference is communicated, which introduces aliasing. Across the network, objects are serialised and copied when they are communicated over JCSP.net channels. This gives us different semantics locally and networked. Also the semantics of channel-ends is not fully transparent in JCSP.net.

The reading-end of a JCSP.net channel may be moved across the same node, but not in the same way across the network. To move a reading-end over the network, the user-level code must 'lease' the channel from the Channel Name Server, move an identifier of the channel to the target node, and re-register the channel with the Channel Name Server. That is, the user-level code must be aware that this is a 'network operation'.

A network environment for C++CSP was also implemented [Bro04]. This is still very basic, however, and lacking most of the dynamic features of JCSP.net and pony. For instance, there is no Channel Name Server such as in JCSP.net, which means that currently network-channels in C++CSP.net are connected by using physical network locations.

1.6.3 Channel Mobility in Icarus

The mobility of ends of network-channel-types was inspired by the mobile channels in Muller and May's Icarus language [MM98]. However, implementing mobility for pony's NCT-ends is substantially more complex because it needs to take into account the special properties of channel-types compared to plain channels. This includes the fact that channel-types are bundles of channels, as well as that channel-type-ends may be shared and that shared ends must be claimed before they can be used. All these features had to be incorporated into NCTs as well, in order to achieve semantic transparency.

1.6.4 Other Languages Based on the π-calculus

There are other languages that are based on the π-calculus, for instance Pict [PT00] and PiLib [CO03]. The latter is a library written in Scala [OAC+05], a Java-style language merging object-oriented and functional programming. Both Pict and PiLib are closely related to the π-calculus, especially as far as the syntax is concerned.

Unlike occam-π, whose concurrency features are based on the classical imperative programming paradigm, Pict and PiLib/Scala are, in principle, functional programming languages. This makes them less intuitive than occam-π, and entails a much steeper learning curve, especially for programmers who are not used to functional programming.

1.7 Other Approaches for Distributed Application Development

There is a vast number of other approaches for developing distributed applications, with a variety of different basic distribution primitives. This section gives an overview of some of the most common architectures and how they relate to the pony/occam-π approach.

1.7.1 The Grid

The most 'traditional' paradigm are remote process or method calls. The basic distribution primitive of Grid-based systems, such as the Globus Toolkit [FK97, IBM03] or the Minimum intrusion Grid [Vin05], are 'jobs'.

There are resource and user nodes — a user node submits a job to be executed by some resource node. This approach is specifically designed to be distributed, and there is a clear distinction between the farming-out of jobs to remote resources and the code that is executed locally.

1.7.2 CORBA

The Object Request Broker mechanism of CORBA [Obj93] provides support for the distribution of object-oriented applications. CORBA allows access to the methods of a remote object by providing a 'proxy' object that receives method calls and forwards them to the remote object. Essentially, CORBA also belongs to the RPC based platforms, just in an OO context.

1.7.3 DSM/ Tuple Spaces

Other architectures are built on distributed shared memory or tuple spaces, for instance Linda [CG89] and Java PastSet [PV02]. Tuple-spaced systems are semantically transparent because both local and networked communication are done via the tuple paradigm.

Since tuple spaces are built on a distributed shared memory system, a growing distributed application can cause pragmatic problems. This is because all available information is stored in the distributed shared memory as a central resource. Keeping a consistent view of the shared memory is not trivial and gets more complex as the system grows.

1.7.4 PVM/MPI

Another common approach for developing distributed applications are PVM (Parallel Virtual Machine) [Sun90] and its de-facto successor MPI (Message Passing Interface) [MPI97], implemented for instance in the popular LAM/MPI library [Ind04]. As in occam-π and pony, message passing is the basic paradigm here.

Contrary to pony, however, a distributed application using an MPI implementation would not be fully transparent in the way described earlier. Either the semantic transparency is missing — because local communication[7] is typically not done through the message passing mechanisms provided by the MPI library — or the pragmatic transparency is missing. Of course, a program which uses MPI's functions for every single communication is imaginable, but the overheads would be significant so that such a program would not meet our criteria of pragmatic transparency.

[7] provided that the host language is also concurrent and based on message passing, otherwise it would not be semantically transparent anyway

1.7.5 Java Isolates

A recent Java related development are Java Isolates [Sun05]. The Java Isolation API allows to run several Java programs, each with its own main method, in the same JVM. Each of these programs is run in an 'isolate' that provides the same level of protection as if the programs were run in separate JVMs. This approach increases performance and scalability without losing security. Compared to occam-π's concurrency model, concurrency in Java Isolates is coarse-grained only.

Part of the specification is the possibility to communicate between isolates via 'links'. If we draw a connection to a process algebra, isolates could be seen as processes whereas links have a certain resemblance with channels. Comparing Java Isolates links to occam-π channels, however, shows that link mobility is much more restricted than in the π-calculus. In particular, a link is permanently bound to the two isolates (at each end) with which it is initiated. Also, spawning links between different JVMs in a transparent way (which would be inevitable for a transparent distribution model such as pony's) is not part of the Java Isolates specification.

Ultimately, the basic idea behind Java Isolates is not distribution but security and efficient memory protection within a single JVM (compared with the memory protection in separate JVMs) — something already present for occam-π processes.

1.7.6 Singularity

Singularity [Mic05] is a recent research project in Microsoft Research that is aimed at creating an operating system whose primary design goal is dependability. The Singularity operating system is built on so-called Software-Isolated Processes (SIPs) which show surprisingly many parallels to the occam/CSP process model, as well as to the way RMoX [BJV03], the occam operating system, is designed. SIPs can be created with low overheads and allow fine-grained concurrency.

There is a strong separation between different SIPs that disallows simultaneous access to the same resource by two processes. Communication between SIPs is through

strongly typed channels, although in Singularity these are bidirectional and asynchronous as opposed to the channels in occam-π and RMoX. Singularity channels are typed by a contract that allows the specification of valid sequences of messages along a (bidirectional) channel.

Singularity also uses the same notion of mobile data as occam-π. When a resource is sent from one process to another, the sending process loses it. This ensures that the ownership of mobile data always belongs to one process only; this semantics is the same as in occam-π.

Although supporting distributed applications is no great concern of the project at the moment, Singularity offers the potential for a unified concurrency model. Since everything in Singularity, from low-level drivers up to user-level software, is built on SIPs, it would seem logical to use the same model for distributed applications as well. How the Singularity project will develop in the years to come remains to be seen.

1.8 Thesis Structure

This thesis consists of three main parts. The first part describes the usage of pony for the development of distributed applications. Chapter 2 introduces the terminology used in this thesis and presents the architecture and the startup of the pony environment. The operation of pony nodes is covered in Chapter 3. Chapter 4 is concerned with pony's error-handling and message-handling mechanisms. Finally, Chapter 5 outlines a sample pony application.

The second part of this thesis presents the design and implementation of the pony environment. Chapter 6 gives an overview of the internal structure of the pony system. pony's protocol-conversion mechanism is examined in Chapter 7. Chapter 8 describes how networked channel-types are implemented in pony. In Chapter 9, the network-specific parts of pony are explained. The second part concludes with Chapter 10, where the implementation of pony's error-handling and message-handling

mechanisms, the Application Name Server and pony's startup and shutdown mechanisms are discussed.

The last part of this thesis evaluates pony's performance and contains the final conclusions. Chapter 11 presents a number of tests that were carried out to examine the performance of the pony environment. Chapter 12 concludes with a discussion of the work presented in this thesis, along with an outline of possible future research.

The appendices provide supplementary information that is not part of the main thesis. Appendix A contains a list of abbreviations and acronyms used in this thesis. Appendix B gives a comprehensive reference of the public interface of the pony environment, consisting of the public processes, data-types and constants. Appendix C compares the traditional implementation of the '`commstime`' benchmark with a distributed one using pony, in order to give a practical example of how to use pony to make an existing application distributed. Finally, Appendix D lists the author's publications that are related to pony and its development.

Part I

Using pony

This part of the thesis is concerned with the usage of the pony environment for the development of distributed applications. Chapter 2 introduces the terminology used in this thesis and presents the architecture and the startup of the pony environment. The operation of pony nodes is covered in Chapter 3. Chapter 4 is concerned with pony's error-handling and message-handling mechanisms. Finally, Chapter 5 outlines a sample pony application.

Chapter 2

Getting Started

2.1 Architecture and Terminology

2.1.1 Applications and Nodes

A group of occam-π programs which are interconnected by the pony infrastructure will be referred to as a pony *application* for the remainder of this thesis. Each application consists of several *nodes* — one *master* node and several *slave* nodes.

The term 'node' refers to an occam-π program which is using the pony environment. That is, there may be several nodes on the same physical computer; these nodes may belong to the same application or to different applications. In the non-networked world, node and application would be congruent. In the networked world, an application is made up of several nodes; the master is the logical equivalent of the main process of a non-networked occam-π program (in the sense that all the 'wiring' of the application originates from there).

2.1.2 Network-channel-types

A *network-channel-type (NCT)* is a channel-type that may connect several nodes, i.e. whose ends may reside on more than one node. An individual NCT-end always resides on a single node, and like any channel-type, an NCT may have many end variables if

one or both of its ends are shared. NCTs are the basic communication primitive for pony applications. Nodes communicate with each other over NCTs, using the same semantics as for conventional channel-types. This includes the protocol semantics of the items that are communicated over the NCT's channels as well as the semantics of NCT-ends.

Like any other channel-type-end, NCT-ends may be communicated over channels, which includes channels of other NCTs. Also, if an NCT-end is shared, it must be claimed before it can be used, and it is ensured by the pony infrastructure interconnecting the application that every shared NCT-end can only be claimed once at any given time across the entire application. Practically, the master node queues claim requests for each end of each NCT and ensures that each NCT-end is only claimed once at any given time.

NCTs are either allocated *explicitly*, under a name that is unique within the application, or *implicitly* by moving ends of locally allocated channel-types to a remote node.

2.1.3 The Application Name Server

An *Application Name Server (ANS)* is an external server that administrates applications. Each application has a name that is unique within the ANS by which it is administrated. Nodes of the application find each other by contacting the ANS. This concept is similar to the 'Channel Name Server' in JCSP.net, only on the level of applications rather than channels (respectively NCTs for pony). This allows a better abstraction, as well as a simpler name-spacing.

With pony, NCTs are still allocated by using names, but this is managed by the master node of the application to which the NCT belongs, rather than by the ANS. This two-level approach makes it simpler to have a single ANS for many applications. In JCSP.net, it is also possible to administrate network-channels of many separate

JCSP.net applications within the same Channel Name Server; however, avoiding naming conflicts is the programmer's task there.

The ANS stores the location of the master node of an application. When a slave node wants to join the application, it would contact the ANS and request the master's location. Then the slave would contact the master node itself. Each slave node of an application has a network *link* to the master node. Links between slave nodes are only established when this becomes necessary, namely when an NCT is stretched between those two slave nodes for the first time.

2.1.4 Network-types

The pony environment has been designed to support potentially many network infrastructures. These are referred to as *network-types* in the following. Currently, the only supported network-type is TCP/IP. However, adding support for other network-types in the future would be easy because the internal structure of pony is modular.

In order to add support for a new network-type, modified versions of the network drivers and the ANS would have to be added to pony. These only comprise a relatively small part of the pony infrastructure. The non-network-type-specific components of pony would interact with the new network drivers using the existing interface.

2.1.5 Variants of Channel-types and Their Graphical Representation

For the remainder of this thesis, we will refer to the following variants of channel-types: *one2one* channel-types have an unshared client-end and an unshared server-end. *any2one* channel-types have a shared client-end and an unshared server-end. *one2any* channel-types have an unshared client-end and a shared server-end. Lastly, *any2any* channel-types have a shared client-end and a shared server-end. This property will henceforth be called the *x2x-type* of the channel-type. The x2x-type is a property of concrete instances of a channel-type, not of its type declaration (see Section 1.5.3 for details).

Figure 2.1 shows how channel-types are depicted in this thesis. The client-end of a channel-type is represented by a straight edge, the server-end by a pointed edge. Shared ends are darkened. So, for instance a one2one channel-type has no darkened edges, whereas an any2one channel-type has the straight edge darkened and the pointed edge not darkened. The other channel-type variants are depicted accordingly.

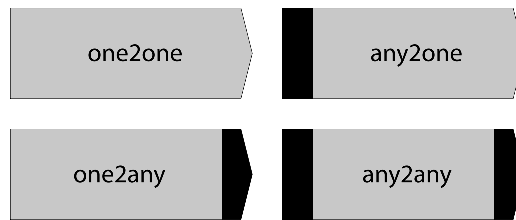

Figure 2.1: Channel-type variants

2.2 Running pony on a Node

On each node of a pony application, the pony environment must be active. This section describes the general mechanisms of how pony operates on a node and how it interacts with the user-level code.

2.2.1 pony-enabled occam-π Programs

The pony environment mainly consists of an occam-π library incorporating pony's functionality. In order to achieve full semantic transparency, however, a small amount of supportive code had to be integrated directly into the KRoC compiler. Details are discussed in Section 6.3.

The compiler support for pony introduces a minor overhead to the handling of channel-types in occam-π programs. Although the additional cost is reasonably small, we want occam-π programmers to be able to choose whether or not to introduce this

overhead to their programs. For this, a new build-time option has been added to KRoC.

If KRoC is built with the '`--with-pony`' option, the compiler support for pony is enabled for occam-π programs compiled with this KRoC build; otherwise traditional occam-π programs are compiled. In the following, we will refer to occam-π programs that are compiled by a '`--with-pony`' KRoC build as *pony-enabled programs*.

Currently, pony-enabled programs and traditional occam-π programs are incompatible as far as the handling of channel-types is concerned. For instance, a library compiled by a traditional KRoC build could not be used by a pony-enabled program, unless the library uses no channel-types. This is not a major drawback at the moment, since any traditional occam-π program (or library) can be re-compiled by a pony-enabled KRoC build without changing its functionality. Only the pony support for handling channel-types, with the small extra cost, would be introduced.

In the future, it would be desirable to make pony-enabled and traditional KRoC builds more compatible. A possible approach is outlined in Section 12.3.2.

2.2.2 The pony Library

In order to make the pony environment available on a node, the node must use the pony library. This is done in the usual occam-π way by adding the following compiler directives to the source code:

```
#INCLUDE "ponylib.inc"
#USE "pony.lib"
```

When the program is being compiled, the following linker options:

```
-lpony -lcif -lcourse -lsock -lfile -lproc
```

must be given to KRoC in order to link the program with the pony library as well as with all libraries that the pony library itself uses.[1] pony uses the CIF library for

[1] It is planned to enable KRoC to recognise the linker options automatically so that they would not have to be given as parameters anymore; this has not been implemented yet, however.

its protocol-converters, and KRoC's course, socket, file and process libraries [Bar00a] for calling various routines that are needed for its functionality.

2.2.3 Public pony Processes and Handles

There is a runtime system which handles the internal functions of pony, called the *pony kernel*. The user-level code of a node interacts with the pony kernel through a set of public pony processes. The number of public processes has been kept to the necessary minimum in order to make the usage of pony as simple and intuitive as possible.

There are public processes for starting the pony kernel, allocating ends of NCTs, shutting down the pony environment, as well as for error- and message-handling. Error-handling is used for the detection of networking errors in pony; message-handling is used for the reporting of status and error messages while pony is active.

The startup process will return a given set of *handles*. A handle is the client-end of a channel-type[2] which is used by the user-level code to interact with the pony kernel. This is done by calling the relevant public process and passing the corresponding handle as a parameter.

Handles returned by the startup process may be shared if this is requested by the user-level code. The user-level code may pass a shared handle to several of its sub-processes, which then need to claim the handle before they can use it for calling a public pony process. This conforms with the general rules for shared channel-type-ends, which makes sense since the handles *are* normal occam-π channel-type-ends.

Apart from the tasks covered by the public processes, all interaction between the user-level code of a node and the pony kernel running on that node is *implicit*. This includes the communication via NCTs between processes on different nodes, the

[2]Please note that in this thesis, the term 'handle' may refer either to the channel-type as such, or to its client-end. Typically, it is clear from the context which of them is meant; in case of doubt, we will refer to 'the handle channel-type' or to 'the client-end of the handle' specifically. The server-end will always be explicitly referred to as 'the server-end of the handle'.

claiming and releasing of shared NCT-ends, as well as the movement of NCT-ends between nodes of an application. All these things are done by the user-level code in exactly the same way as in a traditional (non-networked) occam-π application, which gives us semantic transparency.

By design rule, handles are not allowed to leave their node. That is, they may not be sent to other nodes over NCTs, since this would result in undefined behaviour.

2.3 The Startup Mechanism

The pony environment is started on a node by calling one of pony's *startup processes*. If the startup process completes successfully, it forks off the pony kernel and returns the handles that are needed to call the other public pony processes.

2.3.1 Different Versions of the Startup Process

There are several startup processes with different names, depending on the needs of the node. The name of the startup process specifies which handles it is supposed to return. The following signature[3] describes the naming of the startup processes:

```
pony.startup.(u|s)nh[.(u|s)eh[.iep]][.mh]
```

If the name of the startup process contains '`unh`', an unshared *network-handle* is returned. If it contains '`snh`', the startup process returns a shared network-handle. The network-handle can then be used for calling pony's allocation and shutdown processes; these are described in Sections 3.1 and 3.2.

If the name of the startup process contains '`ueh`' or '`seh`', the startup process returns an (unshared resp. shared) *error-handle* which can then be used by the user-level code to call pony's error-handling processes. An '`iep`' in the name of the startup process means that an *initial error-point* will be returned along with the error-handle.

[3] '`[...]`' means optional, '`|`' is a choice, '`(...)`' is for grouping.

What an error-point is will be explained in Section 4.1.2, which describes pony's error-handling mechanism.

Finally, if the name of the startup process contains '`mh`', a *message-handle* is returned. Message-handles are always unshared. The user-level code must then use the message-handle with one of pony's message-outputters in order to display messages from the pony kernel. Details are given in Section 4.2.

2.3.2 Parameters of the Startup Processes

The different startup processes have different parameters, depending on which handles they are supposed to return. The following parameter list is a superset of all possible parameters:

```
(VAL INT msg.type, net.type,
 VAL []BYTE ans.name, app.name, node.name,
 VAL INT node.type,
 RESULT INT own.node.id,
 RESULT [SHARED] PONY.NETHANDLE! net.handle,
 RESULT [SHARED] PONY.ERRHANDLE! err.handle,
 RESULT INT err.point,
 RESULT PONY.MSGHANDLE! msg.handle,
 RESULT INT result)
```

The order of the parameters is the same for all startup processes. Depending on the name of the startup process, certain parameters may be unused, however. '`[SHARED]`' means that it depends on the startup process whether the parameter is '`SHARED`' or not.

The following parameters are common to all startup processes:

- '`net.type`' is the network-type. At the moment, the only supported network-type is TCP/IP. This is expressed by passing the '`PONYC.NETTYPE.TCPIP`' constant to the '`net.type`' parameter.

- '`ans.name`' is the name of the ANS. This can either be an empty string, or a string consisting of the following allowed characters: letters, digits, dash, dot and underscore. The ANS-name determines which ANS is contacted by the node. Details about this are given in Section 3.3.2.
- '`app.name`' is the name of the pony application to which the node belongs. Under this name, the application is administrated by the ANS. Any string is allowed as the application-name except the empty string.
- '`node.name`' is the name of the node. The same characters are allowed as for the ANS-name. The node-name determines which configuration file is used by pony to resolve the network location of the node. Details are given in Section 3.3.1.
- '`node.type`' is the type of the node. The following values are allowed:
  - '`PONYC.NODETYPE.MASTER`' means that the node is the master node of the application. If the ANS has another master node for this application stored already, the startup process will fail.
  - '`PONYC.NODETYPE.MASTERRESET`' also means that the node is the master node. However, if the ANS has another master node for this application stored already, the entire application will be reset in the ANS, and the requesting node will be made the master of the (new instance of the) application. Unless other errors occur, the startup process always succeeds in this case. This mechanism can be used for recovery after an error has occurred within an application. In this way, an application can be reset in the ANS without the need to shut down and restart the ANS itself.
  - '`PONYC.NODETYPE.SLAVE`' means that the node is a slave node. If the ANS has no master node for this application stored yet, the startup process will fail.
  - '`PONYC.NODETYPE.SLAVEWAIT`' also means that the node is a slave node. If the ANS has no master node for this application stored yet, it will wait

until a master node signs up for this application. Once that happens, the ANS will notify the waiting slave node, and the startup process can finish.

- '`result`' is the result returned by the startup process upon completion. If the startup process completes successfully, the '`result`' parameter will return '`PONYC.RESULT.STARTUP.OK`', otherwise it will return an error. There are several '`PONYC.RESULT.STARTUP.*`' constants for errors that can occur during startup; these are presented in detail in Appendix B.2.6. The implementation of the startup processes is explained in Section 10.4.

- If the startup process completes successfully, '`own.node.id`' returns the ID of the node. Each node of an application is assigned a unique ID. The node-ID of the master is always 0. The master assigns node-IDs to the slaves in the order in which the slaves connect to the master, starting with 1. Please note that the knowledge of the own node-ID is not needed for the function of the pony node; the node-ID is only returned for debugging purposes.

- Finally, if the startup process completes successfully, '`net.handle`' will contain the network-handle. It will be unshared or shared, depending on which startup process is used.

The following parameters are related to pony's error-handling mechanism (described in detail in Section 4.1):

- If the name of the startup process contains '`ueh`' or '`seh`', and if the startup process completes successfully, '`err.handle`' will contain the (unshared resp. shared) error-handle.

- If the name of the startup process contains '`iep`', and if the startup process completes successfully, '`err.point`' will contain the initial error-point.

The following parameters are related to pony's message-handling mechanism (described in detail in Chapter 4.2):

- If the name of the startup process contains '`mh`', the message-type must be passed to '`msg.type`'. There are the following valid message-types: '`PONYC.MSGTYPE.STATUS`' means that the pony kernel will output only status messages. '`PONYC.MSGTYPE.ERR`' means that only error messages are output. Finally, '`PONYC.MSGTYPE.STATUSERR`' tells the pony kernel to output both status and error messages.

- If the name of the startup process contains '`mh`', and if the startup process completes successfully, '`msg.handle`' will contain the message-handle.

2.3.3 Design Rules

There are certain design rules that must be followed in order to ensure the correct function of pony applications. As mentioned already, none of the handles is allowed to be sent to another node. Handles are relevant only to the node that has created them.

As far as the startup of pony is concerned, the general design rule is that on each node, the pony environment is only started once, i.e. that each node only belongs to one pony application.[4] The reason for this design rule is to avoid cases where NCT-ends that belong to one pony application are sent to a node that belongs to another pony application, which would result in undefined behaviour.

As an exception to this general rule, it *is* possible to write pony-enabled occam-π programs that act as a 'bridge' between pony applications. Such a program could be used to exchange data-items of any type, but not NCT-ends, between the different applications. Extra careful programming would therefore be required. Such a 'bridging node' would need to start a pony environment separately for each application, and use separate handles for the different applications. It would be vital not to mix up NCTs of separate applications. That is, no NCT-ends of one application may be

[4]Please recall that by 'node' we mean a pony-enabled occam-π program, not a physical computer. The latter may run many nodes at the same time.

sent to nodes of a different application. As long as this is ensured, a 'bridging node' will function properly.

Another design rule concerns the general order of events regarding pony, namely the startup, the usage and the shutdown of pony. This will be examined in detail in Section 3.2.

2.4 Starting the ANS

As discussed in Section 2.1.3, the ANS may administrate many different applications. Each node of a given application must know the network location of the ANS by which the application is administrated. The ANS itself is a pre-compiled occam-π program coming with KRoC. It is placed in the '`bin`' directory of the KRoC distribution; the same place where the '`kroc`' command itself is located. The ANS for TCP/IP can be started by calling:

```
ponyanstcpip
```

provided that KRoC's '`bin`' directory is in the path of the current shell. The ANS can be configured with its own configuration file; see Section 3.3.3 for details. The implementation of the ANS is discussed in Section 10.3.

Chapter 3

Operation of pony Nodes

3.1 Allocating NCT-ends

The basic communication paradigm in pony are network-channel-types, i.e. channel-types whose ends may reside on more than one node. The process of establishing a new NCT in a pony application is called *allocation*. There are two ways of allocating NCTs. The first possibility is to allocate the ends of an NCT *explicitly*, using one of pony's allocation processes. The other possibility is to send an end of a previously non-networked channel-type to another node. By doing this, the channel-type becomes networked and thus, a new NCT is established in the pony application *implicitly*.

3.1.1 Explicit Allocation

NCT-ends are allocated explicitly by using a name that is unique for the NCT across the entire pony application. This name is a string under which the master node of the application administrates the NCT. The several ends of an NCT can be allocated on different nodes using this unique NCT-name. The NCT-name is a string which is passed as a parameter to pony's allocation processes; it is *not* the variable name of the channel-type-end that is allocated. The variable name may be different for different

ends of the NCT, and may change over time (by assignment and communication) — as usual for occam-π variables.

There are four different allocation processes whose names have the following signature:

```
pony.alloc.(u|s)(c|s)
```

If the name of the allocation process contains '`uc`', it is the process for allocating an unshared client-end of an NCT. The names of the allocation processes for shared client-ends, unshared and shared server-ends contain '`sc`', '`us`' or '`ss`' accordingly. Please note that any end of an NCT may be allocated *at any time*. There is no prerequisite (such as for instance in JCSP.net) that a client-end may only be allocated when a server-end has been allocated first, or similar restrictions.[1] In pony, this characteristic has been 'moved up' to the application level and now applies to the slaves and to the master (although the 'slave/wait' mechanism allows to start a slave node before the master; it just waits in this case, cf. Section 2.3.2).

The parameters of the allocation processes are essentially the same; the only difference is the channel-type-end that is to be allocated. This is the parameter list of the allocation processes:

```
(PONY.NETHANDLE! net.handle,
 VAL []BYTE nct.name, VAL INT other.end.type,
 RESULT <alloc-type> chan.type.end, RESULT INT result)
```

- '`net.handle`' is the network-handle.
- '`nct.name`' is the name of the NCT to which the end belongs that is to be allocated. Under this name, the NCT is administrated by the master node of the application. Any string is allowed as the NCT-name except the empty string.

[1] In JCSP.net, this prerequisite would apply to writing-ends and reading-ends of network-channels rather than client-ends and server-ends of NCTs.

- '`other.end.type`' is the *share-type* of the other end of the NCT, i.e. of the server-end if a client-end is to be allocated and vice versa. Valid values are '`PONYC.SHARETYPE.UNKNOWN`' if we do not know or do not care about whether the other end is shared or not, '`PONYC.SHARETYPE.UNSHARED`' if the other end is meant to be unshared, or '`PONYC.SHARETYPE.SHARED`' if the other end is meant to be shared. Any mismatches will cause the allocation process to return an error and fail. This would be the case for instance if on one node we allocate a client-end of an NCT where we specify the share-type of the server-end to be 'unshared', and later we try to allocate a shared server-end of that NCT on some other node. In this case, the allocation of the server-end would fail because there is a mismatch.

- '`chan.type.end`' is the channel-type-end variable that is to be allocated. '`<alloc-type>`' is a wildcard for the type of the variable. It would be '`MOBILE.CHAN!`' for the '`uc`' version, '`SHARED MOBILE.CHAN!`' for the '`sc`' version, '`MOBILE.CHAN?`' for the '`us`' version, or '`SHARED MOBILE.CHAN?`' for the '`ss`' version.

- '`result`' is the result returned by the allocation process. If the allocation is successful, the '`result`' parameter will return '`PONYC.RESULT.ALLOC.OK`', otherwise it will return an error. The following errors may occur:

  - '`PONYC.RESULT.ALLOC.ILLEGALNCTNAME`': The NCT-name is illegal.
  - '`PONYC.RESULT.ALLOC.CHANTYPEMISMATCH`': There is a mismatch in the type of the channel-type. This means that earlier on, another channel-type-end was allocated under the same name which was of a different type from the end that we try to allocate now. For instance, if after having allocated a client-end of type '`FOO!`' on some node, we try to allocate a server-end of type '`BAR?`' under the same name, this error will be triggered.

- 'PONYC.RESULT.ALLOC.X2XTYPEMISMATCH': There is a mismatch in the x2x-type between the channel-type-end that we try to allocate and a previously allocated one. This error occurs for instance, if after a client-end of an NCT was allocated with the share-type of the server-end being specified as 'unshared', we try to allocate a shared server-end of that NCT.
- 'PONYC.RESULT.ALLOC.X2XCOUNTMISMATCH': This error occurs if although there is no mismatch in the x2x-type as in the previous error, there is a mismatch in the *x2x-count*, i.e. if we try to allocate more than one one2x client-end or more than one x2one server-end.

3.1.2 Usage of NCTs and Implicit Allocation

Once an NCT-end variable has been allocated, it may be used like any other channel-type-end variable. From the point of view of the user-level code, the usage is semantically transparent. This includes the possibility of sending a channel-type-end along a channel.

If the channel over which we want to send a channel-type-end is inside an NCT whose opposite end is on another node, the channel-type-end that we send will end up on that node as well. There are two possibilities now — either the channel-type to which the end that is to be sent belongs is already networked, or not. The latter means that the channel-type-end was originally allocated on our node in the traditional way, together with its opposite end.

If the channel-type is not yet networked, it becomes networked during the send operation. This implicit allocation happens internally and is transparent to the user-level code. The pony environment becomes aware of the new NCT and will henceforth treat it just like an explicitly allocated one. The only difference is that implicitly allocated NCTs have no NCT-name, which means that no other ends of that NCT may be allocated explicitly. This is not necessary, however, since the NCT had originally been allocated in a client-end/server-end pair anyway. If one or both of

its ends are shared, the relevant channel-type-end variable may be multiplied by simply assigning it to another variable or sending it over a channel — as usual for channel-types.

The second possibility is that the channel-type-end that is to be sent belongs to an NCT already, i.e. the pony environment is already aware of this NCT. This may apply to both explicitly and implicitly allocated NCTs. In this case, no prior implicit allocation is done by the pony environment before the end is sent to the target node.

When an end of an NCT arrives on a node where no end of that NCT has been before during the lifetime of the pony application, the NCT-end is established on the target node by the pony infrastructure.[2] Again, this may apply to both explicitly and implicitly allocated NCTs.

In summary, apart from the actual explicit allocation itself, there is no difference between explicitly and implicitly allocated NCTs from the point of view of the user-level code. Any operation that can be done with channel-types can be done with both explicitly and implicitly allocated NCTs as well.

Details about the implementation of explicit and implicit allocation and the movement of NCT-ends can be found in Part II of this thesis.

3.2 Shutting Down Nodes

At the end of a pony-enabled program, the pony environment must be shut down. This is done by calling the pony shutdown process. The only parameter of the shutdown process is the network-handle:

```
PROC pony.shutdown (PONY.NETHANDLE! net.handle)
```

[2]Future research may enhance pony's performance by not establishing the entire infrastructure needed for an NCT-end if the end is just 'passing through' a node and never used for communication on the node itself. Details are given in Section 12.2.6.

By design rule, the pony shutdown process may only be called after all usage of networked (or possibly networked) channel-type-end variables has finished. 'Usage' here means:

- claiming/releasing the channel-type-end if it is shared
- using the channel-type-end for communication over its channels (either way)

The occam-π programmer must make sure that none of the above is happening in parallel with (or after) calling the shutdown process. Of course, the user-level code may use channel-types in parallel with or after calling '`pony.shutdown`', but the programmer must ensure that none of these channel-types are networked. Typically, calling '`pony.shutdown`' would be the very last thing the node does, except possibly checking for errors via the error-handling processes (cf. Section 4.1) or waiting for the last few messages from the message-handle (cf. Section 4.2) — none of which involves *networked* channel-type-ends. Algorithm 3.1 shows the typical structure of a pony node.

Algorithm 3.1: Typical structure of a pony node

```
SEQ
  ...  Do stuff
  pony.startup... (...)
  ...  Do stuff like forking off the message-outputter
  FORKING
    ...  Do stuff using NCTs, i.e.
    ...      * allocate them
    ...      * claim/release them
    ...      * communicate over their channels
    ...      * fork off processes that are using NCTs
  ...  Do stuff
  pony.shutdown (...)
  ...  Do stuff like a final check for errors and shutting down
  ...    the error-handler
```

The '`FORKING`' block is there to ensure that all processes using NCTs that might have been forked off have finished before the shutdown process is called. Of course,

if no processes are forked off during the node's lifetime, the '`FORKING`' barrier would not be necessary. Please note that the above layout is just a general suggestion. Programmers may organise the user-level code differently if they wish, as long as the general order of pony-related events is preserved.

The shutdown process tells the pony kernel to shut down, which includes shutting down all its components. If our node is the master node of the application, the pony kernel also notifies the ANS about the shutdown, which will then remove the application from its database. This will prevent any further slave nodes from connecting to the master. On slave nodes, the shutdown process finishes immediately after the pony infrastructure on that node has been shut down. On the master node, the pony kernel waits for all slaves to shut down before shutting down itself.

3.3 Configuration

The configuration of the pony environment depends on the network-type that is used. Apart from the networking settings, no configuration is needed by pony. This section is concerned with the configuration for TCP/IP (which is currently the only supported network-type) on Linux/x86 (which is currently the only platform on which pony runs).

Since a node must be able both to contact other nodes and the ANS *and* to be contacted by other nodes and the ANS, it is vital that the node can be contacted via the same IP address/ port number from all computers involved in the pony application (i.e. all computers that are running nodes or the ANS). This includes the computer on which the node itself is running. Therefore, topologies with Network Address Translation between computers involved in the application are not supported at the moment. Please note that if *all* computers involved in the application are located on a sub-network that uses NAT to communicate with the outside world, the NAT has no impact on the pony application. Similarly, if there is only one computer involved in the application (i.e. all nodes and the ANS are running on the same computer),

the loopback IP address may be used to identify the location of nodes and the ANS; in this case only the ports would be different.

pony's network-specific components are configured using simple plain-text configuration files that contain the relevant settings. Settings may be omitted, in which case either defaults are used or the correct setting is detected automatically. There are three different configuration files, which are discussed in the following sections.

3.3.1 The Node-file

During startup, a node-name must be supplied to the startup process (cf. Section 2.3.2). This name may be an empty string, or a string consisting of letters, digits, dash, dot and underscore. This name is used to determine the name of the configuration file that is used to resolve the location of the node on the network (the *node-file*). In TCP/IP terms, 'location' means the IP address and port number over which the node can be contacted by other nodes or by the ANS. If the node-name is an empty string, the name of the node-file is:

```
.pony.tcpip.node
```

Otherwise it is:

```
.pony.tcpip.node.<node-name>
```

where '`<node-name>`' is the name of the node. The startup process will look for the node-file first in the directory from which the node is started; if the node-file is not there, the startup process will look in the user's home directory. If the node-file is found, the startup process will look for the following settings in the node-file:

```
ip=<ip-address>
port=<port-number>
```

where '`<ip-address>`' is the IP address (in standard notation) and '`<port-number>`' is the port number under which the node can be contacted. This IP address/ port

number pair is used as a unique identification for the node's location across the entire application.

If no node-file is found, or if one or more of the settings are missing in the node-file, the relevant settings will be determined automatically by the startup process. If no IP address is found, the startup process will attempt to resolve the default outgoing IP address of the computer. If this is not possible, the startup process will fail. If no port number is found, pony will automatically assign the first free port that is greater or equal to port 7500, the default port number for pony nodes. With this mechanism, it is possible to run several pony nodes on the same physical computer and use the same node-name for all of them. If the port number is not specified in the corresponding node-file, pony automatically chooses the next free one.

It is possible to run pony nodes on computers which get their IP address via DHCP, as long as the current IP address can be resolved (which should normally be no problem). Since the application does not know (and does not need to know) about the location of a node until the node effectively joins the application, computers with variable IP addresses do not present a problem.

3.3.2 The ANS-file

Similarly to the node-name, the name of the ANS must be given to the pony startup process. The same rules as for the node-name apply to the validity of the ANS-name. The ANS-name is used to determine the name of the *ANS-file*, which is used to find out the location of the ANS. The name of the ANS-file is either:

```
.pony.tcpip.ans
```

if the ANS-name is an empty string; otherwise it is:

```
.pony.tcpip.ans.<ans-name>
```

where '`<ans-name>`' is the ANS-name. Again, the startup process will look for the ANS-file first in the current directory and then in the user's home directory. If the

ANS-file is found, the startup process will look for the following settings in the ANS-file:

```
host=<hostname-or-ip-address>
port=<port-number>
```

where '`<hostname-or-ip-address>`' is either the hostname or the IP address (in standard notation) and '`<port-number>`' is the port number under which the ANS can be contacted.

If no ANS-file is found, or if one or more of the settings are missing in the ANS-file, the startup process will use default settings instead. If no hostname is found, the startup process will use the loopback IP address to try to contact the ANS — which will fail if the ANS is not running on the same computer as the node itself. If no port number is found, port 7400 will be used as the default port number for the ANS.

The location of the ANS must be known by all nodes in order to be able to start the pony application. Therefore, running the ANS on a computer using DHCP is not advisable. An exception might be static DHCP configurations where the computer running the ANS is always assigned the same hostname/ IP address by the DHCP server.

3.3.3 The ANS-configuration-file

The last file is the *ANS-configuration-file*, which is used by the ANS to find out its own port number.[3] The name of the ANS-configuration-file is:

```
.pony.tcpip.ans-conf
```

Again, the file is searched for in the current and in the home directory. If it is found, the following setting is looked up:

[3]The ANS does not need to know its own IP address, since it never notifies any nodes about it at runtime — nodes find the ANS via the ANS-file.

```
port=<port-number>
```

where '<port-number>' is the port number under which the ANS listens for connections. If the file or the setting are not found, the default ANS port of 7400 is used.

Chapter 4

Error-handling and Message-handling

4.1 Error-handling

4.1.1 Non-transparent Error-handling

'Error-handling' in pony refers to the detection of networking errors that occur during the operation of the pony application. All other errors that may occur with respect to the pony environment are errors during the call of one of pony's public processes. As already pointed out in the previous chapters, these errors are handled by returning the relevant error via the '`result`' parameter of the respective process. Networking errors, on the other hand, cannot be handled by result parameters, because they may occur at arbitrary points in time during the operation of the pony application.

This raises the question of error- and exception-handling in occam-π (and similar parallel processing architectures) in general — a topic that is still under ongoing discussion in the parallel processing community. Although there have been several interesting approaches, such as Hilderink's [Hil05], there is no 'common model' for error- or exception-handling in parallel processing architectures yet. Also occam-π lacks a

general fault tolerance mechanism. Whether ideas such as Barnes' 'TRY'/'CATCH' proposal [Bar03] or a similar approach will be implemented in the near future, cannot be foreseen at the moment.

Since there is currently no built-in error-handling or fault-tolerance mechanism in occam-π on which pony's error-handling could be built, we have chosen an approach specific to the pony environment. Therefore, error-handling is the only part of pony that is not semantically transparent in the way defined earlier in this thesis. Programmers of pony nodes have two options now. They may either do it the 'traditional' occam way, namely trust that the application is error-free and live with the consequences otherwise, or use pony's (non-transparent) error-handling mechanism to check for errors.

4.1.2 pony's Error-handling Mechanism

If the startup process is requested to return an error-handle, the pony kernel starts an *error-handler* which keeps track of errors that occur during the lifetime of the node. Every time a networking error happens on the node, the error-handler is notified about it and stores it. The user-level code may check for errors kept by the error-handler by using pony's error-handling processes together with the error-handle.

Please note that pony's error-handling mechanism was specifically designed for the *detection* of networking errors, not for fixing them. The absence of a general fault-tolerance mechanism in occam-π as a base for pony's error-handling makes it very difficult to develop a mechanism that allows to recover from networking errors in *all* possible situations where they might occur. It is therefore up to the user-level programmer to react to errors appropriately. This may involve using existing patterns such as poisoning [Wel89].

pony's error-handling allows the user-level code to check for errors that happened after a given point in time. This point is called an *error-point*. The general mechanism is that the user-level code asks the error-handler for an error-point *before* the

occurrence of an event that shall be watched for errors. The event may be anything the programmer wishes, be it communication over NCTs, allocating new NCT-ends, etc. The user-level code may now at any given time check for errors that happened after the error-point, i.e. after the event started.

Algorithm 4.1 shows an example where a networking event is watched for errors. Please note that this is just an example. There may be situations where a timeout of ten seconds, or even a timeout in general, would not be appropriate. pony's error-handling processes have specifically been designed as a *general* tool for the detection of errors — how to use them is up to the user-level code.

Algorithm 4.1: An error-handling example

```
SEQ
  ...  Get error-point from error-handler
  CHAN BOOL sync:
  PAR
    SEQ
      ...  Event that is to be watched
      sync ! TRUE       -- Event has finished, synchronising
    TIMER tim:
    INT t:
    SEQ
      tim ? t           -- Prepare timer
      PRI ALT
        BOOL any:
        sync ? any      -- Event has finished successfully
          ...  Prepare to continue normally
        tim ? AFTER t PLUS 10000000      -- 10 seconds timeout
          SEQ
            ...  Get errors that happened after the error-point
            ...  React accordingly
```

A special error-point is the so-called *initial error-point* which may be returned by the startup process if requested. The purpose of the initial error-point is to cover errors that may occur right from the startup of the pony environment. There are networking events happening even before the startup process finishes. It would not be possible for the user-level code to get an error-point *before* those events via the error-handling processes; hence it is returned by the startup process itself. Once the

user-level code has acquired it, the initial error-point is a normal error-point just like the ones acquired later.

Errors returned by the error-handler are records that contain specific information about the error that occurred. The layout of the error record is as follows:

```
DATA TYPE PONY.ERROR
  RECORD
    BOOL ans.concerned:
    BOOL master.concerned:
    BOOL remote.node.concerned:
    INT remote.node.id:
    INT err.code:
:
```

The fields of the record are fairly self-explanatory. The three Boolean flags denote whether the error had anything to do with the ANS, the master or a remote node[1]. '`remote.node.id`' contains the ID of the remote node if applicable. Finally, '`err.code`' contains the error-code. For TCP/IP (which is currently the only supported network-type), this may be one of the constants set out in Appendix B.2.9.

4.1.3 Getting Information About NCTs and Remote Nodes

In order to evaluate the '`remote.node.id`' field of a returned error properly, we need to be able to find out which remote nodes might be involved in a networking event. Between slave nodes, network communication may only happen while the link between them is established or shut down, or when there is a *session* of an NCT between the two nodes. The latter means that the NCT is stretched between them for communication, i.e. the client-end of the NCT is located on one of the nodes, the server-end on the other, and each of the ends is either unshared, or shared and

[1] Note that the master may be a remote node too.

claimed on the respective node. Any other network communication may only happen between a slave and the master, or between the master and the ANS during shutdown.

With pony's error-handling processes, it is possible to find out about the *current remote node* of an NCT-end. This is the node on which the other end of the NCT is currently located (if unshared) or claimed (if shared). Finding out about the current remote node of an NCT-end is done in two steps. The first step is to find out the ID of the NCT to which the end belongs. Each NCT has a unique ID across the entire pony application which is assigned when the NCT is allocated. This applies both to explicitly and implicitly allocated NCTs. Once the NCT-ID has been assigned, it does not change during the lifetime of the NCT. Finding out the NCT-ID for an NCT-end would typically be done right after its (explicit or implicit) allocation.

There are four processes for finding out the NCT-ID whose names have the following signature:

```
pony.err.get.nct.id.(u|s)(c|s)
```

The naming follows the same scheme as for the allocation processes. That is, there are processes for unshared/shared client-ends/server-ends. Apart from the channel-type-end that is to be checked, the parameters are the same for all four processes:

```
(<end-type> chan.type.end,
 RESULT INT nct.id, result)
```

- '`chan.type.end`' is the channel-type-end variable that is to be checked. '`<end-type>`' is a wildcard for the type of the variable. It would be '`MOBILE.CHAN!`' for the '`uc`' version, '`SHARED MOBILE.CHAN!`' for the '`sc`' version, '`MOBILE.CHAN?`' for the '`us`' version, or '`SHARED MOBILE.CHAN?`' for the '`ss`' version.
- '`result`' is the result returned by the process. It can be one of the following constants:

  - 'PONYC.RESULT.ERR.GNI.OK': The process completed successfully. In this case, 'nct.id' now contains the NCT-ID for the channel-type-end that was checked.
  - 'PONYC.RESULT.ERR.GNI.CTENDUNDEFINED': The channel-type-end variable was undefined.
  - 'PONYC.RESULT.ERR.GNI.CTENDNOTNETWORKED': The channel-type belonging to the channel-type-end variable was not networked.

Knowing the ID of a given NCT, we may now at any time ask the error-handler about its current remote node. This is done with the following process:

```
PROC pony.err.get.current.remote.node
            (PONY.ERRHANDLE! err.handle,
             VAL INT nct.id,
             RESULT INT remote.node.id, result)
```

- 'err.handle' is the error-handle.
- 'nct.id' is the ID of the NCT that is to be checked.
- The 'result' can be one of the following constants:
  - 'PONYC.RESULT.ERR.GCRN.OK': The process completed successfully. If this is the case, 'remote.node.id' now contains the ID of the current remote node of the checked NCT.
  - 'PONYC.RESULT.ERR.GCRN.INVALIDNCTID': The NCT-ID is invalid. This does not necessarily mean that there is no NCT with this ID anywhere in the pony application, but that on *this* node, there has never been an NCT-end with the given NCT-ID since the application has started.
  - 'PONYC.RESULT.ERR.GCRN.NOSESSION': There is currently no session between this node and a remote node for the given NCT.

   – 'PONYC.RESULT.ERR.GCRN.SAMENODE': There is currently a session for the given NCT, but both of its ends are on the same node (namely this one).

4.1.4 Getting and Deleting Error-points

To get a new error-point from the error-handler, the following process must be called:

```
PROC pony.err.new.err.point
            (PONY.ERRHANDLE! err.handle,
             RESULT INT err.point)
```

where 'err.handle' is the error-handle. The new error-point will be returned by 'err.point'.

If an error-point is no longer needed, it should be deleted. This is done by calling the following process:

```
PROC pony.err.delete.err.point
            (PONY.ERRHANDLE! err.handle,
             VAL INT err.point,
             RESULT INT result)
```

Again, 'err.handle' is the error-handle. 'err.point' is the error-point that is to be deleted. The 'result' will be either 'PONYC.RESULT.ERR.DEP.OK' if the error-point has been deleted successfully, or 'PONYC.RESULT.ERR.DEP.INVALIDERRPOINT' if the error-point was invalid.

The main purpose for deleting an error-point is to save the error-handler from unnecessary work. Internally, the error-handler only keeps errors if there was an error-point beforehand, otherwise the errors are discarded. This makes sense, since without an error-point, the errors cannot be retrieved by the user-level code. So, if an error-point that is no longer needed is not deleted, the error-handler will keep accumulating errors that may (if there were no other error-points beforehand) never be retrieved. For the user-level code itself, not deleting error-points that are no longer

needed makes no practical difference, because errors that have not been retrieved are discarded automatically when the error-handler is shut down. It is more a matter of 'clean programming', such as freeing memory that is no longer used in C programs.

4.1.5 Getting Errors That Happened After a Given Error-point

The following process will retrieve errors that happened after a given error-point:

```
PROC pony.err.get.errs.after
            (PONY.ERRHANDLE! err.handle,
             INT err.point,
             VAL BOOL check.ans, check.master, check.all.nodes,
             VAL []INT nodes.to.check,
             RESULT MOBILE []PONY.ERROR err.array,
             RESULT INT result)
```

- '`err.handle`' is the error-handle.
- '`err.point`' is the error-point. If the process completes successfully, the value of the error-point will be adapted. Logically, the error-point is 'moved' to the current point in time. If it is used again later to retrieve errors, the error-handler will then return errors that happened after now.
- The '`check.ans`' and '`check.master`' flags tell the error-handler whether to return errors that involve the ANS or the master node of the application.
- '`nodes.to.check`' is an array of node-IDs. The error-handler will return errors involving these nodes. If the '`check.all.nodes`' flag is '`TRUE`', all errors involving remote nodes are returned; in this case the content of '`nodes.to.check`' is irrelevant.
- The returned '`result`' will be either '`PONYC.RESULT.ERR.GEA.OK`', in which case '`err.array`' will return all errors that happened after '`err.point`' and fit

the above criteria, or '`PONYC.RESULT.ERR.GEA.INVALIDERRPOINT`' if the error-point was invalid.

4.1.6 Shutting Down the Error-handler

Unlike the other internal components of the pony kernel, the error-handler is not shut down automatically when the '`pony.shutdown`' process is called. Instead, it must be shut down manually by calling a special shutdown process, whose only parameter is the error-handle:

```
PROC pony.err.shutdown (PONY.ERRHANDLE! err.handle)
```

The reason for having a separate shutdown process is that while the pony environment is being shut down, there are still networking events happening which may fail. Hence, the user-level code may want to check for errors during (i.e. in parallel with) or after the shutdown of the pony kernel.

After the '`pony.shutdown`' process has been called, it is not allowed to call '`pony.err.get.nct.id.*`' or '`pony.err.get.current.remote.node`' any longer, because the infrastructure necessary for calling them is not in place anymore. It would not make much sense to call them at this point anymore anyway, since as explained in Section 3.2, NCTs are not allowed to be used anymore after the pony kernel has been shut down. All other error-handling processes may still be used after the shutdown of the pony kernel, up until the error-handler is shut down itself.

4.2 Message-handling

'Message-handling' in pony refers to the reporting of status and error messages by the pony kernel. It is intended for debugging purposes and *no* requirement for the actual function of the pony environment.

If requested during the startup of the pony environment on a node, the pony kernel reports status and/or error messages (cf. Section 2.3.2). Status messages are there

primarily for debugging and testing purposes. They inform about important events during the operation of the pony environment, such as the communication between the internal components of the pony kernel, the forking off of new components, and networking events. Error messages are reported when networking errors occur, i.e. whenever the error-handler would be notified about an error as well.

If the startup process is requested to return a message-handle, the pony kernel starts a *message-handler*. The internal components of the pony kernel send all messages to the message-handler, which buffers them until they are output. This mechanism is similar to how the error-handler works.

The actual output of the messages is performed by pony's *message-outputters*. Unlike the error-handling processes, a message-outputter is called once, and then autonomously outputs all messages it gets from the message-handler until the pony kernel, and the error-handler if applicable, have been shut down. After the last message has been taken by the message-outputter, the message-handler shuts itself down automatically. The same applies to the message-outputter after the last message has been output.

Typically, a message-outputter would run in parallel with everything else the node does that might trigger messages from the pony environment. It should be either forked off after the startup of the pony kernel, or put in a 'PAR' construct to run in parallel with the rest of the node's function.

There are eight different message-outputters whose names have the following signature:

```
pony.msg.out.(u|s)(o[.(u|s)e]|e)
```

There are message-outputters with unshared/shared output channels (for status messages) and/or unshared/shared error channels (for error messages) — depending on which kind(s) of messages are supposed to be reported. Output and error channels are ordinary occam-π channels carrying 'BYTE's. Although the message-outputters may be plugged into any 'BYTE' channels the programmer wishes, it would typically

be the standard output and standard error channels declared in the header of the node's main process.

The different message-outputters have different parameters, depending on which channels are meant to be plugged in. The following parameter list is a superset of all possible parameters:

```
(PONY.MSGHANDLE! msg.handle,
 [SHARED] CHAN BYTE out!,
 [SHARED] CHAN BYTE err!)
```

Depending on the name of the message-outputter, one of the channels may be not used. '`[SHARED]`' means that it depends on the message-outputter whether the channel is '`SHARED`' or not.

- '`msg.handle`' is the message-handle.
- If the name of the message-outputter contains '`uo`' or '`so`', '`out`' is the (unshared resp. shared) output channel for pony's status messages.
- If the name of the message-outputter contains '`ue`' or '`se`', '`err`' is the (unshared resp. shared) error channel for pony's error messages.

Chapter 5

A Sample Application

This chapter presents a sample pony application in order to enable a better understanding of what has been discussed so far. This sample application has purposely been kept simple. The idea is to draw the attention of the reader to the interplay of the different aspects of the pony environment, rather than presenting a very realistic but unnecessarily complex application. Therefore, parts of the code that are not directly related to pony are usually folded in the sample algorithms.

The sample application consists of three types of nodes. The master node is a *broker* that establishes connections between *worker* nodes and *customer* nodes. The workers provide some service for the customers. Both workers and customers connect to the broker via an explicitly allocated NCT, the *broker-handle*. When a worker becomes ready, it passes the client-end of a channel-type (the *worker-handle*) to the broker; the worker itself holds the server-end of the worker-handle. When the client-end of the worker-handle is sent to the broker for the first time, it becomes implicitly networked.

The broker keeps the client-ends of the worker-handles in a database. When a customer needs the service of a worker, it notifies the broker, which then passes a worker-handle from its database to the customer if there is one available. The customer and the worker can now communicate over the worker-handle about the service needed by the customer. When the transaction between the customer and

the worker is finished, the customer sends the client-end of the worker-handle back to the worker over the worker-handle itself. The worker can then re-register with the broker.

Algorithm 5.1 shows the declarations of the handles and the protocols that are carried by the channels inside the handles. These declarations are in an include file that will be included by the three nodes. Algorithms 5.2 through 5.4 show the implementation of the broker, worker and customer nodes.

For the sake of simplicity, the broker and the worker are running infinitely. Only the customer node terminates. To keep the algorithms short, pony's error-handling and message-handling are only demonstrated in the customer node. Especially the demonstration of the error-handling is very simple — the entire operation of the node is guarded by a single timeout and watched for errors collectively. Typically, the user-level code would watch the individual networking events for errors separately. For an example timeout/ error watching mechanism, please refer to Algorithm 4.1.

Figure 5.1 shows a possible layout of the sample application. Since the topology of the application changes dynamically, the figure can only be a 'snapshot' of a given point in time. There are seven nodes altogether, namely the broker, three workers and three customers.[1] All workers and customers are connected to the broker via the broker-handle. Customer 1 currently holds the worker-handle connecting to worker 1; the other customers have not acquired a worker-handle yet. Worker 2 may have just started and not yet registered with the broker, or just finished the service for a customer but not yet re-registered with the broker. Therefore, worker 2 currently holds the client-end of its worker-handle itself. Finally, worker 3 is currently registered with the broker, which holds the relevant worker-handle.

[1]For the sake of simplicity, nodes and processes are depicted as a single box, because in this sample application, on each node there is only the main process. Generally, it is important to distinguish between nodes and processes, since many processes may run on the same node.

Algorithm 5.1: Sample application: Declarations

```
-- Filename: 'decls.inc'

-- Forward declaration
CHAN TYPE WORKERHANDLE:

-- To broker
PROTOCOL BROKERHANDLE.TO.BROKER
  CASE
    -- Register worker
    reg.worker; WORKERHANDLE!
    -- Get worker
    get.worker
:
-- From broker
PROTOCOL BROKERHANDLE.FROM.BROKER
  CASE
    -- No worker available
    no.worker.available
    -- Return worker-handle
    get.worker.confirm; WORKERHANDLE!
:
-- Broker-handle
CHAN TYPE BROKERHANDLE
  MOBILE RECORD
    CHAN BROKERHANDLE.TO.BROKER to.broker?:
    CHAN BROKERHANDLE.FROM.BROKER from.broker!:
:

-- To worker
PROTOCOL WORKERHANDLE.TO.WORKER
  CASE
    ...  Stuff dealing with the service provided by the worker
    -- Finish transaction and return worker-handle
    finish; WORKERHANDLE!
:
-- From worker
PROTOCOL WORKERHANDLE.FROM.WORKER
  CASE
    ...  Stuff dealing with the service provided by the worker
:
-- Worker-handle
CHAN TYPE WORKERHANDLE
  MOBILE RECORD
    CHAN WORKERHANDLE.TO.WORKER to.worker?:
    CHAN WORKERHANDLE.FROM.WORKER from.worker!:
:
```

Algorithm 5.2: Sample application: The broker

```
#INCLUDE "decls.inc"
#INCLUDE "ponylib.inc"
#USE "pony.lib"

PROC broker (CHAN BYTE key?, scr!, err!)
  INT own.node.id, result:
  PONY.NETHANDLE! net.handle:
  BROKERHANDLE? broker.handle.svr:
  SEQ
    -- Start pony
    pony.startup.unh (PONYC.NETTYPE.TCPIP, "", "sample-app",
                      "", PONYC.NODETYPE.MASTER,
                      own.node.id, net.handle, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    -- Allocate server-end of broker-handle
    pony.alloc.us (net.handle, "broker-handle", PONYC.SHARETYPE.SHARED,
                   broker.handle.svr, result)
    ASSERT (result = PONYC.RESULT.ALLOC.OK)
    -- Start infinite loop (therefore no shutdown of pony kernel later)
    WHILE TRUE
      -- Listen to requests from broker-handle
      broker.handle.svr[to.broker] ? CASE
        -- Register worker
        WORKERHANDLE! worker.handle:
        reg.worker; worker.handle
          ...  Store ‘worker.handle’ in database
        -- Get worker
        get.worker
          IF
            ...  Worker available
              WORKERHANDLE! worker.handle:
              SEQ
                ... Retrieve ‘worker.handle’ from database
                broker.handle.svr[from.broker] ! get.worker.confirm;
                                                 worker.handle
            TRUE
              broker.handle.svr[from.broker] ! no.worker.available
:
```

Algorithm 5.3: Sample application: The worker

```
#INCLUDE "decls.inc"
#INCLUDE "ponylib.inc"
#USE "pony.lib"

PROC worker (CHAN BYTE key?, scr!, err!)
  INT own.node.id, result:
  PONY.NETHANDLE! net.handle:
  SHARED BROKERHANDLE! broker.handle:
  WORKERHANDLE! worker.handle:
  WORKERHANDLE? worker.handle.svr:
  SEQ
    -- Start pony
    pony.startup.unh (PONYC.NETTYPE.TCPIP, "", "sample-app",
                      "", PONYC.NODETYPE.SLAVEWAIT,
                      own.node.id, net.handle, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    -- Allocate shared client-end of broker-handle
    pony.alloc.sc (net.handle, "broker-handle", PONYC.SHARETYPE.UNKNOWN,
                   broker.handle, result)
    ASSERT (result = PONYC.RESULT.ALLOC.OK)
    -- Allocate worker-handle
    worker.handle, worker.handle.svr := MOBILE WORKERHANDLE
    -- Start infinite loop (therefore no shutdown of pony kernel later)
    WHILE TRUE
      SEQ
        -- Register with broker
        CLAIM broker.handle
          broker.handle[to.broker] ! reg.worker; worker.handle
        -- Inner loop
        INITIAL BOOL running IS TRUE:
        WHILE running
          -- Listen to requests from worker-handle
          worker.handle.svr[to.worker] ? CASE
            ...  Stuff dealing with the service provided by the worker
              ...  Deal with it
            -- Finish transaction and get worker-handle back
            finish; worker.handle
              -- Exit inner loop
              running := FALSE
:
```

Algorithm 5.4: Sample application: The customer

```
... Compiler declarations
PROC customer (CHAN BYTE key?, scr!, err!)
  INT own.node.id, err.point, result:
  PONY.NETHANDLE! net.handle:
  PONY.ERRHANDLE! err.handle:
  PONY.MSGHANDLE! msg.handle:
  SEQ
    pony.startup.unh.ueh.iep.mh    -- Start pony
         (PONYC.MSGTYPE.STATUSERR, PONYC.NETTYPE.TCPIP, "", "sample-app",
          "", PONYC.NODETYPE.SLAVEWAIT, own.node.id, net.handle,
          err.handle, err.point, msg.handle, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    CHAN BOOL sync:
    PAR
      pony.msg.out.uo.ue (msg.handle, scr!, err!)    -- Message-outputter
      SHARED BROKERHANDLE! broker.handle:
      SEQ
        ... Allocate shared client-end of broker-handle
        IF
          result <> PONYC.RESULT.ALLOC.OK
            ... Deal with allocation error
          TRUE
            BOOL worker.available:
            WORKERHANDLE! worker.handle:
            SEQ
              CLAIM broker.handle    -- Get worker-handle from broker
                SEQ
                  broker.handle[to.broker] ! get.worker
                  broker.handle[from.broker] ? CASE
                    no.worker.available
                      worker.available := FALSE
                    get.worker.confirm; worker.handle
                      worker.available := TRUE
              IF
                worker.available
                  SEQ
                    ... Communicate over worker-handle regarding service
                    -- Finish transaction and return worker-handle
                    worker.handle[to.worker] ! finish; worker.handle
                TRUE
                  ... Deal with absence of workers
        pony.shutdown (net.handle)    -- Shut down pony kernel
        sync ! TRUE
      SEQ
        ... Wait for ‘sync’ signal or timeout and check for errors
        pony.err.shutdown (err.handle)    -- Shut down error-handler
:
```

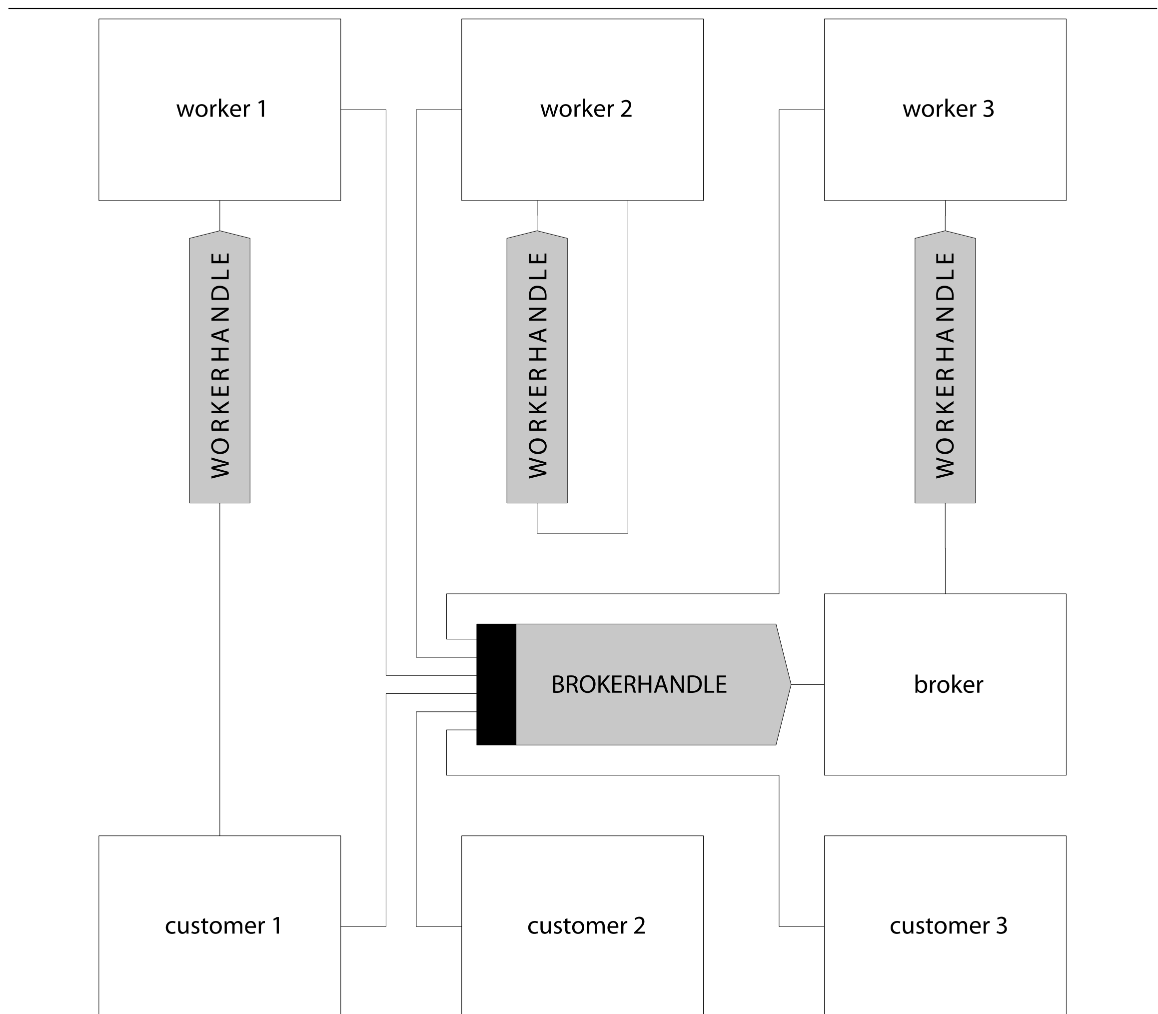

Figure 5.1: Sample application: Possible dynamic layout

Part II

Design and Implementation of pony

This part of the thesis presents the design and implementation of the pony environment. Chapter 6 gives an overview of the internal structure of the pony system. pony's protocol-conversion mechanism is examined in Chapter 7. Chapter 8 describes how networked channel-types are implemented in pony. In Chapter 9, the network-specific parts of pony are explained. This part concludes with Chapter 10, where the implementation of pony's error-handling and message-handling mechanisms, the Application Name Server and pony's startup and shutdown mechanisms are discussed.

Chapter 6

Structure of pony

The first part of this chapter introduces the internal components of the pony system and defines certain terms that are necessary in order to explain pony's functionality. The second part discusses the compiler support for pony-enabled KRoC builds, and the functionality of the main pony kernel.

6.1 NCTs and CTBs

There are two important terms related to pony which it is vital not to get confused: network-channel-types and channel-type-blocks. As already defined, a network-channel-type (NCT) is a channel-type that may connect several nodes. An NCT is a *logical* construct that comprises a networked channel-type across the entire pony application. Each NCT has a unique ID, and a unique name if it was allocated explicitly, across the application.

A *channel-type-block (CTB)* is the memory block of a channel-type on an individual node. This memory structure holds all information that is needed for the function of the channel-type. CTBs are located in the dynamic mobilespace of the node. All channel-type-end variables belonging to a certain channel-type are pointers to that channel-types's CTB. Details about the layout of a CTB can be found in [Bar03].

In the pony environment, we distinguish between *non-networked* and *networked* CTBs. A traditional (intra-processor) channel-type is made up of exactly one, non-networked, CTB. An NCT is made up of several, networked, CTBs, namely one CTB on each node where there are (or have been) ends of that NCT. The CTBs of an NCT are interconnected by the pony infrastructure. Non-networked CTBs can become networked by implicit allocation, cf. Section 3.1.2.

6.2 Internal Components of pony

Apart from some compiler support for CTBs in pony-enabled programs (discussed in detail in Section 6.3), pony is implemented entirely as an occam-π library. Most parts of this library were implemented in occam-π. Some auxiliary functions were implemented in C. The protocol-converters (see below) were implemented as CIF processes. Figure 6.1 shows the layout of the pony environment with its various components and the external and internal handles[1] used for communication between the individual components.

The figure assumes that the network-handle and the error-handle are unshared, the node is the master, and the network-type is TCP/IP. In order to keep the figure uncluttered, each component is just depicted once, even if it may occur several times within the pony environment. Unshared client-ends of internal handles are held by the process in which the end is located in the figure.[2] Shared client-ends of internal handles may be held by several component processes at the same time. If such an end extends into another process (the instant-handler in the CTB-handler or one of the managers), this means that the relevant process holds the end and will pass it to other components on request.

[1] Both the external and the internal handles are channel-types.

[2] This applies to the internal decode- and encode-handles, whose client-ends are held by the relevant CTB-handler.

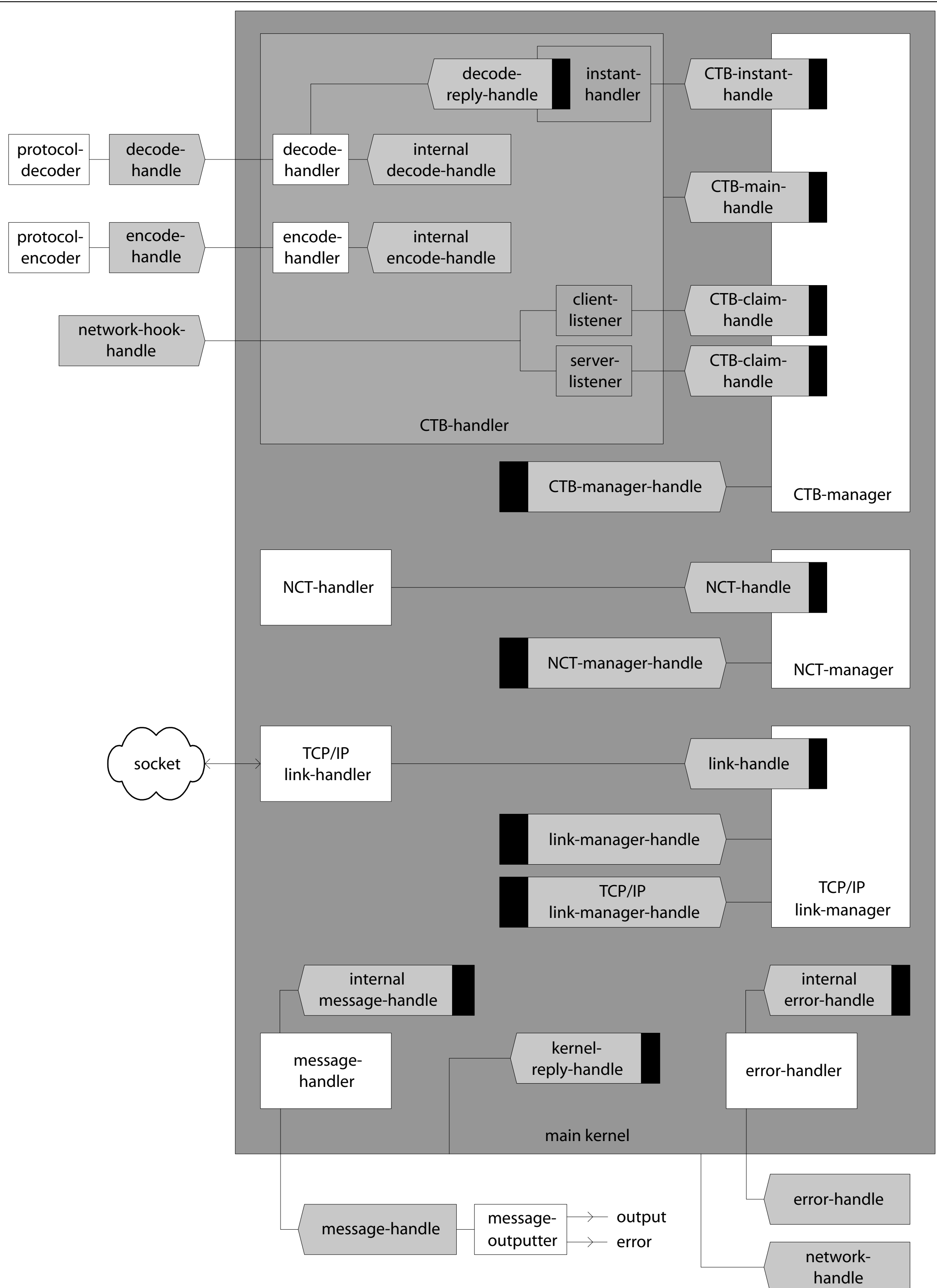

Figure 6.1: Layout of the pony environment

The communication between the individual components of the pony environment follows the principle of cycle-free client/server communication as set out in [MW96]. Although the communication structure between the individual components may change dynamically, it is guaranteed that at any given time, the client/server digraph is cycle-free; the communication is therefore deadlock-free.

6.2.1 The Individual Components

This section briefly introduces the individual components of the pony environment. A detailed description of their functionality, which includes the usage of the internal handles for communication between the components, will follow in the next chapters.

The Protocol-converters

The purpose of the *protocol-converters* is to enable the pony environment to support networked channels carrying all common occam-π protocols. For each networked channel (i.e. for each channel in a networked CTB), there is one set of protocol-converters, consisting of a *protocol-decoder* and a *protocol-encoder*.

On the sending node, the decoder decodes the incoming protocol into a special protocol that is used internally by the pony kernel. After something has been sent from one node to another via the pony environment, the encoder on the receiving node takes the intermediary pony protocol and encodes it back into the user-level protocol before passing it on to the receiving user-level process.[3] The protocol-converters are described in detail in Chapter 7.

[3]The terms ‘decode’ and ‘encode’ are to be seen from the point of view of the user-level protocol. This is just a convention — from the point of view of the intermediary pony protocol, the two terms would have to be exactly the other way round.

Decode-handler and Encode-handler

The *decode-handler* takes the data from the decoder and packs it into a suitable format for sending it over the network. On the receiving node, the *encode-handler* takes the packed data coming from the network, unpacks it, and passes it on to the encoder. Additionally, the decode-handler and the encode-handler deal with the implications arising from the movement of NCT-ends over networked channels and the implicit allocation of NCT-ends where applicable. Details are given in Section 7.8.

The CTB-handler

The *CTB-handler* deals with the function of a networked CTB. There is a CTB-handler for each networked CTB on the node. The CTB-handler handles incoming claim and release requests for the ends of the CTB, as well as the communication along its channels. Please note that the instant-handler, the client-listener and the server-listener in the CTB-handler (cf. Figure 6.1) are no actual components of pony but just simple sub-processes of the CTB-handler. What they do is explained in Section 8.1, which describes the functionality of the CTB-handler in detail.

The CTB-manager

The *CTB-manager* is responsible for starting new CTB-handlers when needed (during explicit allocation and when making a previously non-networked CTB networked). It also keeps the various internal handles for existing CTB-handlers and passes them to other pony components on request (via the *CTB-manager-handle*). The implementation of the CTB-manager is discussed in Section 8.2.

The NCT-handler

NCT-handlers only exist on master nodes. There is one NCT-handler for each NCT in the application. The NCT-handler is responsible for handling claim and release

requests coming from the CTB-handlers on the various nodes of the application. This involves queueing claim requests (if several nodes try to claim the same NCT-end) until they get served. Details can be found in Section 8.3.

The NCT-manager

The *NCT-manager* resides on the master node and starts new NCT-handlers when needed. This is the case when the first end of an NCT is allocated explicitly, or when a previously non-networked CTB is made networked on a node and a new NCT needs to be allocated implicitly. The NCT-manager keeps the *NCT-handles* for existing NCT-handlers and passes them (via the *NCT-manager-handle*) to requesting link-handlers[4]. Section 8.4 describes the implementation of the NCT-manager.

The link-handler

Link-handlers handle network links between two nodes of a pony application. On each node, there is a link-handler for each link that has been established to another node. The link-handler takes messages from pony's various components and passes them on to the remote node via the link. When the link-handler on the receiving node gets a network-message over its link, it passes it on to the component for which the message is intended. Section 9.1 discusses the implementation of the link-handler for TCP/IP.

The link-manager

The *link-manager* establishes new links (and starts new link-handlers) when necessary. For TCP/IP, this means that new socket connections to other nodes are established or incoming socket connections from other nodes are accepted. The link-manager keeps

[4] No other components will ever request an NCT-handle.

the *link-handles* for existing link-handlers and passes them (via the *link-manager-handle*) to requesting pony components. The implementation of the link-manager for TCP/IP is explained in Section 9.2.

All messages exchanged between two nodes are multiplexed over the link between the nodes. This applies especially to messages sent over networked channels. The multiplexing of possibly many networked channels over a single link was inspired by the Virtual Channel Processor of the T9000 transputer [Inm93], although the routing in pony is dynamic because NCT-ends may move to other nodes. pony's routing is a dynamic version of the 'crossbar' routing found in JCSP.net [WAF02].

Error-handler and Message-handler

The error-handler and the message-handler have already been introduced in Chapter 4. A detailed description of the implementation of pony's error-handling mechanism is given in Section 10.1. pony's message-handling mechanism is discussed in detail in Section 10.2.

6.2.2 Modular Design of pony

The structure of the pony environment is modular, which makes it easy to replace components when needed. The most obvious application for this feature would be adding support for new network-types to pony. This could easily be done by adding new network drivers (a link-handler and a link-manager), as well as a new ANS, for the new network-type. The other pony components would not need to be modified and could communicate with the new network drivers via the existing interface (the internal handles). During startup, the correct link-manager is started by the pony environment, depending on the network-type used.

The implementation of the startup and shutdown of the various components, as well as the ANS, are described in detail in Sections 10.3 through 10.5. To understand the details of the startup and shutdown of pony properly, it is necessary to understand

the implementation of pony's components first, therefore startup and shutdown are discussed in the last chapter of Part II.

6.3 CTBs in pony-enabled Programs

In order to accommodate the needs of NCTs, CTBs in a pony-enabled KRoC build contain more information than their 'traditional' counterparts. This applies to both networked and non-networked CTBs, since non-networked CTBs may implicitly become networked at any time. In traditional (non-pony-enabled) KRoC builds, the memory layout of CTBs remains as it used to be.[5] The changes in the memory layout and the handling of CTBs in pony-enabled programs were implemented in the KRoC compiler by Fred Barnes from the University of Kent. A traditional CTB contains the following:

- A reference-count that keeps track of how many ends of the channel-type are currently in scope. It is 2 at the point of allocation, but may increase (when shared ends get auto-cloned) or decrease (when an end goes out of scope) during the lifetime of the channel-type. When the reference-count reaches zero, the CTB is deallocated.
- The channel-words for the channels in the channel-type.
- Semaphores for the client-end and the server-end if the respective end is shared. The server-semaphore is located above the client-semaphore. The memory for the server-semaphore is not allocated if the server-end is unshared; the memory for the client-semaphore is always allocated unless *both* ends are unshared. The latter is to ensure that the server-semaphore is always located at the same memory offset.

[5] A possible future improvement to keep also in pony-enabled programs the traditional, smaller, memory layout for CTBs that will never become networked is outlined in Section 12.3.2.

Table 6.1 shows the layout of a CTB in a pony-enabled KRoC build. The order is bottom-up, i.e. the reference-count is located at the lowest address. The reference-count, the channel-words and the client-/server-semaphores are the same as for traditional CTBs. Please note that there are always both semaphores in the CTB (even if neither end is shared) because the state-field and the state-semaphore are stored on top of them.

Size	**Item**
1 word	reference-count
1 word	pointer to type-descriptor
1 word	network-hook pointer
n words	channel-words of the channels in the channel-type
2 words	client-semaphore
2 words	server-semaphore
1 word	state-field
2 words	state-semaphore

Table 6.1: Memory layout of CTBs in a pony-enabled KRoC build

6.3.1 The Pointer to the Type-descriptor

As discussed in Section 1.5.6, in pony-enabled KRoC builds, the compiler generates type-descriptors for all channel-types that have been declared. In the CTB of each channel-type, there is a pointer to the relevant type-descriptor. The type-descriptor describes the structure of the CTB and the protocols carried by the channels within the CTB. The information stored in the type-descriptor is used during the explicit allocation of NCT-ends, for the implicit allocation of an NCT when a non-networked CTB becomes networked, and when an end of an NCT arrives on a node for the first time and a new networked CTB needs to be allocated for it.

As already pointed out in Section 1.5.6, pony's allocation processes use a special '`MOBILE.CHAN`' parameter for the end that is to be allocated. Since the actual type of the end is unknown at compile-time, the type-descriptor, which is supplied to the allocation process as a second, hidden, parameter, is used to determine the structure

of the channel-type. This is necessary both to allocate the memory for the CTB and its internal structure, as well as for setting up the necessary infrastructure for the NCT in the pony environment. The explicit allocation mechanism is described in detail in Section 6.4.2.

When sending an end of a non-networked CTB over the network for the first time, the CTB must be made networked. Although no memory for the CTB needs to be allocated (since it already exists), the infrastructure for the new, implicitly allocated, NCT to which the CTB will belong still needs to be set up. In order to do this, pony's protocol-decoder (described in Section 7.5) needs to extract the necessary information about the structure of the channel-type from its type-descriptor. The mechanism for making a non-networked CTB networked when necessary is described in detail in Section 6.3.2.

The third case where type-descriptors are needed is when an end of an NCT arrives on a node for the first time. In this case, pony's protocol-encoder (described in Section 7.6) needs to allocate a new networked CTB and set up the infrastructure for the NCT on the node where the end arrives. This mechanism is similar to the mechanism used for explicit allocation by the allocation processes, and likewise needs to access the information stored in the type-descriptor.

When a CTB becomes (explicitly or implicitly) networked, a set of protocol-converters (one protocol-decoder and one protocol-encoder) are started for each channel of the CTB. Both the decoder and the encoder have access to the *protocol-descriptor* of the channel they serve. By 'protocol-descriptor' we mean the type-descriptor of a channel, which describes the protocol carried by the channel. It is part of the type-descriptor of the channel-type to which the channel belongs. For channels carrying channel-type-ends, the protocol-descriptor of the channel contains a pointer to the type-descriptor of the channel-type to which the carried end belongs.

To access a type-descriptor when a CTB is being made networked, the decoder uses the pointer stored in that CTB. When the encoder needs to access a type-descriptor in order to allocate the CTB for a newly arriving NCT-end, it uses the pointer stored in the protocol-descriptor of the channel it serves.[6]

6.3.2 The Claim/Release Mechanism

The network-hook, the state-field and the state-semaphore are necessary for the correct handling of claiming and releasing shared ends of networked CTBs, as well as for making a CTB networked during an implicit allocation. In traditional CTBs, the claiming or releasing of a shared end is implemented as a claim or release of the respective semaphore in the CTB. In networked CTBs, the compiler additionally needs to inform the pony kernel about the claim or release request. This is done via the *network-hook*.

For each networked CTB on a node, the relevant CTB-handler holds the server-end of a *network-hook-handle* which contains two channels: one for the client-end and one for the server-end of the CTB. When the respective end is claimed or released, a claim or release signal is sent to the CTB-handler via the correct channel in the network-hook-handle. The CTB-handler will read from the channel using an extended rendezvous, which will be released when the claim has been accepted, respectively, when the release operation has been completed. This is important especially for the claim — as soon as the handshake for the claim signal has been completed, KRoC knows that the end is now claimed in the pony application, i.e. our node is the only one allowed to use the end. Thus, KRoC may now access the CTB's channel-words. It is important to note the two-step approach here. When the end of a networked CTB is claimed, first the client- or server-semaphore is claimed as usual. Once that

[6]The decoder might as well use the type-descriptor pointer stored in the protocol-descriptor of the channel it serves, since it is identical to the pointer stored in the CTB that is to be made networked. Using the pointer in the CTB simplifies the implementation, however.

claim has been accepted, the claim signal is sent over the network-hook. For a release, first the network-hook signal is sent, then the client- or server-semaphore is released.

The network-hook pointer in a networked CTB 'hooks' into the network-hook-handle, which means that it points to the base of its channel-words. Using this mechanism, the network-hook pointer in the CTB replaces the client-end of the network-hook. During the lifetime of a networked CTB, the reference-count of its network-hook-handle is therefore 1, which means that the network-hook-handle will automatically get deallocated when the pony kernel is shut down and the server-end of the network-hook-handle goes out of scope. Details are given in Section 6.3.3.

For non-networked CTBs in a pony-enabled program, the network-hook pointer is null. To find out whether a pony-enabled CTB is networked or not, its network-hook pointer is checked for nullness.

There are two purposes for the *state-field* in a pony-enabled CTB — one for non-networked and one for networked CTBs. While the CTB is non-networked, its state-field contains the state of the CTB's ends. More precisely, it stores for both the client-end and the server-end whether they are unshared or shared, and, if an end is shared, whether it is currently claimed or not. This information is necessary when an end of a non-networked CTB is sent to another node for the first time and the CTB becomes networked in the process.

For each NCT in a pony application, the relevant NCT-handler on the master node keeps track, for both the client-end and the server-end, of whether they are currently claimed, and if so, on which node they are claimed. When a claim or release signal for the end of a CTB comes in via the network-hook, the CTB-handler notifies the NCT-handler on the master node, which then updates the current state of the respective NCT. This involves queueing claim requests (if several nodes try to claim the same end of the NCT) until they get served. In order for this mechanism to work properly, it is vital that the NCT-handler always clearly knows which nodes have requested claims, and which nodes currently hold a claim, for all NCT-ends across the application.

When an NCT gets allocated explicitly, it always starts with both of its ends unclaimed. For implicit allocation, this is not necessarily the case. As a reminder: implicit allocation happens when an end of a currently non-networked CTB is sent to another node for the first time. At the point in time when this happens, either or both ends of the CTB may be claimed. This applies to the opposite end, but also to the end that is currently being sent. In the latter case, the end variable that is to be sent is an auto-clone of the claimed variable (cf. Section 1.5.3). When a non-networked CTB becomes networked, pony uses the state-field to find out about the current unshared/shared claimed/released state of the CTB's ends so that the (implicitly allocated) NCT can start its life in a clearly defined state.

Once a CTB is networked, it will stay networked for the rest of its lifetime. Hence keeping track of the unshared/shared claimed/released state in the state-field is only relevant for non-networked CTBs. For networked CTBs, this has no significance anymore, since once a CTB becomes networked (which for CTBs belonging to an explicitly allocated NCT is right from the start), the pony environment will keep track of whether an end is claimed or not by using the network-hook mechanism.

For networked CTBs, the state-field instead stores the ID of the NCT to which the CTB belongs. This information is needed when ends of that CTB are being sent over the network. During an implicit allocation, what is stored in the state-field is 'swapped'. As mentioned before, this happens when a non-networked CTB becomes networked, i.e. when one of its ends is sent over the network for the first time. For explicitly allocated NCTs, their CTBs start their life being networked straight away, therefore their state-field stores the NCT-ID right from the start.

Algorithm 6.1 shows the compiler mechanism for claiming and releasing ends of a CTB in a pony-enabled program. The *state-semaphore* is used to protect the state-field. This is necessary in order to prevent race hazards between the update of the state-field for non-networked CTBs and the readout of the state-field during an implicit allocation. Every time an end of a CTB is being sent over a networked

channel[7], the protocol-decoder serving that networked channel checks whether the CTB that is sent over the networked channel is already networked or not. In the latter case, the CTB is made networked and a new NCT for the CTB is allocated implicitly. This mechanism is shown in Algorithm 6.2.[8]

Algorithm 6.1: Claiming/releasing ends of a pony-enabled CTB

```
SEQ
  ...  In case of a claim: claim client- resp. server-semaphore now
  ...  Claim state-semaphore
  IF
    ...  CTB is non-networked (i.e. network-hook is null)
      SEQ
        ...  Update the state-field (depending on whether we are
        ...    claiming/releasing a client/server)
        ...  Release state-semaphore
    TRUE  -- i.e. CTB is networked (network-hook <> null)
      SEQ
        ...  Release state-semaphore
        ...  Send relevant claim or release signal over the client
        ...    or server channel of the network-hook
        ...  Don't change the state-field (it contains the NCT-ID)
  ...  In case of a release: release client- resp. server-semaphore now
```

An important characteristic in Algorithm 6.1 is that for non-networked CTBs, the state-semaphore is released *after* the state-field has been updated, whereas for networked CTBs, the state-semaphore is released *before* the claim/release signal is sent over the relevant network-hook channel. The latter is vital in order to avoid race conditions between a client-claim and a server-claim, or between a claim and the check in Algorithm 6.2. As pointed out above, the handshake for the claim signal over the network-hook is not released until the claim has been accepted. Therefore, it is vital that the state-semaphore is released *before* a claim signal is sent over the network-hook, because otherwise a pending claim might block the possibility to claim the other end of the CTB, or to send an end of the CTB over a networked channel.

[7]In pony, all networked channels are part of an NCT.

[8]These algorithms are in a pseudo-occam notation and not actually implemented in occam-π. Algorithm 6.1 is implemented directly in the compiler; Algorithm 6.2 is part of the protocol-decoder, which is implemented as a CIF process.

Algorithm 6.2: Making a CTB networked when necessary

```
SEQ
  ...  Claim state-semaphore
  IF
    ...  CTB is non-networked (i.e. network-hook is null)
      SEQ
        ...  Check state of client and server in state-field
        ...  Notify pony kernel about client- and server-state and
        ...    everything necessary for an implicit allocation
        ...  Wait for pony kernel to implicitly allocate a new NCT
        ...    for the CTB
        ...  Get NCT-ID, client-end of network-hook-handle and
        ...    handles for the protocol-converters from pony kernel
        ...  Store NCT-ID in state-field
        ...  Set up network-hook pointer with channel-words in
        ...    network-hook-handle
        ...  Decrease reference-count of network-hook-handle
        ...  Fork off protocol-converters for all channels in the CTB
    TRUE  -- i.e. CTB is networked (network-hook <> null)
      SKIP
  ...  Release state-semaphore
```

6.3.3 Shutting Down pony-enabled CTBs

Contrary to non-networked CTBs, when the reference-count of a networked CTB reaches zero, the CTB's memory is *not* freed. Algorithm 6.3 shows the modified reference-count check that is made by the compiler after the reference-count of a CTB has been decreased.

Algorithm 6.3: Reference-count check for a pony-enabled CTB

```
SEQ
  ...  The CTB's reference-count has just been decreased
  IF
    ...  (Reference-count = 0) AND
    ...      (CTB is non-networked (i.e. network-hook is null))
      ...  Deallocate CTB
    TRUE
      SKIP
```

The reason for keeping networked CTBs with a reference-count of zero in place is that when a new end of the NCT to which the CTB belongs is allocated explicitly,

or when another end of that NCT arrives on our node, we can re-use that CTB as well as its infrastructure (including the protocol-converters).

When a new end of an NCT comes into existence (by explicit allocation or by arriving over a networked channel) on a node where there already is a networked CTB for that NCT, no allocation needs to be done. In such a case, simply the reference-count of the CTB is increased. This also applies to CTBs whose reference-count is currently zero (for instance because all ends of the NCT to which the CTB belongs have left our node earlier). When a new end for such a CTB comes into existence on our node, pony (more specifically, the allocation process resp. the encoder) increases the reference-count of the CTB — to 1 in this case.

The deallocation of networked CTBs is done as part of the pony shutdown process. As discussed in Section 3.2, the shutdown process may only be called after all usage of networked (or possibly networked) channel-type-end variables has finished. Therefore, it is safe now to deallocate networked CTBs, because neither the user-level code nor the pony kernel will access them anymore.

The shutdown process tells the pony kernel to shut down, which in turn will shut down all its components. After this has happened, the pony kernel returns an array that contains the pointers to all networked CTBs on the node. At this point, the reference-count of those CTBs matches exactly the number of ends of the particular CTB that are in scope within the user-level code (as it would be for non-pony-enabled occam-π programs).

If we consider the typical structure of a pony node shown in Algorithm 3.1, it emerges that if all possibly networked channel-type-ends are declared right before (or inside) the 'FORKING' block, obviously, all their reference-counts would be zero now. However, possibly networked channel-type-end variables may have been declared elsewhere by the user-level code — even right at the start of the program for instance. In that case, some of the networked CTBs would have reference-counts greater than zero now.

The pony shutdown process now calls an auxiliary C function[9], which is given the array of pointers to networked CTBs that was returned by the pony kernel. The auxiliary function performs Algorithm 6.4 for each networked CTB.

Algorithm 6.4: Shutdown of a networked CTB

```
IF
  ...  Reference-count = 0
    ...  Deallocate CTB
  TRUE
    SEQ
      ...  Set network-hook pointer in CTB to null
      ...    (thus making it look non-networked)
      ...  Don't deallocate the CTB
```

After the pony shutdown process has finished, all networked CTBs with a reference-count of zero have been deallocated. The formerly networked CTBs which still have ends in scope, and whose reference-count is still greater than zero, will be deallocated automatically by the normal compiler mechanism (see Algorithm 6.3) as soon as their last end goes out of scope.

Please note that in Algorithm 6.4 we use the term 'making it *look* non-networked' rather than 'making it non-networked' because the only point in setting the network-hook pointer to null is to 'fool' the reference-count check in Algorithm 6.3, so that it will deallocate the CTB when the reference-count reaches zero. The CTB is *not* a 'normal' non-networked CTB, since the state-field still contains the NCT-ID and not the unshared/shared claimed/released state of its ends as for 'normal' non-networked CTBs. This is not a problem, however, since, as mentioned before, the ends of those formerly networked CTBs are not allowed to be used anymore anyway, hence it is irrelevant what is stored in the state-field. The only thing that will still happen to those CTBs is their deallocation when their last end goes out of scope.

[9] This function, as well as the other auxiliary C functions mentioned in this thesis, have largely been implemented by Fred Barnes from the University of Kent.

6.4 The Main pony Kernel

6.4.1 Layout and Startup

As discussed earlier, user-level processes communicate with the pony kernel via the network-handle, the error-handle and the message-handle if applicable. While the server-ends of the latter two are held by the error-handler and the message-handler respectively, the server-end of the network-handle is held directly by the main pony kernel.

During the startup of the pony environment (cf. Section 10.4), the main kernel starts off its various subcomponents, namely the error-handler, the message-handler and the various managers as the case may be. Then it waits for requests via the network-handle. These may be requests for the explicit allocation of NCT-ends, or the request to shut down the pony environment. The implementation of the explicit allocation is discussed in Section 6.4.2, the shutdown mechanism in Section 10.5.

6.4.2 Explicit Allocation of NCT-ends

When a user-level process wants to allocate a new NCT-end, it needs to call one of pony's allocation processes (cf. Section 3.1.1). The first thing the allocation process does is checking whether the NCT-name is an empty string. If this is the case, the allocation process terminates returning an error. Otherwise, it calls an auxiliary C function that extracts the following information from the type-descriptor of the channel-type (which the auxiliary function can access via the hidden parameter discussed in Section 1.5.6):

- The number of channels in the channel-type.
- The number of reading-ends in the channel-type from the point of view of its server-end — henceforth called 'readers-in-server'.

- The *type-hash* of the channel. This is calculated by the compiler for each channel-type that is declared on a node. The type-hash is de-facto unique for each channel-type, since although duplicate type-hashes are theoretically possible, chances for them to occur are vanishingly small.

Now the allocation process requests the explicit allocation from the main pony kernel via the network-handle. Doing this, it gives the following information to the main kernel:

- The *direction-type* of the end that is to be allocated, i.e. whether the end is a client-end or a server-end.
- The share-type of the end that is to be allocated. This may be 'unshared' or 'shared'.
- The share-type of the opposite end. As discussed in Section 3.1.1, this may be 'unknown', 'unshared' or 'shared'.
- The number of channels.
- The number of readers-in-server.
- The type-hash of the channel-type.
- The NCT-name under which the end is to be allocated.

The main kernel now forwards all this information, except the number of channels and the number of readers-in-server, to the NCT-manager on the master node. This is done by first passing the message (via the corresponding link-handle) to the link-handler that holds the link to the master — henceforth called 'master-link-handler' — which then passes it on to the master node via its link.

Then the main kernel waits for a reply from the NCT-manager. This will come via the link from the master node. When getting the reply, the master-link-handler passes it on to the main kernel via the *kernel-reply-handle*. The reply may either be

an error or a confirmation. An error would be returned to the allocation process, which would then return it to the user-level process through its result parameter.

If a confirmation arrives from the NCT-manager, it comes together with the ID of the NCT to which the NCT-name belongs. For the NCT-manager, this may have been a new or an existing NCT, depending on whether the given NCT-name was used before to allocate an NCT-end somewhere across the pony application.[10] If the NCT is new, the NCT-manager has forked off a new NCT-handler for it. For the main kernel, it is irrelevant whether the NCT is new or not; the only thing that matters is that the NCT-manager has returned the NCT's ID.

The main kernel now requests the allocation of a new channel-type-end for that NCT-ID from the CTB-manager. This is done via the CTB-manager-handle. There are now two possible replies from the CTB-manager. If there already is a CTB for the NCT with the given ID on our node, the CTB-manager will tell the main kernel that no new CTB needs to be allocated but just the reference-count of the existing CTB must be increased. Together with this message, the CTB-manager gives the main kernel the pointer to the CTB — henceforth called 'CTB-pointer'.[11] The main kernel then passes this information on to the allocation process, which in turn calls an auxiliary C function that increases the CTB's reference-count and returns a channel-type-end pointing to the CTB. This channel-type-end is then returned to the user-level process by the allocation process.

If there is no CTB for the NCT with the given ID on our node yet, the CTB-manager tells the main kernel to allocate a new CTB. The main kernel then notifies the CTB-manager about the number of channels and the number of readers-in-server in the CTB that is to be allocated. The CTB-manager now allocates a set of handles and returns the following (client-ends of the) handles to the main kernel:

[10] The NCT-manager uses the type-hash to ensure that only NCT-ends of the same type are allocated under the same NCT-name.

[11] In the occam-π context, the CTB-pointer is not the actual pointer, but its integer value.

- A network-hook-handle.
- An array of *decode-handles* — henceforth called ‘decode-handle-array’.
- An array of *encode-handles* — henceforth called ‘encode-handle-array’.

The handles in the two arrays will be used by the protocol-converters to communicate with the decode-handlers and encode-handlers. The size of the arrays equals the number of channels in the CTB. The main kernel now tells the allocation process that a new CTB must be allocated, and passes the NCT-ID, the network-hook-handle and the two arrays to the allocation process. The allocation process then calls an auxiliary C function that does the following:

- Allocate a new CTB according to the type-descriptor.
- Initialise the reference-count to 1.
- Store the type-descriptor pointer in the CTB.
- Initialise the client-, server- and state-semaphores.
- Store the NCT-ID in the state-field.
- Set up the network-hook pointer with the channel-words in the network-hook-handle.
- Decrease the reference-count of the network-hook-handle.
- Fork off a protocol-decoder and a protocol-encoder for each channel-word in the CTB.

For each channel-word in the CTB, *both* a decoder and an encoder must be started. This is necessary because every channel may be used in both directions — depending on whether the client-end or the server-end of the NCT are claimed on our node (which may change dynamically). This, obviously, applies to both readers-in-server and writers-in-server (the latter being channels that have their writing-end in

the server-end of the channel-type, accordingly). The protocol-converters are given the channel-word, the protocol-descriptor of the channel, and the correct decode- or encode-handle from the arrays as parameters.

The CTB-handler, which will be holding the server-ends of the decode- and encode-handles, needs to know which of them are connected to protocol-converters serving readers-in-server, and which of them are connected to protocol-converters serving writers-in-server. This is necessary so that the channels in the CTB can always be used in the correct direction, both when the client-end and when the server-end of the NCT are claimed on our node. Therefore, the order of the handles in the decode- and encode-handle-arrays is vital.

The rule for the order of the handles is that both arrays contain first the handles for the readers-in-server, then the handles for the writers-in-server, and that the same index in the arrays refers to the handles for the same channel-word. The auxiliary function must obey this rule when forking off the protocol-converters for the channels in the CTB. Please note that the actual order of the channels *in the CTB itself* is irrelevant for the CTB-handler. The CTB-handler only needs to know how many readers-in-server there are in the CTB in order to distinguish between the handles for readers-in-server (at the beginning of the arrays) and the handles for writers-in-server (at the end of the arrays).

With this rule, the CTB-handler knows that if the client-end of the NCT is claimed on our node, the handles at the beginning of the arrays are used by protocol-converters serving channels to which the user-level-process writes, and the handles at the end of the arrays are used by protocol-converters serving channels from which the user-level-process reads. If the server-end of the NCT is claimed on our node, it is exactly the other way round.

The auxiliary function returns the newly allocated channel-type-end and its CTB-pointer to the allocation process. The allocation process then passes the CTB-pointer on to the main pony kernel and terminates, returning the newly allocated channel-type-end to the user-level process. The main kernel now passes the CTB-pointer

on to the CTB-manager, which stores it and forks off a new CTB-handler for the newly allocated CTB. The CTB-handler for its part forks off a decode-handler and an encode-handler for each channel in the CTB.

CTB-handlers do not distinguish between unshared and shared ends. They always treat both ends of the CTB as if they were shared. Unshared ends are claimed and released by the pony environment *internally* if applicable. They are claimed when they are allocated explicitly or when they arrive on our node via a networked channel; they are released when they are sent to a remote node over a networked channel.

Internal claim and release messages are sent via a *CTB-claim-handle*. For each CTB-handler, there are two CTB-claim-handles — one for the client-end and one for the server-end. If the newly allocated NCT-end was unshared (no matter whether a new CTB was allocated or an existing CTB's reference-count was increased), the main kernel now requests the CTB-claim-handle for the newly allocated end from the CTB-manager, and uses it to send a claim signal to the end's CTB-handler. In this way, the newly allocated end gets internally claimed.

Chapter 7

Protocol-conversion

Figure 7.1 shows the pony components related to protocol-conversion that are discussed in this chapter.

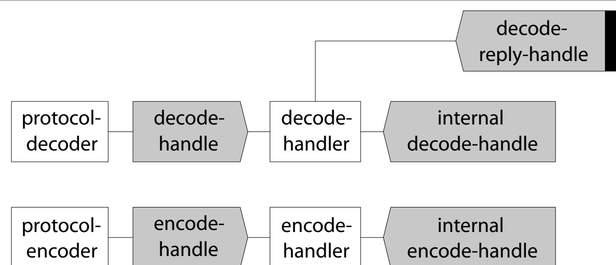

Figure 7.1: pony components related to protocol-conversion

7.1 The Protocol-converters

The protocol-converters are the interface between the user-level process and the pony environment (or, more specifically, the decode-handler and the encode-handler discussed in Section 7.8). As introduced above, they are used with networked channels to convert any given user-level protocol into a special protocol understood by the pony kernel, and back.

pony's protocol-converters were implemented in C as CIF processes by Adam Sampson from the University of Kent. They are an extended version of the generic

'DECODE.CHANNEL' and 'ENCODE.CHANNEL' processes presented in [SBW03]. They are more complex, however, since they need to cope with channel-type-ends as well, rather than just with data-items as the old generic versions.

One reason for implementing the protocol-converters as CIF processes was that they need to be able to fork off other protocol-converters — something not possible for a generic process created directly by the compiler. Additionally, implementing them in C allows a greater flexibility than the generic implementation did. Implementing the protocol-converters in occam-π itself was not an option because the low-level manipulations of channels and CTBs needed cannot be done in occam-π.

For each channel-word in a networked CTB, there are *both* a protocol-decoder and a protocol-encoder. These are forked off when the CTB becomes networked. This happens either during explicit allocation (cf. Section 6.4.2) or inside other protocol-converters. The latter can happen either when an end of a previously non-networked CTB is sent over a networked channel and the CTB becomes networked in the process (this is handled by the decoder), or when an end of an NCT arrives on a node where there is no CTB for this NCT yet (this is handled by the encoder).

7.2 Levels of Communication

The term 'communication' generally refers to an occam-π communication (using the '!' and '?' notations in the source code), but is too general for the purpose of describing the functionality of the protocol-converters, and the pony environment in general. Therefore, we distinguish between three different levels of communication.

A *user-level communication (ULC)* is the entire communication carried by an occam-π channel, i.e. everything noted after the '!' or the '?'. So, for instance:

```
c ! x; y; z
```

would be one ULC. A ULC therefore comprises the entire user-level protocol as described in the protocol-descriptor for the channel (which is part of the type-descriptor of the channel-type to which the channel belongs).

A *compiler-level communication (CLC)* is an *actual* communication on a channel-word. For instance, a ULC carrying a sequential protocol consists of several CLCs.

A *network-level communication (NLC)* is a communication between the protocol-decoder and the decode-handler, or between the encode-handler and the protocol-encoder. It is important to note that an NLC is a theoretical construct describing an atomic unit of information exchange between the protocol-converters and the rest of the pony environment. An NLC is *not* measuring the actual number of occam-π communications between the protocol-converters and the pony kernel, since one NLC may be implemented as several occam-π communications between decoder and decode-handler or encode-handler and encoder respectively.

We distinguish between NLCs for data-items and NLCs for channel-type-ends. Data-item NLCs carry an address/size pair, as in the earlier '`DECODE.CHANNEL`' and '`ENCODE.CHANNEL`' [SBW03]. These are the address and size (in bytes) of a piece of data that is to be sent from one node to another. Typically, this would be the address and size of the data-item itself, although some data-items may require a more complex implementation; see below. For most purposes, a CLC can be decoded as one NLC. Some data-items require more than one NLC per CLC; other cases allow the decoding of more than one CLC in a single NLC. Details about the implementation of the decoding and encoding of occam-π's various data-item protocols are given in Section 7.7.1. Channel-type-end NLCs are implemented as special sequences of communications between the protocol-converters and the pony kernel; see Section 7.7.2 for details.

7.3 Implementation of the Protocol-converters

When a user-level process sends something along a networked channel, the decoder takes the first CLC of the ULC as an extended input, i.e. the user-level process is not released until the decoder specifically releases it. The decoder can now calculate how many data-item NLCs and how many channel-type-end NLCs there are for the first CLC.[1] The convention for counting the NLCs is that for data-item NLCs, each address/size pair is counted as one NLC. For channel-type-end NLCs, the sequence of communications between the protocol-converters and the pony kernel that deals with *one* channel-type-end is counted as *one* NLC, i.e. each channel-type-end is equivalent to one channel-type-end NLC.

The decoder passes the number of NLCs in the first CLC, together with a Boolean flag telling whether there are any 'remaining' CLCs in the ULC (i.e. CLCs that will follow the first one), to the pony kernel, followed by the NLCs of the first CLC themselves. The pony kernel sends the first CLC, together with the information just mentioned, to the receiving node, where it gets passed to the protocol-encoder. If the remote protocol-encoder is able to output the first CLC of the ULC to the remote user-level process, an acknowledgement is sent back to the sending node. In this case, the procedure can be repeated for the 'remaining' CLCs of the ULC if there are any. Since the remote user-level process is now committed to take the entire ULC, the pony kernel can send the 'remaining' CLCs to the receiving node in one go. Again, the 'remaining' CLCs are preceded by the number of data-item NLCs and the number of channel-type-end NLCs in them.[2] When they have been taken by the user-level process on the receiving node, this again gets acknowledged. After having received this acknowledgement (or the acknowledgement for the first CLC in case there are

[1]Currently, no protocols are implemented in the protocol-converters where a single CLC would need to be decoded as a mixture of data-item NLCs and channel-type-end NLCs, or even as more than one channel-type-end NLC. Nevertheless, the pony kernel is able to cope with such protocols for future uses.

[2]The 'remaining' CLCs may obviously contain mixtures of NLCs, or several channel-type-end NLCs, since there may be more than one 'remaining' CLC (for instance in a sequential protocol).

no 'remaining' CLCs), the decoder knows that the ULC has been taken completely by the remote user-level process, and can therefore now release the sending user-level process from the extended rendezvous.

7.4 Cancelling Started ULCs

There are cases where the first CLC of a ULC will not be taken by the user-level process on the remote node. This may happen if the remote NCT-end gets released on the remote node (in case it is shared), or if it gets moved over a networked channel to some other node (in case it is unshared). In the latter case, the remote NCT-end would get internally released by the pony environment, as discussed earlier.

In cases like these, started ULCs must be *cancelled*. There are two possibilities now. Either the first CLC of the started ULC has already reached the encoder on the receiving node, or not. For the decoder on the sending node, this does not matter. It would in any case receive a 'cancel' message rather than the acknowledgement for the first CLC. When this happens, the decoder must cancel the started ULC, which means that it must leave the outputting user-level process in the same state as it was before the ULC started. This specifically means that the extended input for the first CLC, which the decoder had started, is not released, and thus the outputting user-level process remains suspended.

After the cancel, the decoder gets deactivated until the opposite end of the NCT is available again — when it is claimed again somewhere in case it is shared, or when it arrives on the target node (and gets internally claimed there) in case it is unshared. Then the decoder gets activated again and restarts with the extended input of the first CLC. The essential property of the cancel operation is that when the decoder gets activated the next time, it must be able to access the channel-word and perform the decoding of the first CLC as if the previous attempt (which has been cancelled) had never happened in the first place.

The second possibility is that the first CLC of the started ULC has reached the encoder on the receiving node already when the 'cancel-triggering' event happens. In this case, the encoder is now attempting to output the first CLC to the user-level process — which will not take it. If this happens, the decoder that is connected to the same channel-word as the outputting encoder (i.e. *not* the decoder on the sending node, but the one on the *receiving* node) will get a 'cancel-encode' message from the pony kernel.

This decoder then inputs the first CLC that the matching encoder tries to output. The decoder must undo all operations that the encoder performed before outputting the CLC. This especially involves deallocating any previously allocated memory (for dynamic mobiles) and decreasing any previously increased reference-counts (for channel-type-ends). Then, the decoder releases the encoder from the communication. Semantically, the 'cancel-encode' operation behaves like a user-level process that reads the first CLC of the ULC from the networked channel (i.e. from the encoder), but then every variable it has just read goes out of scope.

After the 'cancel-encode', everything must be in the same state as if the first CLC of the ULC had never reached the encoder in the first place. If the cancelled ULC consists of more than one CLC, the pony kernel will then notify the encoder that the ULC was cancelled, so that the encoder will not wait for the 'remaining' CLCs of the ULC anymore. If the ULC has just one CLC, it is irrelevant for the encoder whether the ULC was cancelled or not, because the first CLC of the ULC is output by the encoder in any case — either to the matching decoder if the ULC is cancelled, or the user-level process otherwise.

7.5 The Protocol-decoder

As mentioned above, the protocol-decoder may be *activated* or *deactivated*. It is activated when the NCT-end which holds the writing-end of the channel served by the decoder is claimed (externally or internally) on our node, and the opposite end is

claimed on another node. At all other times, the decoder is deactivated. As discussed above, the deactivation of the decoder may happen in the middle of a started ULC, in which case the ULC is cancelled. The decoder communicates over the decode-handle with its decode-handler, which for its part communicates over the *internal decode-handle* with the CTB-handler that is responsible for the CTB which holds the channel that is served by the decoder.

When the decoder starts its main loop, it waits for requests from the decode-handler. This may be an activation, a 'cancel-encode' request, or a shutdown signal. The 'cancel-encode' operation was discussed in Section 7.4 already; pony's shutdown mechanism will be explained in Section 10.5.

When a decoder gets activated by the decode-handler, it ALTs over the user-level channel and the incoming channel of the decode-handle. From the decode-handler, it may get a 'cancel' message, in which case the decoder is deactivated and returns to the main loop, after having sent a cancel confirmation back to the decode-handler. From the user-level channel, the decoder may get the first CLC of a new ULC. This would be taken by the decoder as an extended input.

After getting the first CLC from the user-level process, the decoder tries to output the number of NLCs in the first CLC to the decode-handler.[3] In parallel to that, it listens to the incoming channel of the decode-handle for a possible 'cancel' message from the decode-handler, as shown in Algorithm 7.1. This mechanism[4] is used instead of an output guard, which does not exist in occam-π and CIF. The counterpart mechanism in the decode-handler is shown in Algorithm 7.2.

Using this mechanism, the decode-handler may send a 'cancel' message to the decoder at any time before it has received the number of NLCs in the first CLC (which starts the ULC) from the decoder. It will definitely get a cancel confirmation, but may get a started ULC beforehand — which it must ignore.

[3]The number of NLCs in the first CLC is always accompanied by a Boolean flag telling whether there are any 'remaining' CLCs in the ULC (cf. Section 7.3). For simplicity, this Boolean flag will not be mentioned separately below.

[4]Implemented in C like the rest of the decoder; the notation in the algorithm is pseudo-occam.

Algorithm 7.1: Pseudo output guard mechanism used in the protocol-decoder

```
BOOL was.cancelled:
SEQ
  CHAN BOOL cancel.sync:
  PAR
    SEQ
      ...  Send number of NLCs in first CLC to decode-handler
      cancel.sync ! TRUE
    ALT
      ...  Get ‘cancel’ message from decode-handler
        SEQ
          BOOL any:
          cancel.sync ? any
          was.cancelled := TRUE
      BOOL any:
      cancel.sync ? any
        was.cancelled := FALSE
  IF
    was.cancelled
      SEQ
        ...  Cancel the started ULC and deactivate decoder
        ...  Send cancel confirmation to decode-handler
    TRUE
      ...  Continue with ULC normally
```

If no ‘cancel’ message comes in from the decode-handler during the mechanism in Algorithm 7.1, the decoder will now output all NLCs of the first CLC to the decode-handler. Afterwards, the decoder waits for a reply from the decode-handler. This may be an acknowledgement for the first CLC, or a ‘cancel’ message. Again, if it gets a ‘cancel’ message, the decoder would perform the cancel operation and send a cancel confirmation back to the decode-handler. In this case, however, the decode-handler can be sure that only a cancel confirmation can come from the decoder, therefore no complex mechanism as in Algorithm 7.2 is needed in the decode-handler here.

If the first CLC was acknowledged, the decoder can now do the necessary cleanup. This involves deallocating the memory of dynamic mobiles that were sent away, and decreasing the reference-count of the CTBs for all channel-type-end NLCs in the first CLC. If there are ‘remaining’ CLCs in the ULC, the decoder now reads them from

Algorithm 7.2: Counterpart to pseudo output guard in the decode-handler

```
SEQ
  ...  We need to cancel a started ULC
  ...  Send ‘cancel’ message to decoder
  ...  Get CASE input from decoder
    ...  Cancel confirmation
      SKIP
    ...  Number of NLCs in first CLC  -- ignore!
      ...  Get ‘real’ cancel confirmation from decoder
```

the user-level channel, again with an extended input (for the very last CLC). After this is done, the decoder sends the number of NLCs in the ‘remaining’ CLCs to the decode-handler, followed by the NLCs themselves. They will definitely be taken, since the remote user-level process is now committed to take the entire ULC. The decoder now waits for the acknowledgement from the decode-handler; then it does the cleanup after the ‘remaining’ CLCs. After having done this cleanup (or after the cleanup for the first CLC in case there are no ‘remaining’ CLCs), the decoder releases the user-level process from the extended rendezvous and returns to the main loop.

7.6 The Protocol-encoder

The protocol-encoder is significantly simpler than the decoder for two main reasons. Firstly, the encoder does not need to do any cancelling itself — started ULCs that the encoder tries to output are cancelled by the matching decoder with the 'cancel-encode' operation. Secondly, the encoder does not need to `ALT` between messages from the encode-handler and the user-level process — since it *outputs* to the user-level channel. Therefore, we do not need to distinguish between activated and deactivated states for the encoder.

The encoder communicates over the encode-handle with its encode-handler, which for its part communicates over the *internal encode-handle* with the CTB-handler that is responsible for the CTB which holds the channel that is served by the encoder. When the encoder starts its main loop, it waits for a request from the encode-handler. This may be either a shutdown signal (cf. Section 10.5 for pony's shutdown mechanism) or the number of NLCs in the first CLC of a new ULC. In the latter case, the encoder would input all NLCs of the first CLC from the encode-handler and output the first CLC to the user-level channel. Once the first CLC has been taken, the encoder sends an acknowledgement to the encode-handler.

If there are 'remaining' CLCs in the ULC, the encoder then waits for the next message from the encode-handler. If this is a 'cancel' message, the encoder now knows that the first ULC was not taken by the user-level process, but by the matching decoder performing a 'cancel-encode'. In this case, the encoder does not need to wait for the 'remaining' CLCs and may return to the main loop. The other possibility is that the encoder gets the number of NLCs in the 'remaining' CLCs from the encode-handler. In this case, the encoder then inputs all NLCs of the 'remaining' CLCs from the encode-handler and outputs all 'remaining' CLCs to the user-level channel. Once they have been taken, the encoder sends an acknowledgement to the encode-handler. After this is done (or after having sent the acknowledgement for the first CLC in case there are no 'remaining' CLCs), the encoder returns to the main loop.

Algorithm 7.3: Implementation of 'cancel-encode' operation in the decoder

```
SEQ
  ...  We just got a 'cancel-encode' request
  ALT
    ...  Input first CLC that the matching encoder is sending
      SEQ
        ...  Cancel that CLC
        ...  Release matching encoder from communication
        -- Wait for acknowledgement that CLC was output by encoder
        dec.handle[to.decoder] ? CASE cancel.encode.ack
        -- Return confirmation
        dec.handle[from.decoder] ! encode.cancelled
    -- Acknowledgement that CLC was output by encoder
    dec.handle[to.decoder] ? CASE cancel.encode.ack
      -- Return confirmation
      dec.handle[from.decoder] ! encode.not.cancelled
```

Algorithm 7.3 shows how the 'cancel-encode' operation is implemented in the decoder. Regarding the 'cancel-encode' operation, the decode-handler behaves like an 'invisible' intermediary between the CTB-handler and the decoder. It exchanges all messages between the two, and behaves towards the CTB-handler in exactly the same way as the decoder would. Algorithm 7.4 shows how the 'cancel-encode' operation is initiated in the CTB-handler, and how the CTB-handler reacts.

This mechanism is necessary because the CTB-handler may get an (external or internal) release signal for an NCT-end containing the reading-end of a channel *while* an output is pending in the encoder which serves that channel. 'Pending' here means that the first CLC has already been sent to the encoder (with the encode-handler as an intermediary), but no acknowledgement has come back yet. In such a case, the CTB-handler will initiate the 'cancel-encode' operation, but does not know yet whether the first CLC will actually be cancelled or whether the user-level process had already taken it when the release was triggered. The CTB-handler will only know this *after* getting the reply from the decode-handler. Please note that if there is more than one CLC in the ULC, only the decoder performing the 'cancel-encode' may have

Algorithm 7.4: Initiating 'cancel-encode' operation in the CTB-handler

```
SEQ
  ...  We need to do a 'cancel-encode' because of a release
  ...    (external or internal)
  ...  Send 'cancel-encode' request to decode-handler
  ...  Wait for acknowledgement for first CLC from encode-handler
  -- Send 'cancel-encode' acknowledgement to decode-handler
  int.dec.handle.array[i][to.handler] ! cancel.encode.ack
  -- Get reply from decode-handler
  int.dec.handle.array[i][from.handler] ? CASE
    -- First CLC was cancelled
    encode.cancelled
      SEQ
        ...  Send 'cancel' message to encode-handler
        ...  Pass 'cancel' on to remote CTB-handler
    -- First CLC was taken by user-level process
    --   (can only be the case if there are no 'remaining' CLCs)
    encode.not.cancelled
      SEQ
        ...  Send confirmation to encode-handler that output was successful
        ...  Pass acknowledgement on to remote CTB-handler
```

taken the first CLC. This is obvious from the fact that the 'cancel-encode' operation can only be triggered by a release — which cannot happen in the middle of a ULC.

The encode-handler must be notified whether the ULC was taken by the user-level process or not. This is necessary because if the output was successful, the encode-handler needs to internally claim all unshared NCT-ends that were output as part of the ULC. If the ULC was cancelled, this is not the case.

7.7 Decoding and Encoding the Various occam-π Protocols

The following occam-π protocols are currently supported by the protocol-converters:

- Regular (i.e. non-MOBILE) simple types:
  - INT
  - INT16
  - INT32
  - INT64
  - BYTE
  - BOOL
  - REAL32
  - REAL64
- Regular arrays (single- and multi-dimensional) of regular simple types.
- Regular RECORDs containing regular simple types or regular arrays or regular RECORDs.
- Regular arrays (single- and multi-dimensional) of regular RECORDs.
- MOBILE simple types.
- Fixed-size MOBILE arrays (single- and multi-dimensional) of regular simple types or regular RECORDs.
- MOBILE RECORDs containing regular simple types or regular arrays or regular RECORDs.
- Dynamic MOBILE arrays (single- and multi-dimensional) of regular simple types or regular RECORDs.

- Channel-type-ends.
- Counted array `PROTOCOL`s.
- Sequential `PROTOCOL`s of the above.
- Tagged `PROTOCOL`s of the above.

The following occam-π protocols are currently not supported by the protocol-converters:

- '`MOBILE.CHAN`' ends.
- Nested `MOBILE`s such as:
  - `MOBILE` arrays (fixed-size or dynamic) of `MOBILE` simple types.
  - `MOBILE` arrays (fixed-size or dynamic) of `MOBILE` arrays.
  - `MOBILE` arrays (fixed-size or dynamic) of `MOBILE RECORD`s.
  - `MOBILE` arrays (fixed-size or dynamic) of channel-type-ends or '`MOBILE.CHAN`' ends.
  - `MOBILE RECORD`s containing `MOBILE` simple types.
  - `MOBILE RECORD`s containing `MOBILE` arrays.
  - `MOBILE RECORD`s containing `MOBILE RECORD`s.
  - `MOBILE RECORD`s containing channel-type-ends or '`MOBILE.CHAN`' ends.
- New occam-π features such as:
  - `MOBILE PROC`esses.
  - `MOBILE BARRIER`s.
  - Arrays/`RECORD`s of these.

Some of the currently unsupported protocols are not yet fully working in KRoC itself. This applies especially to nested mobile types; the only nested mobiles that are fully working at the moment are dynamic mobile arrays of dynamic mobile arrays of `BYTE`s and dynamic mobile arrays of channel-type-ends. The above list takes into account the general rule on nested mobiles in occam-π, which says that "everything containing a mobile must be mobile itself" — hence there are no 'mobiles-in-non-mobiles' in the above list of unsupported protocols, since they are not allowed in occam-π anyway.

For most of the currently unsupported protocols, supporting them in the future will only require adapting the protocol-converters. Others, such as mobile processes and mobile barriers, will require further adaptations in the internal structure of the pony kernel itself. Some thoughts on that are discussed in Sections 12.2.1 and 12.2.2.

7.7.1 Data-item NLCs

As mentioned above, data-item NLCs are address/size pairs, which are exchanged between the decoder and the decode-handler on the sending node, and between the encode-handler and the encoder on the receiving node. On the sending node, the address may point to any place in memory that holds a piece of data which needs to be sent to the receiving node. This may be the workspace or vectorspace of the process, but also the static or dynamic mobilespace — depending on what kind of data the NLC actually decodes.

On the receiving node, data-item NLCs are address/size pairs as well. However, no matter which user-level protocol is encoded, the address/size pair always refers to a dynamic mobile array (of bytes) that was *detached* by the encode-handler before passing the NLC on to the encoder. This dynamic mobile array contains the relevant data for the NLC, which was received from the network. The encode-handler uses the generic '`DETACH.DYNMOB`' process presented in [SBW03] to detach a dynamic mobile array and acquire its address and size:

```
PROC DETACH.DYNMOB (MOBILE []BYTE dynmob, RESULT INT addr, size)
```

This process leaves the dynamic mobile array variable undefined (i.e. its size-slot set to zero) and returns the address and size of the data in the array. The address returned is the address of the *actual* data (in the dynamic mobilespace), *not* the address of the array variable (which would be in the normal workspace). This mechanism is the same as with the old generic '`ENCODE.CHANNEL`' process — which also had to be given address/size pairs of detached dynamic mobile arrays.

Non-mobile Data-items

Regular, non-mobile, data-items are decoded as a single NLC which carries their address (in workspace or vectorspace) and size. When the encoder receives an NLC for a regular data-item from the pony kernel, it copies the relevant data from the detached dynamic mobile array to the user-level channel. Once the data has been taken, the encoder frees the memory that was held by the detached array and returns it to the free-list.

Sequential and Tagged Protocols

Sequential and tagged protocols are decoded as several NLCs — one or more NLCs for each CLC, i.e. for each item in the list.[5] The individual items are treated in the same way as if they were the only CLC of the ULC. This applies to all types supported by pony, including mobile data-items and channel-type-ends, whose decoding/encoding is discussed below.

This means that the number of NLCs in the first CLC equals the number of NLCs for the first item in the list. The number of NLCs in the 'remaining' CLCs equals the sum of the NLCs for all remaining items. A special case regarding the number of NLCs are counted array protocols; see below.

A special feature of decoding sequential protocols is that for regular (non-mobile) data-items, the data is copied into a temporary piece of memory, and the address

[5]The tag of a tagged protocol counts as one item, decoded as one NLC.

of that temporary is used for the NLC, *unless* the data-item is the first or the very last CLC of the ULC. This is necessary because the data-item might have been an expression evaluated by the compiler that ended up on the stack, and in a sequential protocol, the same stack address might be used several times. For instance, this ULC:

```
chan ! i + 1; i + 2; i + 3
```

would use the same stack address three times. For the first and the last CLC (i.e. '`i + 1`' and '`i + 3`'), this would be no problem. The first CLC is decoded, sent to the remote node and acknowledged before the second CLC is being read by the decoder. The 'remaining' CLCs, however, are sent to the remote node in one go. This means that for the second and the third CLC, the same address (from the stack) would be sent, which would result in the same value (namely '`i + 3`') being output to the receiving user-level process for both of them.

Therefore, the data for all CLCs except the first and the very last needs to be copied, and the address of the copy then used for the NLC that is to be sent to the remote node as part of the 'remaining' CLCs. The very last CLC obviously does not need to be copied since its stack address cannot be re-used by a subsequent CLC. This means that for non-sequential protocols, as well as sequential protocols with only two elements — which will be the bulk of communication in a typical pony application — no copying is necessary. Also, no copying is necessary for mobile data, since even if the data-item is the result of an evaluated expression, the compiler would place the result into the relevant mobilespace before engaging in the CLC, and not communicate from the stack directly.

Counted Array Protocols

A counted array consists of two CLCs, one for the count and one for the array data. According to the general rule for CLCs and NLCs, a counted array would always have to be decoded as two NLCs, namely one for each CLC. For performance reasons, however, it is desirable to decode a counted array as a single NLC — which

can be done as long as both CLCs of the counted array are part of the 'remaining' CLCs. In this case, the count is decoded implicitly in the size of the array data, with the count being the size of the array data divided by the size of the base type of the array.

If the ULC only consists of the counted array, or if the counted array is at the beginning of a sequential protocol, the count and the array data have to be decoded in two separate NLCs. This is necessary in order to enable the ULC to be cancelled if necessary. The decoder will only read the array data from the user-level process after having received the acknowledgement from the remote encoder that the remote user-level process has taken the count; otherwise the ULC is cancelled as described in Section 7.4. Therefore, the CLC containing the count has to be decoded separately, in the same way as an individual data-item of the count type (e.g. `INT` or `BYTE`) would be decoded.

A special case is a count of zero. KRoC has a built-in optimisation which works in such a way that if the count of a counted array happens to be zero, no CLC takes place for the array data anymore. For pony, this means that if the counted array is at the beginning of a ULC, only the count (of zero) is decoded. If nothing follows the counted array, the relevant ULC would contain no 'remaining' CLCs. If the counted array is not at the beginning of the ULC, which means that the count is part of the 'remaining' CLCs, only the array data is decoded as described above. In this case, the size of the relevant NLC would be zero.

Algorithm 7.5[6] shows several examples of decoding counted arrays, varying in the position of the counted array in the ULC, as well as the count being zero or not. '`chan`' here stands for the user-level channel served by the decoder. '`first.clc`' is the message from the decoder to the pony kernel that contains the number of data-item NLCs and channel-type-end NLCs in the first CLC, as well as a flag indicating whether there are any 'remaining' CLCs in the ULC. '`rest.clcs`' is the message

[6]Please note that this is not an 'algorithm' in the proper sense of the word, but a collection of examples.

from the decoder that contains the number of data-item NLCs and channel-type-end NLCs in the 'remaining' CLCs.

The encoding of a counted array is done by copying, since both the count and the array data are regular (non-mobile) data.

Static Mobiles

Static mobiles are normally communicated in KRoC using pointer-swapping. Both the source and the target variable are pointers to pre-allocated pieces of memory located in the static mobilespace of the occam-π program. When a communication between two static mobile variables takes place, their pointers are swapped, and the source variable is treated as undefined afterwards. Decoding a static mobile variable does not involve pointer-swapping. Instead, the protocol-decoder simply uses the address and size of the data (in mobilespace) for the NLC that is to be sent to the remote node, and leaves the pointer of the variable unchanged. After the CLC is completed, the source variable is left undefined as usual.

Encoding a static mobile is done using pointer-swapping, i.e. the user-level process will not notice any difference from a non-networked communication. Inside the encoder, the operation is more complex, however, since the encoder must be able to encode static mobiles of *any* type, whose size it does not know until the relevant NLC arrives from the network. Therefore, the encoder uses a special pool of memory. This pool is shared by all encoders across the entire pony-enabled program.

When an NLC for a static mobile data-item arrives in the encoder, the encoder browses the static mobile pool for a piece of memory of the relevant size. If there is a piece of memory of that size, it is taken out of the pool; otherwise the encoder allocates a piece of memory of the relevant size from the *dynamic* mobilespace. The data from the detached array is then copied into the memory from the pool (resp. the newly allocated memory), after which the memory that was held by the detached array is freed and returned to the free-list. Then the memory from the pool is used

Algorithm 7.5: Several examples of decoding counted arrays

```
* chan ! 0::array
  first.clc; 1; 0; FALSE  -- the count (= 0)

* chan ! 0::array; an.integer
  first.clc; 1; 0; TRUE  -- the count (= 0)
  rest.clcs; 1; 0        -- the ‘an.integer’

* chan ! count.greater.than.zero::array
  first.clc; 1; 0; TRUE  -- the count
  rest.clcs; 1; 0        -- the array

* chan ! count.greater.than.zero::array; an.integer
  first.clc; 1; 0; TRUE  -- the count
  rest.clcs; 2; 0        -- the array, the ‘an.integer’

* chan ! a.byte; 0::array
  first.clc; 1; 0; TRUE  -- the ‘a.byte’
  rest.clcs; 1; 0        -- the array (of size 0)
                         -- [no count sent since count is implicit]

* chan ! a.byte; 0::array; an.integer
  first.clc; 1; 0; TRUE  -- the ‘a.byte’
  rest.clcs; 2; 0        -- the array (of size 0)
                         -- [no count sent since count is implicit]
                         -- the ‘an.integer’

* chan ! a.byte; count.greater.than.zero::array
  first.clc; 1; 0; TRUE  -- the ‘a.byte’
  rest.clcs; 1; 0        -- the array (of size > 0)
                         -- [no count sent since count is implicit]

* chan ! a.byte; count.greater.than.zero::array; an.integer
  first.clc; 1; 0; TRUE  -- the ‘a.byte’
  rest.clcs; 2; 0        -- the array (of size > 0)
                         -- [no count sent since count is implicit]
                         -- the ‘an.integer’
```

for an ordinary mobile output operation; the user-level process takes the static mobile by performing the normal pointer-swap.

After the output operation has been completed, i.e. after the pointers have been swapped, the encoder returns the piece of memory whose pointer it just received from the user-level process to the static mobile pool. Using this mechanism, the pool, which is initially empty, grows whenever a static mobile NLC of a new size arrives from the network. The same applies if there has been a static mobile NLC of that size before, but the relevant piece of memory from the pool is currently in use by an encoder.

The items in the static mobile pool are never freed until the operating system automatically frees them anyway at the end of the pony-enabled program. This is because the encoder does not know whether the pointer that was returned by the user-level process during the swap points into the static or the dynamic mobilespace. The latter would be the case if the static mobile variable in the user-level process had been used for networked communication before. This means that after a while, the static mobile pool will typically contain pieces of memory from both the static and the dynamic mobilespace. Accordingly, throughout the pony-enabled program, there will be both static mobile variables pointing to the static mobilespace and static mobile variables pointing to the dynamic mobilespace.

Although the static mobile pool may appear ever-growing (and therefore dangerous) at first glance, this is not the case. Since the items in the pool are re-used when possible, there is a saturation in the growth of the pool after some time. This is because the longer the program runs, the more likely it gets that the pool will already contain a piece of memory of the size needed for a particular NLC — which means that the likelihood of having to add a new item to the pool will get smaller and smaller.

Dynamic Mobile Arrays

Dynamic mobile arrays are allocated at runtime, i.e. their size is not known to the compiler. Dynamic mobile array variables are implemented as *(n + 1)* words in workspace — one for the pointer to the array data (in the dynamic mobilespace), and n words for the dimension counts, where n is the number of dimensions of the array.

When decoding and encoding dynamic mobile arrays, the protocol-converters distinguish between single- and multi-dimensional arrays. Single-dimensional arrays are decoded as a single NLC containing the address and size of the array data (in the dynamic mobilespace). The dimension count is decoded implicitly in the size of the array data. As for counted array protocols (see above), the dimension count can be calculated by dividing the size of the array data by the size of the base type of the array. On the receiving node, the encoder stores the address of the detached dynamic mobile array, as well as the calculated dimension count, in a temporary dynamic mobile array variable. This is then output to the user-level process.

Please note that no copying of array data is necessary here. The memory from the detached array is *not* freed, since it is now used by the user-level process. Only the temporary dynamic mobile array variable is freed by the encoder after the output to the user-level process has been completed. On the sending node, after receiving the acknowledgement, the decoder frees the memory of the source dynamic mobile array variable and returns it to the free-list. The dimension count is set to zero, so that the variable is now undefined for the sending user-level process.

Multi-dimensional dynamic mobile arrays are decoded as two NLCs. The first NLC contains the dimension counts of the source dynamic mobile array variable (with the address pointing to the start of the dimension counts in workspace). The second NLC decodes the array data in the dynamic mobilespace as for single-dimensional arrays. The encoder again uses a temporary dynamic mobile array variable, into which it copies the dimension counts from the first NLC, after which the memory

from the detached array of the first NLC is freed and returned to the free-list. The rest is the same as for single-dimensional arrays, i.e. the address of the detached array of the second NLC is stored in the temporary variable etc.

7.7.2 Channel-type-end NLCs

Making a Non-networked CTB Networked

When encountering a channel-type-end NLC, the decoder must check whether the CTB is already networked, and make it networked if necessary, *before* outputting the actual NLC to the decode-handler. This was described in Algorithm 6.2. Since this algorithm was presented relatively early in this thesis, certain details could not yet be discussed. Therefore, Algorithm 7.6 contains a more detailed description.

When the decode-handler gets the request from the decoder, it notifies the NCT-manager on the master node. This is done through the master-link-handler, in the same way as during an explicit allocation (cf. Section 6.4.2). The following information is passed to the NCT-manager:

- The NCT-ID and *channel-ID* of the decode-handler making the request. The channel-ID is the ID of the channel in the CTB. The corresponding index in the decode-handle-array is used as the channel-ID.
- A Boolean flag indicating whether the client-end of the CTB is currently claimed on our node. This is true if the client-end is shared and claimed, or if it is unshared. In the latter case, it is treated as being internally claimed.
- A Boolean flag indicating whether the server-end of the CTB is currently claimed on our node.

Unlike for explicit allocation, the type-hash of the channel-type does not need to be given to the NCT-manager, since no other ends of an implicitly allocated NCT may ever be allocated explicitly. Hence, a check of the type-hash will never be necessary

Algorithm 7.6: Making a CTB networked — detailed description

```
SEQ
  ...  Claim state-semaphore
  IF
    ...  CTB is non-networked (i.e. network-hook is null)
      SEQ
        ...  Check state of client and server in state-field
        ...  Notify decode-handler about
        ...      * CTB-pointer
        ...      * client- and server-state
        ...      * number of channels in CTB
        ...      * number of readers-in-server
        ...  Wait for pony kernel to implicitly allocate a new NCT
        ...    for the CTB
        ...  Get reply from decode-handler containing
        ...      * NCT-ID
        ...      * network-hook-handle
        ...      * decode-handle-array
        ...      * encode-handle-array
        ...  Store NCT-ID in state-field
        ...  Set up network-hook pointer with channel-words in
        ...    network-hook-handle
        ...  Decrease reference-count of network-hook-handle
        ...  Fork off a decoder and an encoder for each channel-word
        ...    in the CTB
    TRUE  -- i.e. CTB is networked (network-hook <> null)
      SKIP
  ...  Release state-semaphore
```

in the NCT-manager. The NCT-manager now forks off a new NCT-handler, which takes into account whether the client-end and/or the server-end are currently claimed on our node when setting up its initial state.

The decode-handler waits for a reply from the NCT-manager, which will come via the link from the master node. When getting the reply, the master-link-handler passes it on to the decode-handler via the *decode-reply-handle*. In order for the master-link-handler to acquire the correct decode-reply-handle, it needs the NCT-ID and channel-ID of the decoder — which is why this information was sent along with the

original request to the NCT-manager. To get the decode-reply-handle, the master-link-handler first contacts the CTB-manager via the CTB-manager-handle, and requests the *CTB-instant-handle* of the CTB-handler for the corresponding NCT-ID. Then the master-link-handler contacts the CTB-manager via the CTB-instant-handle and requests the decode-reply-handle for the corresponding channel-ID. Having got the decode-reply-handle, the master-link-handler passes the reply from the NCT-manager on to the decode-handler.

As can be seen from Figure 6.1, there is a small sub-process inside the CTB-handler called the *instant-handler*. This deals with requests that need to be answered instantly while the 'main' CTB-handler may be engaged in other things (which may involve communication with other components), in order to avoid deadlock.

The only purpose of the decode-reply-handle is passing the reply from the NCT-manager on to the decode-handler. This reply contains the NCT-ID of the newly (implicitly) allocated NCT. Having got that, the decode-handler sends an implicit allocation request to the CTB-manager via the CTB-manager-handle. This request contains the following information:

- The NCT-ID.
- The CTB-pointer.
- A Boolean flag indicating whether the client-end of the CTB is currently claimed on our node.
- A Boolean flag indicating whether the server-end of the CTB is currently claimed on our node.
- The number of channels in the CTB.
- The number of readers-in-server.

The CTB-manager stores the CTB-pointer for the new NCT-ID and forks off a new CTB-handler, which for its part forks off a decode-handler and an encode-handler

for each channel in the CTB. The new CTB-handler takes into account whether the client-end and/or the server-end are currently claimed on our node when setting up its initial state. Now the CTB-manager sends a reply to the decode-handler, containing the network-hook-handle and the decode- and encode-handle-arrays (which have just been allocated by the CTB-manager). These are then returned to the decoder, together with the NCT-ID of the implicitly allocated NCT.

The decoder then continues making the CTB networked as set out in Algorithm 7.6. This involves forking off a set of protocol-converters for each channel in the CTB. The same rules for the order of the handles in the decode- and encode-handle-arrays apply as during an explicit allocation.

Please note that the mechanism in the decoder that checks whether a CTB is networked or not, and makes it networked if necessary, is *not* part of, but done *before* the actual channel-type-end NLC. Hence, it will *not* be reversed if a started ULC is cancelled. Once a CTB is networked, it will stay networked for the rest of its lifetime. So, even if a started ULC is cancelled, only the actual channel-type-end NLC is cancelled; the CTB itself will remain networked.

Actual Channel-type-end NLCs

The actual channel-type-end NLC performed by the decoder consists of a single message to the decode-handler containing the following information about the NCT-end that is to be sent:

- The NCT-ID.
- The direction-type (client-end or server-end).
- The share-type (unshared or shared).

The decode-handler will internally release the end in case it is unshared. Then the NLC gets packed into a suitable network format (cf. Section 7.8.2) and passed on

to the CTB-handler, which will send it to the remote node. There, the relevant CTB-handler passes it on to the encode-handler.

When the encode-handler on the receiving node encounters a channel-type-end NLC, it requests the allocation of a new channel-type-end for the given NCT-ID from the CTB-manager. This is done using the same request as during an explicit allocation — the CTB-manager does not know whether the request comes from the main kernel during an explicit allocation or from an encode-handler dealing with a channel-type-end NLC. As discussed earlier, the reply from the CTB-manager may either be a notification that the reference-count of an existing CTB must be increased, or a notification that a new CTB must be allocated. In either case, the reply is forwarded to the encoder.

If the reference-count of an existing CTB must be increased, the message from the encode-handler to the encoder contains the CTB-pointer of the existing CTB. After the encoder has increased the CTB's reference-count, the channel-type-end NLC is ready to be output to the user-level channel.

If a new CTB must be allocated, the message from the encode-handler to the encoder contains the NCT-ID for the received NCT-end. The encoder will now do the following:

- Allocate a new CTB according to the type-descriptor.
- Initialise the reference-count to 1.
- Store the type-descriptor pointer in the CTB.
- Initialise the client-, server- and state-semaphores.
- Store the NCT-ID in the state-field.

The encoder then sends a reply to the encode-handler containing the CTB-pointer of the newly allocated CTB, the number of channels in the CTB, and the number

of readers-in-server. As during an explicit allocation, the CTB-manager is now notified about the number of channels and the number of readers-in-server. The CTB-manager sends a reply to the encode-handler, containing the network-hook-handle and the decode- and encode-handle-arrays (which have just been allocated by the CTB-manager). These are then passed on to the encoder, which will complete the allocation by doing the following:

- Set up the network-hook pointer with the channel-words in the network-hook-handle.
- Decrease the reference-count of the network-hook-handle.
- Fork off a protocol-decoder and a protocol-encoder for each channel-word in the CTB.

When the protocol-converters are forked off, the same rules for the order of the handles in the decode- and encode-handle-arrays apply as explained above. After the encoder has completed the allocation, the channel-type-end NLC is ready to be output to the user-level channel.

The encode-handler completes the channel-type-end NLC by notifying the CTB-manager about the CTB-pointer of the newly allocated CTB. The CTB-manager stores the CTB-pointer and forks off a new CTB-handler, which for its part forks off a decode-handler and an encode-handler for each channel in the CTB. The reason why the CTB-pointer was not given to the CTB-manager earlier, together with the number of channels and the number of readers-in-server, is that in this way, we can use the same request for allocating an NCT-end both in the main kernel during an explicit allocation and in the encode-handler during a channel-type-end NLC. Thus, the corresponding mechanism in the CTB-manager only had to be implemented once.

As pointed out earlier, after the first CLC has been output to the user-level channel, the encoder acknowledges this to the encode-handler, which passes the acknowledgement on to the CTB-handler. If the output of the first CLC was successful (i.e.

taken by the user-level process), the CTB-handler either notifies the encode-handler about it (if there is only one CLC in the ULC) or sends the 'remaining' CLCs to the encode-handler (in which case the encode-handler automatically knows that the output was successful). If the output of the first CLC was cancelled during a 'cancel-encode' operation, the CTB-handler sends a 'cancel' message to the encode-handler, which will pass the 'cancel' message on to the encoder if there is more than one CLC in the ULC. After the encode-handler has acknowledged the output of the 'remaining' CLCs (if there are any) to the CTB-handler, no notification is returned, because it is obvious that only the user-level process may have taken the 'remaining' CLCs.

At this point, the encode-handler knows whether the ULC has been taken by the user-level process. If this is the case, the encode-handler now internally claims all unshared NCT-ends that were output to the user-level process as part of this ULC. If the first CLC was cancelled, the decode-handler on the sending node re-claims all previously released unshared ends that were part of the cancelled first CLC.

Dealing With the Reference-count

The general rule for dealing with the reference-count of networked CTBs during the operation of pony is that when it needs to be increased, this is done *before* completing the operation that requires the increase; when it needs to be decreased, this is done *after* completing the relevant operation, or when a previous operation is cancelled. The reference-count is *increased* in the following situations:

- In the allocation process during an explicit allocation.
- In the encoder during a channel-type-end NLC.

This applies both to new CTBs (whose reference-count is initialised to 1) and existing CTBs (whose reference-count is 'really' increased), and is done before the user-level process gets the relevant NCT-end.

The reference-count is *decreased* in the following situations:

- In the decoder after receiving the acknowledgement for the first CLC or the 'remaining' CLCs. This is done for *all* channel-type-end NLCs in the relevant CLC(s) at once.
- In the decoder during a 'cancel-encode' operation. Again, this is done for all channel-type-end NLCs in the first CLC that is cancelled.

It is worth noting that the same rules as for increasing and decreasing the reference-count of CTBs also apply for operations related to data-item NLCs. This particularly applies to the allocation and deallocation of memory for dynamic mobiles.

Dealing With Internal Claims and Releases

The rule for internally claiming and releasing unshared NCT-ends during the operation of pony is that when they need to be claimed, this is done *after* completing the operation that requires the internal claim, or when a previous operation is cancelled; when they need to be released, this is done *before* completing the relevant operation. Unshared NCT-ends are internally *claimed* in the following situations:

- In the main kernel after an explicit allocation.
- In the encode-handler after the entire ULC has been taken by the user-level process. This is done for *all* channel-type-end NLCs in the relevant ULC at once.
- In the decode-handler, when the first CLC has been cancelled. In this case all previously released unshared ends are re-claimed.

When a previously non-networked CTB is being made networked, unshared ends are treated as being internally claimed when the new NCT-handler and the new CTB-handler are set up. Hence, no internal claim signal needs to be sent.

Unshared NCT-ends are internally *released* in the following situations:

- In the decode-handler during a channel-type-end NLC.
- During the shutdown of the pony kernel. See Section 10.5 for details about pony's shutdown mechanism.

7.8 Decode-handler and Encode-handler

The decode-handler and the encode-handler present the interface between the protocol-converters and the CTB-handler. They are forked off by the CTB-handler when the CTB-handler starts. Their general structure is the same as that of the decoder and the encoder, and will therefore not be repeated in this section. Instead, we will focus on the differences between the protocol-converters on the one hand, and the decode-handler and the encode-handler on the other.

7.8.1 Differences Compared With the Protocol-converters

Most messages passing through the decode-handler and the encode-handler are just forwarded between the protocol-converters and the CTB-handler. The decode-handler and the encode-handler behave towards the protocol-converters in the same way as the CTB-handler behaves towards them, and towards the CTB-handler in the same way as the relevant protocol-converter behaves towards them. There are only the following minor exceptions to this rule:

- While between the protocol-converters and the decode- or encode-handler, all NLCs are exchanged separately (see above), between the decode- or encode-handler and the CTB-handler, the entire first CLC or the entire 'remaining' CLCs are exchanged at once as a special *CLC-packet*. For this, all NLCs of the first CLC, or of the 'remaining' CLCs, are packed by the decode-handler and unpacked by the encode-handler. Details about this are given in Section 7.8.2.

- Between notifying the CTB-handler about the number of NLCs in the first CLC or in the ‘remaining’ CLCs and passing the relevant CLC-packet to the CTB-handler, the decode-handler sends a special message to the CTB-handler if the CLC-packet contains a channel-type-end NLC where an NCT-end is sent over itself.[7] This would need to be treated in the CTB-handler as a special case; see Section 8.1.6 for details.

- As mentioned in Section 7.7.2, if there is a ULC with only one CLC, the CTB-handler notifies the encode-handler whether the output was successful or not, whereas the encode-handler does not send such a notification to the encoder. The reason for this is that the encoder does not care about a cancelled first *and only* CLC, since the handling of such a ULC is completed anyway after the first CLC has been output, no matter whether the user-level process has taken it or the matching decoder during a ‘cancel-encode’ operation. The encode-handler, on the other hand, needs to know whether it was actually the user-level process that has taken the ULC, because only then it would internally claim the unshared NCT-ends that were output as part of the ULC.

Apart from these exceptions, the communication structure between the decoder and the decode-handler is identical to the one between the decode-handler and the CTB-handler. The same applies to the communication structure between the CTB-handler and the encode-handler and the one between the encode-handler and the encoder accordingly.

The other differences between the decode-handler and the encode-handler on the one hand and the protocol-converters on the other, which do not affect the communication structure as discussed above, are the following things related to the handling of channel-type-end NLCs:

[7]The decode-handler finds out about this by comparing the NCT-ID of the end that is being sent with the ID of its own NCT. If they are identical, and the end that is being sent is unshared, this means that the end is sent over itself.

- Making a CTB networked if necessary in the decode-handler. This involves communication with the NCT-manager on the master node, and with the CTB-manager.
- Dealing with incoming channel-type-end NLCs in the encode-handler. This involves communication with the CTB-manager.
- Internally claiming or releasing unshared NCT-ends where applicable. This involves communication with the CTB-manager, and with the responsible CTB-handler via the relevant CTB-claim-handle.

All these things were discussed in Section 7.7.2.

7.8.2 CLC-Packets

A CLC-packet contains all NLCs of a CLC — either of the first CLC, or of all 'remaining' CLCs. Actually, a CLC-packet is not a single 'packet', but consists of two arrays in a sequential protocol. In this way, it is passed from the decode-handler to the CTB-handler, and then on to the link-handler that holds the link to the target node. After arriving on the target node, the link-handler there forwards the CLC-packet to the relevant CTB-handler, which in turn passes it on to the encode-handler responsible for our networked channel. Details about this are given in Section 8.1.5.

CLC-packets look different on the sending node and on the receiving node. In the decode-handler, they are assembled as an *address-array* and a *size-array*. These arrays contain the addresses and sizes of pieces of data that are relevant for the CLC. These are sent in a single operation over the link; see Section 9.3. On the receiving node, CLC-packets consist of a *data-array* and a size-array. The data-array is a dynamic mobile byte array containing the entire data received over the link as part of the CLC, consecutively in a single chunk of memory. The size-array is the same as on the sending node. Naturally, the sum of all sizes in the size-array equals the size of the data-array.

If there are no channel-type-end NLCs in the relevant CLC, all pieces of data in the CLC-packet are data-item-NLCs. On the receiving node, they are copied by the encode-handler from the data-array into separate dynamic mobile arrays. If the CLC-packet only contains one data-item NLC, the data-array is used without copying. For each NLC, the encode-handler detaches the relevant dynamic mobile array (cf. Section 7.7.1) and passes the address/size pair on to the encoder.

If the relevant CLC contains channel-type-end NLCs, the CLC-packet contains a special piece of data, called the *CLC-descriptor*. This is located at the very end of the CLC-packet, after all data-item NLCs that are part of it. The CLC-descriptor describes the layout of the CLC. For each NLC, it stores whether it is a data-item NLC or a channel-type-end NLC. This is necessary to preserve the correct order of the NLCs in the CLC when sending them from the encode-handler to the encoder. For each channel-type-end NLC, the CLC-descriptor additionally contains the NCT-ID, the direction-type and the share-type of the relevant NCT-end.

The CLC-descriptor is organised as a byte array. It consists of consecutively laid out *NLC-descriptors*. The size of an NLC-descriptor is one byte for data-item NLCs or five bytes (one byte plus an integer) for channel-type-end NLCs. So, the total size of the CLC-descriptor in bytes is the number of data-item NLCs plus five times the number of channel-type-end NLCs in the CLC.

The first (and, for data-item NLCs, only) byte of the NLC-descriptor denotes the type of the NLC (data-item or channel-type-end), plus the direction-type and the share-type of the NCT-end in case of channel-type-end NLCs. Table 7.1 shows the layout of that byte. 'Bit 1' here means the least significant bit; the five most significant bits are always zero. If the NLC is a channel-type-end NLC, the remaining four bytes of the NLC-descriptor contain the NCT-ID of the NCT-end.

The decode-handler uses the following generic process to determine the address and size of the CLC-descriptor after it has been allocated:

```
PROC DECODE.DATA (* data, RESULT INT addr, size)
```

Bit					
Bit 1	0 data-item NLC	1 channel-type-end NLC			
Bit 2	0	0 client-end		1 server-end	
Bit 3		0 shared	1 unshared	0 shared	1 unshared
Total value	000 0	001 1	101 5	011 3	111 7

Table 7.1: Layout of the first byte of an NLC-descriptor

Similarly to ‘`DETACH.DYNMOB`’, this process returns the address and size of an occam-π variable. ‘`*`’ may be any occam-π type here. Unlike ‘`DETACH.DYNMOB`’, however, ‘`DECODE.DATA`’ leaves the variable untouched, i.e. it is still defined when the process returns. The decode-handler stores the address and size of the CLC-descriptor at the last index of the address- and size-arrays of the CLC-packet. It is ensured that the address/size pair for the CLC-descriptor remains valid until the CLC-packet has been sent to the remote node, because the CLC-descriptor remains in scope until the acknowledgement for the CLC has been received.

The encode-handler on the receiving node extracts the CLC-descriptor from the data-array of the CLC-packet. Then it runs through the CLC-descriptor. When the encode-handler encounters a data-item NLC, it copies the next piece of data into a new dynamic mobile array, detaches it, and passes the address/size pair on to the encoder. When a channel-type-end NLC is encountered, the relevant operation is started according to the content of the NLC-descriptor for that NLC.

Chapter 8

Handlers and Managers for CTBs and NCTs

Figure 8.1 shows the CTB-handler with its sub-components, the CTB-manager, the NCT-handler and the NCT-manager. These pony components are discussed in this chapter.

8.1 The CTB-handler

The CTB-handler deals with the function of an individual networked CTB. This involves handling incoming claim and release requests for the ends of the CTB, as well as the communication along its channels. The latter is done using the decode- and encode-handlers, which the CTB-handler forks off when it starts. The CTB-handler can be contacted by link-handlers via the *CTB-main-handle*.[1] This is available from the CTB-manager on request, as are the CTB-instant-handle and the two CTB-claim-handles, which will be discussed in Sections 8.1.1 and 8.1.2.

[1]No other components, except the CTB-manager during shutdown (cf. Section 10.5), will ever contact the CTB-manager via the CTB-main-handle.

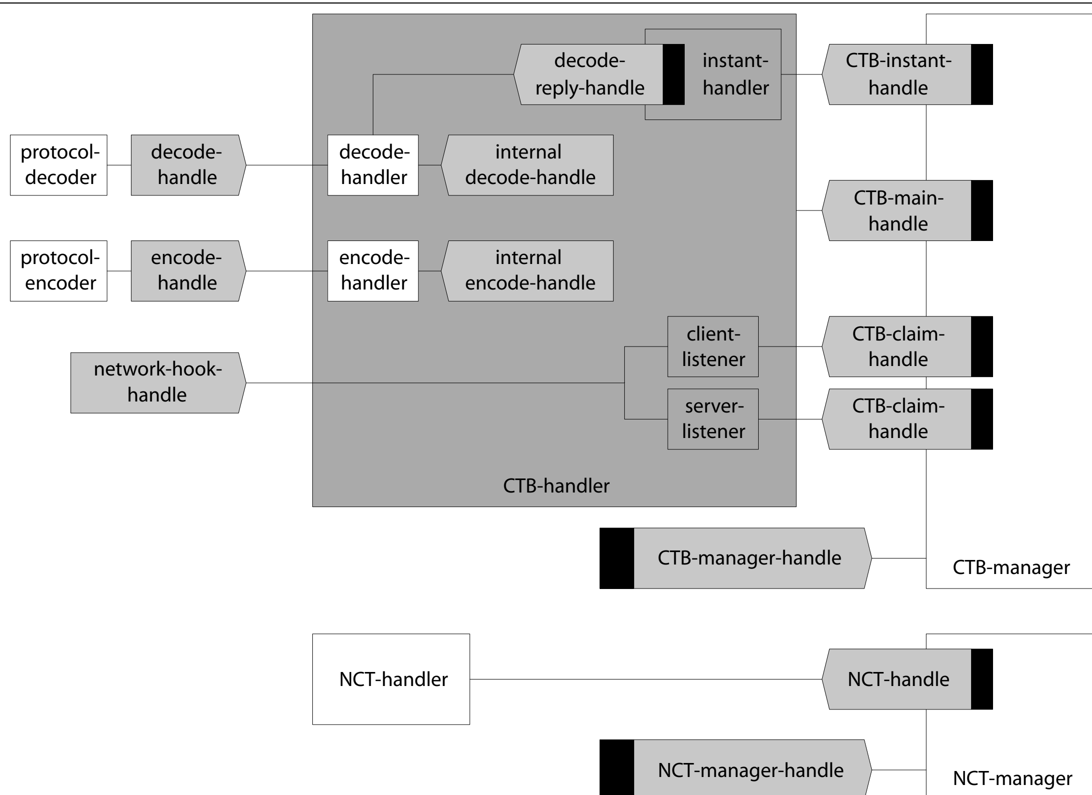

Figure 8.1: pony components related to CTBs and NCTs

8.1.1 The Instant-handler

As already mentioned in Section 7.7.2, the instant-handler is a small sub-process of the CTB-handler dealing with requests that need to be answered instantly while the 'main' CTB-handler may be engaged in other things. Since during its function, the 'main' CTB-handler needs to communicate with other pony components, it would not be available to answer such 'instant' requests via the CTB-main-handle. In the worst case, this would lead to deadlock; to avoid that, we need the instant-handler.

The instant-handler can be contacted by other pony components via the CTB-instant-handle. The following requests are handled by the instant-handler:

- Getting the decode-reply-handle for a given channel-ID. This is used by the link-handler for replies to the decode-handler coming from the NCT-manager. Details were discussed in Section 7.7.2.

- Getting the current remote node of the CTB. This is used by the error-handler during the call of the '`pony.err.get.current.remote.node`' process (cf. Section 4.1.3).

8.1.2 Client-listener and Server-listener

As can be seen from Figure 8.1, the *client-listener* and the *server-listener* are two small sub-processes of the CTB-handler that listen to claim and release signals for the ends of the CTB. This applies to both external and internal claim or release signals.

Both the client-listener and the server-listener are `ALT`ing over the relevant channel of the network-hook-handle and the relevant CTB-claim-handle. In both cases, this is done using the extended rendezvous. When an (external or internal) claim or release signal arrives, the listener notifies the 'main' CTB-handler about it via an internal channel.

The 'main' CTB-handler then deals with the claim or release request. Doing this, it is irrelevant for the 'main' CTB-handler whether the request was external or internal — they are handled in exactly the same way. After the handling of the claim or release request has been completed, the 'main' CTB-handler sends a confirmation to the relevant listener, which will then release the channel in the network-hook-handle or the CTB-claim-handle from the extended rendezvous.

8.1.3 Sessions for NCTs

In order to understand the internal function of the CTB-handler, it is necessary to understand the concept of *sessions* in pony. There are three general states in which a CTB-handler can be (plus two special ones discussed in Section 8.1.6):

- outside a session
- inside an external session

- inside an internal session

This distinction will be referred to as the *session-state* of the CTB-handler in the following. Being inside a session means that one end of the NCT is claimed on our node while the opposite end of the NCT is claimed as well. If the opposite end is claimed on another node, the session is *external*. In this case, the CTB-handler on our node communicates with the CTB-handler on the remote node via the link between the two nodes. If both ends of the NCT are claimed on our node, the session is *internal*.

During external sessions, the CTB-handlers on both nodes activate the decode-handlers (which in turn activate the decoders) of the channels whose writing-ends are in the NCT-end that is claimed on their node.[2] In this way, for each channel in the NCT, exactly one decoder is activated — either on the client-end node or on the server-end node, depending on the direction of the channel in the channel-type.

During internal sessions, all decoders are deactivated. Since both ends of the NCT are claimed on the same node, they can both access the channel-words of the same CTB. Therefore, channel communication during internal sessions does not involve the protocol-converters at all, but works in the 'traditional' way; the outputting and the inputting user-level process simply access the relevant channel-word.

To handle claim and release requests, the CTB-handler needs to communicate with the NCT-handler that is responsible for our NCT. This is done in the usual way, using the link between our node and the master. When the link-handler gets a message for a CTB-handler, it requests the relevant CTB-main-handle from the CTB-manager and then passes the message on. Likewise, if a message for an NCT-handler arrives, the link-handler requests the relevant NCT-handle from the NCT-manager before forwarding the message. During external sessions, the CTB-handler communicates with the CTB-handler on the node which holds the opposite end of the NCT. Again, this is done over the link between the two nodes.

[2]To which channels this applies is determined by using the number of readers-in-server, and the correct order of the handles in the decode- and encode-handle-arrays; cf. Section 6.4.2 for details.

There are three channels in the CTB-main-handle from which the CTB-handler can input — one for messages coming from the NCT-handler regarding claim or release requests for the client-end, one corresponding channel for the server-end, and one channel for messages coming from the remote CTB-handler during external sessions. There is a clear separation between when a CTB-handler is outside a session, and when it is inside a session. The CTB-handler communicates with the NCT-handler if and only if it is outside a session; it communicates with a remote CTB-handler if and only if it is inside an (external) session.

8.1.4 Starting a Session

When the CTB-handler gets a claim signal for one of its ends from the relevant listener, the claim request is passed on to the NCT-handler. The NCT-handler maintains queues for the client-end and for the server-end of the NCT, where it stores nodes from which it has received claim requests for the respective end. When a claim request arrives in the NCT-handler, it is stored at the end of the relevant queue.

The NCT-handler deals with claim requests either immediately when they arrive — if no other node has currently claimed the respective end — or after the node that had previously claimed the end has released it. In both cases, the NCT-handler checks whether the opposite end of the NCT is currently claimed. If this is the case, a 'new-other-end' message is sent to the CTB-handler of the opposite end. Otherwise, a 'claim-confirm-nootherend' message is sent to the CTB-handler from which the claim request came. What these messages mean is explained below.

Both ends of a CTB can be in one of three possible states in the CTB-handler:

- released
- pending
- claimed

'Pending' means that the CTB-handler has sent a claim request for the end to the NCT-handler, but not received a reply yet. 'Claimed' means that our node has reached the top of the respective queue in the NCT-handler, and the NCT-handler has therefore sent a claim confirmation for the end to the CTB-handler. There are two possible claim confirmations that may arrive for a pending NCT-end: 'claim-confirm-nootherend' and 'claim-confirm-otherendclaimed'. Which of the two confirmations arrives from the NCT-handler depends on whether the opposite end of the NCT is currently claimed on another node or not. If a 'claim-confirm-nootherend' message arrives, the CTB-handler changes the state of the pending end to 'claimed' and sends a confirmation to the relevant listener, but does not start a session (i.e. the session-state remains unchanged).

If an end is claimed in a CTB-handler while there is no session[3], there are two things that may happen — either a release signal comes in from the relevant listener (which would need to be forwarded to the NCT-handler), or a 'new-other-end' message (carrying a node-ID) arrives from the NCT-handler. The latter means that a new claim request for the opposite end of the NCT is now being dealt with by the NCT-handler. Since both things may happen at the same time, we need a mechanism to avoid race conditions. Therefore, release requests are generally confirmed by the NCT-handler with a dummy 'new-other-end' message[4] *unless* the NCT-handler had sent a real 'new-other-end' message before, for which it is now awaiting a confirmation from the CTB-handler. The following sequences of events may occur while an NCT-end is claimed but the CTB-handler is outside a session:

- The end gets released. The CTB-handler sends a release request to the NCT-handler. The NCT-handler receives the request and returns a dummy 'new-other-end' message as confirmation. When it has arrived, the CTB-handler

[3]This may be the case either if the end has only just been claimed and the CTB-handler received a 'claim-confirm-nootherend' message from the NCT-handler, or if there was a session before which was closed (cf. Section 8.1.5) because the opposite end was released.

[4]carrying a dummy node-ID of -1

changes the state of the end to 'released' and sends a confirmation to the relevant listener.

- The end gets released. The CTB-handler sends a release request to the NCT-handler. At the same time, the NCT-handler sends a real 'new-other-end' message to the CTB-handler. The CTB-handler discards the 'new-other-end' message (in fact, it does not care whether it is a real or a dummy message), changes the state of the end to 'released', and sends a confirmation to the relevant listener. The NCT-handler then handles the incoming release request.
- The NCT-handler sends a real 'new-other-end' message to the CTB-handler. The CTB-handler takes the message and handles it.

In the first case, the NCT-handler removes the node of the CTB-handler from which the release request came from the relevant queue. Then it checks whether there is another node in that queue. If this is the case, the NCT-handler now sends a 'claim-confirm-nootherend' message to the new node, since the opposite end is currently not claimed.

If the NCT-handler gets a release request from a CTB-handler to which it has just sent a 'new-other-end' message (the second case), it removes that CTB-handler's node from the relevant queue. Then it sends a 'claim-confirm-nootherend' message to the CTB-handler of the opposite end. If there is another node in the queue for the first end, a 'new-other-end' message, containing the ID of the new node in the first queue, is sent immediately to the opposite CTB-handler (which will therefore receive a 'claim-confirm-nootherend' and a 'new-other-end' message from the NCT-handler in quick succession).

If the CTB-handler gets a real 'new-other-end' message from the NCT-handler (the third case), it checks the node-ID that comes with it. If it is different from the ID of its own node, the CTB-handler can now start an external session (i.e. the session-state is changed to 'external session'). To start an external session, the CTB-handler contacts the link-manager to acquire the link-handle for the link-handler

that is responsible for the remote node. Then the CTB-handler activates all decode-handlers for channels whose writing-ends are in the NCT-end claimed on our node. When this is done, the CTB-handler sends a 'new-other-end-confirm' message to the NCT-handler, and waits for a 'start-session' message from the remote CTB-handler.

When the NCT-handler gets a 'new-other-end-confirm' message, it knows that the end for which the 'new-other-end' message was sent has not been released. In this case, it now sends a 'claim-confirm-otherendclaimed' message to the other node, *unless* both ends are claimed on the same node.

If the claim confirmation from the NCT-handler for a pending end is 'claim-confirm-otherendclaimed' (which will carry the ID of a remote node), the state of the pending end is changed to 'claimed' and a confirmation is sent to the relevant listener. Now an external session can be started. The CTB-handler changes the session-state to 'external session', acquires the relevant link-handle from the link-manager, and activates the relevant decode-handlers. When this is done, the CTB-handler sends a 'start-session' message to the remote CTB-handler.

If a CTB-handler receives a 'new-other-end' message which carries a node-ID that is identical to the ID of its own node, this means that an internal session needs to be started (i.e. the session-state is changed to 'internal session'). In this case, no link-handle needs to be acquired from the link-manager, and no decode-handlers need to be activated. The 'new-other-end-confirm' message still needs to be sent to the NCT-handler, however, to comply with the procedure mentioned above. In this case, it is obvious that the state of the other end is 'pending'. Since that other end is on our own node, the CTB-handler can now change the other end's state to 'claimed' and send a confirmation to the relevant listener *without* having to await a claim confirmation from the NCT-handler. As mentioned above, the NCT-handler only sends the claim confirmation if both ends are claimed on different nodes.

Analysing the above protocol, it emerges that before an external session is started, one of the nodes receives a 'new-other-end' message and the other node receives a

'claim-confirm-otherendclaimed' message. The 'new-other-end' message always arrives on the one node first and gets acknowledged; only then will the 'claim-confirm-otherendclaimed' message arrive on the other node. In order to preserve a clear separation between the states of the CTB-handler (i.e. between times when it is outside a session and times when it is inside a session), there must be a synchronisation between the two nodes. Therefore, the 'start-session' message is sent from the node that has received the 'claim-confirm-otherendclaimed' message to the node that has received the 'new-other-end' message before the session is actually started. This ensures that no session-related communication is sent from one node to the other before the other node is actually aware of the session.

8.1.5 Handling Sessions

As mentioned above, during internal sessions, all communication over the channels of the NCT is done locally by accessing the channel-words in the CTB directly from the user-level processes which are writing to and reading from the channels. No decoders are active in this case, which means that no session-related messages are exchanged between nodes.

During external sessions, the CTB-handler must forward messages from the decode- and encode-handlers to the CTB-handler on the remote node. This has been discussed in the previous chapters. Communication between the two CTB-handlers is done in the usual way. One of the CTB-handlers sends a message to the relevant link-handler over the link-handle that was acquired at the start of the session. The message is then sent over the link. The link-handler on the receiving node passes the message on to the relevant CTB-handler via its CTB-main-handle.

The CTB-handler maintains a special flag for all channels in the CTB called the *ULC-state*. There are four possible ULC-states:

- No ULC pending.
- First CLC is pending, no 'remaining' CLCs in ULC.

- First CLC is pending, 'remaining' CLCs yet to come.
- 'Remaining' CLCs are pending.

Using the ULC-state, the CTB-handler is able to correctly communicate with its decode- and encode-handlers, both when getting messages from the decode- or encode-handler for the remote node and vice versa. For the handling of ULCs, please refer to Chapter 7. The ULC-state is also important when it comes to closing sessions; see below for details.

During a session, ends that are claimed on our node may be released. In such a case, a release signal arrives in the CTB-handler from the relevant listener. If the session is external, this obviously applies only to one of the ends; if it is internal, both of the ends may be released on our node.

If an end gets released during an internal session, the CTB-handler sends a release request for that end to the NCT-handler. Then it waits for a dummy 'new-other-end' message from the NCT-handler as confirmation. When the CTB-manager has received the confirmation from the NCT-handler, the session-state is changed to 'no session'. The state of the relevant end is changed to 'released', and a confirmation is sent to the relevant listener. If an end gets released during an external session, all this needs to be done as well. Before sending the release request to the NCT-handler, however, the session needs to be closed (see below). This is necessary to maintain a clear separation between the states of the CTB-handler.

When the NCT-handler gets a release request from a CTB-handler while there is a session (i.e. while both ends are claimed), it returns a dummy 'new-other-end' message and removes the CTB-handler's node from the relevant queue. Then it checks whether there is another node in that queue. If this is the case, the NCT-handler now sends a 'new-other-end' message, containing the ID of the new node, to the opposite CTB-handler.

Algorithm 8.1 shows how the closing of a session is initiated.[5] Please note that all decode-handlers can be deactivated straight away since none of them may be pending. This is clear because if the end is shared, it cannot be released in the middle of a ULC. If the end is unshared, the end cannot be moved (which is the only event that can trigger an internal release for an unshared end) in the middle of a ULC (unless the end is being sent over itself, but this is a special case; see Section 8.1.6). Algorithm 8.2 shows how a CTB-handler reacts when it gets a 'close-session' message.

Algorithm 8.1: Initiating the closing of a session

```
SEQ
  ...  We just received a release signal
  ...  Deactivate all active decode-handlers
  ...  Send 'close-session' message to remote CTB-handler
  ...  For all pending encode-handlers
    IF
      ...  'Remaining' CLCs pending
        SEQ
          ...  Wait for acknowledgement (will definitely come)
          ...  Pass acknowledgement on to remote CTB-handler
      TRUE  -- First CLC pending
        SEQ
          ...  Initiate 'cancel-encode'
          ...  Depending on the outcome, pass either cancel
          ...    or acknowledgement on to remote CTB-handler
  -- Wait for confirmation from remote CTB-handler
  INITIAL BOOL running IS TRUE:
  WHILE running
    ctb.main.handle.svr[to.handler] ? CASE
      ...  First CLC of a new ULC for one of the channels
        ...  Discard, send cancel message to remote CTB-handler
      ...  'close-session' message
        running := FALSE
```

After a CTB-handler has reacted to an incoming 'close-session' message, it changes the session-state to 'no session'. The state of the end that is claimed on our own node

[5] Updates of ULC-states for the individual channels are not shown in the algorithms in this chapter in order to keep them short. Please note that no update of the session-state is included in Algorithm 8.1, because the session-state is updated *afterwards*, when the release request has been sent to and confirmed by the NCT-handler. The same applies to the state of the end that has been released.

Algorithm 8.2: Reacting to a 'close-session' message

```
INITIAL INT num.pending.decode.handlers IS 0:
SEQ
  ...  We just received a ‘close-session’ message from remote CTB-handler
  ...  For all active decode-handlers
    IF
      ...  Decode-handler is not pending
        ...  Deactivate decode-handler
      TRUE  -- Decode-handler is pending
        num.pending.decode.handlers := num.pending.decode.handlers + 1
  -- Wait for cancel or acknowledgement for all pending decode-handlers
  SEQ i = 0 FOR num.pending.decode.handlers
    ctb.main.handle.svr[to.handler] ? CASE
      ...  Cancel CLC
        ...  Pass cancel on to decode-handler (this deactivates it)
      ...  Acknowledgement
        SEQ
          ...  Pass acknowledgement on to decode-handler
          ...  Deactivate decode-handler
  ...  Send ‘close-session’ message to remote CTB-handler
  ...  Change session-state to ‘no session’
```

is *not* changed, since the closing of the session was initiated by the *remote* end getting released. Our own end is now in exactly the same state as if it had just been claimed and the NCT-handler had confirmed the claim with a 'claim-confirm-nootherend'.

If both ends of the NCT are getting released at the same time, the CTB-handlers on both nodes will close the session using Algorithm 8.1. This is no problem, however, since the confirmation for a 'close-session' request is the 'close-session' message itself. If both ends get released at the same time, no ULCs (in either direction) may be pending. Therefore, on either node, no encode-handlers may be pending and no 'first CLC' messages may arrive from the remote CTB-handler anymore, which enables a safe handshake between the two CTB-handlers both using Algorithm 8.1.

A special characteristic regarding internal claim and release signals has not yet been discussed. As described in Section 7.7.2, when an unshared NCT-end arrives on our node, it is internally claimed by the encode-handler *after* it has been taken by the user-level process. When an unshared end is sent to another node, it is

internally released by the decode-handler *before* the CLC-packet is passed on to the CTB-handler.

In cases where an NCT-end arrives on our node and is immediately sent away again, the decode-handler responsible for sending the end may try to release the end before the encode-handler that has just output the end to the user-level process has had the chance to claim it. Therefore, the CTB-handler accepts release signals also for ends that are currently released if the CTB-handler is currently not inside a session.[6] The CTB-handler keeps a count for each end which stores how often a release signal for the end was received while the end was released. When such a release signal comes in, the count is increased and a confirmation is sent to the relevant listener. No release request is sent to the NCT-handler in such a case.

When a claim signal comes in for an end whose count of previous dummy releases is greater than zero, the count is decreased and a confirmation is sent to the relevant listener. Again, no message is sent to the NCT-handler. The reason for keeping a count rather than a Boolean flag is that an NCT-end may arrive on our node and be immediately sent away again several times in a row (via different networked channels), so that several consecutive dummy release signals might arrive in the CTB-handler.

Admittedly, the chance of the dummy release count ever being greater than 1 is very low, since this would require an NCT-end to arrive on our node (in an encode-handler), be sent away, return to our node via another networked channel (i.e. in another encode-handler), and be sent away a second time, *before* the first encode-handler has had the chance to internally claim the end. Since the network latency is magnitudes larger than occam-π's context-switch time, the chances of this happening are infinitesimal. Nevertheless, it may theoretically happen, so it is necessary to cater for this possibility. A dummy release count of 1 may occur more frequently, since this only requires the decode-handler to release the end before the encode-handler claims it; no network latency is involved here.

[6]During sessions, the CTB-handler accepts release signals only for ends that are currently claimed.

8.1.6 Sending an NCT-end Over Itself

One special case has not yet been discussed: sending an NCT-end over itself (or to be more specific, over one of its channels). In the user-level code, such a communication would look like this:

```
foo.cli[chan] ! foo.cli
```

For the traditional (non-networked) occam-π communication mechanism, this presents no problem, since only a pointer (to '`foo.cli`' in the above example) is communicated over a channel-word ('`foo.cli[chan]`'). It is irrelevant whether the pointer belongs to the same channel-type as the channel-word over which it is sent, or not. For networked communication, however, this case is non-trivial, and cannot be handled using the ordinary mechanisms described so far.

When a shared NCT-end is sent over a clone (cf. Section 1.5.3) of itself, no special care needs to be taken by the pony environment. This is because no internal claims and releases are involved in this case.

When an unshared NCT-end is sent over itself, however, the normal mechanisms would not work. The decode-handler would try to internally release its own CTB-handler, which in turn would try to deactivate the decode-handler — this would deadlock the NCT. Therefore, we need two more session-states in the CTB-handler:

- sending an end over itself (shorthand: 'sending eoi')
- suspended

As described in Section 7.8.1, before sending a CLC-packet to the CTB-handler that contains a channel-type-end NLC where an NCT-end is sent over itself, the decode-handler sends a special message to the CTB-handler. When the CTB-handler receives such a message, it must *suspend* the session. A suspended session is one where all decode-handlers are deactivated *except* the one dealing with the 'end-over-itself' CLC. During a suspended session, the session-state of the CTB-handler on the node

where the end is sent over itself is 'sending eoi'; the session-state of the opposite CTB-handler is 'suspended'.

Algorithm 8.3 shows how the suspension of a session is initiated by the CTB-handler. The reaction of the remote CTB-handler when it gets the 'suspend-session' message is shown in Algorithm 8.4. It can be seen that initiating a suspension is similar to initiating the closing of a session. Likewise, the algorithms reacting to the respective messages are similar.

Algorithm 8.3: Initiating the suspension of a session

```
SEQ
  ...  We just got an 'end-over-itself CLC' message from a decode-handler
  ...  Get the CLC itself from 'eoi' decode-handler
  ...  Deactivate all active decode-handlers except the 'eoi' one
  ...  Send 'suspend-session' message to remote CTB-handler
  ...  'Cancel-encode' (or get acknowledgement from) all
  ...    pending encode-handlers and pass either cancel
  ...    or acknowledgement on to remote CTB-handler
  -- Wait for confirmation from remote CTB-handler
  INITIAL BOOL running IS TRUE:
  WHILE running
    ctb.main.handle.svr[to.handler] ? CASE
      ...  First CLC of a new ULC for one of the channels
        ...  Discard, send cancel message to remote CTB-handler
      ...  'close-session' message (only possible if 'eoi' in first CLC)
        SEQ
          ...  Send cancel message to 'eoi' decode-handler
          ...    (this deactivates it)
          ...  Send 'close-session' message to remote CTB-handler
          ...  Change session-state to 'no session'
          running := FALSE
      ...  'suspend-session' message (only possible if 'eoi' in first CLC)
        ...  Do nothing -- leave this NCT to deadlock (as would
        ...    a non-networked channel-type in the same situation)
      ...  'suspend-session-confirm' message
        SEQ
          ...  Change session-state to 'sending eoi'
          ...  Pass the CLC on to the remote CTB-handler
          running := FALSE
```

Algorithm 8.4: Reacting to a 'suspend-session' message

```
SEQ
  ...  We just received a 'suspend-session' message from remote CTB-handler
  ...  Deactivate all non-pending active decode-handlers
  ...  Wait for cancel or acknowledgement from remote CTB-handler for all
  ...    pending decode-handlers and pass the relevant message on to the
  ...    respective decode-handler. Deactivate decode-handlers to which we
  ...    passed on an acknowledgement.
  ...  Send 'suspend-session-confirm' message to remote CTB-handler
  ...  Change session-state to 'suspended'
```

If an attempt is made to send both ends of the NCT over themselves at the same time, the NCT will deadlock, because each CTB-handler will send a 'suspend-session' message to the other one. This is no problem, however, since a non-networked channel-type would deadlock as well in the same situation, so semantic transparency is preserved in this case too.

When the last CLC of the ULC that contained the 'end-over-itself' CLC has been output to the receiving user-level process, the suspended CTB-handler gets an acknowledgement from the encode-handler as usual. When this happens, the suspended CTB-handler changes its session-state to 'internal session', and the state of the end that was sent over itself to 'claimed', since that end is now claimed on the receiving node. Then a special 'eoi-done' message is sent to the NCT-handler. When the NCT-handler gets this message, it updates the queue of the respective end, so that the queue now contains the receiving node on the top (and only[7]) position. No further communication happens between the NCT-handler and the CTB-handler on the sending node.

The acknowledgement from the encode-handler is sent to the CTB-handler on the sending node as usual. When it arrives there, it is forwarded to the decode-handler. Then the decode-handler is deactivated. The session-state of the CTB-handler on the sending node is changed to 'no session'; the state of the end that was sent over itself is changed to 'released'.

[7]because the end is unshared

The last thing that needs to be done in order to fully support NCT-ends being sent over themselves is adapting Algorithms 8.1 and 8.2. If a suspended CTB-handler gets a release signal, obviously no decode-handlers are active — hence no active decode-handlers need to be deactivated. If during Algorithm 8.1 a suspended CTB-handler receives an acknowledgement from an encode-handler, this can only come from the encode-handler whose encoder has just output the last CLC of the 'end-over-itself' ULC to the user-level process. In this case, the state of the end that was sent over itself is changed to 'claimed', and an 'eoi-done' message is sent to the NCT-handler. There is no point in changing the session-state, because after the closing of the session has been completed, the session-state will be changed to 'no session' anyway — since the closing of the session was initiated because the NCT-end over which the 'end-over-itself' end was received has been released.

Algorithm 8.2 must be adapted as well. If the NCT-end on the receiving node is released, the CTB-handler on the sending node, whose session-state is 'sending-eoi', will receive a 'close-session' message from the suspended CTB-handler on the receiving node. Obviously, there is only one active decode-handler in the 'sending-eoi' CTB-handler, namely the 'end-over-itself' decode-handler. Hence, no non-pending active decode-handlers need to be deactivated. If during Algorithm 8.2 an acknowledgement arrives for the 'end-over-itself' decode-handler, the state of the end that has been sent over itself must be changed to 'released'. The session-state is changed to 'no session' at the end of Algorithm 8.2 anyway.

8.2 The CTB-manager

The CTB-manager keeps the CTB-main-, -instant- and -claim-handles for all CTB-handlers on a node. They are given to other pony components on request. Additionally, the CTB-manager stores the CTB-pointers of all networked CTBs. They are used when the CTB-manager handles requests to make a previously non-networked

CTB networked (cf. Section 7.7.2) or to allocate a new NCT-end (cf. Sections 6.4.2 and 7.7.2).

The CTB-manager maintains arrays to store the various handles and the CTB-pointers. The index in the arrays corresponds to the NCT-ID of the relevant CTB. Since not for every NCT across a pony application there is necessarily a networked CTB (and a CTB-handler) on every node, some indices of the arrays may not be used. However, as discussed in the previous chapters, a new networked CTB may be allocated on a node at any given time. Therefore, the free indices may come into use at any time as well.

The arrays used are dynamic mobile arrays. When a new CTB must be allocated on our node with an NCT-ID greater or equal to the current size of the arrays, the arrays are extended. To determine the new size of the arrays, the current size is doubled until the new size is greater than the NCT-ID of the new CTB. Then arrays of the new size are allocated and the old arrays are copied to the beginning of the new arrays.

During the shutdown of the pony kernel, the CTB-manager is responsible for internally releasing unshared NCT-ends that are on our node, and for shutting down the CTB-handlers. pony's shutdown mechanism is explained in detail in Section 10.5.

8.3 The NCT-handler

The NCT-handler resides on the master node and is responsible for the claiming and releasing of the ends of an individual NCT. It handles claim and release requests from the CTB-handlers across the pony application. The general functionality of the NCT-handler was described in Sections 8.1.3 through 8.1.6 already and will therefore not be repeated here.

The claim request queues for the client-end and the server-end are implemented as ring buffers using dynamic mobile arrays. When a queue is full, the relevant array

is doubled. This involves allocating a new array twice the size of the old one and copying the old array to the beginning of the new one.

8.4 The NCT-manager

The NCT-manager resides on the master node and keeps the NCT-handles for all NCT-handlers. They are given to link-handlers on request. Additionally, the NCT-manager stores the following information for explicitly allocated NCTs:

- NCT-name
- client-state
- server-state
- type-hash

The client- and server-states can be one of the following:

- unknown
- unshared, not yet allocated
- unshared, already allocated
- shared

All data is kept in dynamic mobile arrays. The index in the arrays corresponds to the NCT-ID. When a new NCT comes into existence, the relevant data is stored at the end of the arrays. When the arrays are full, they are doubled.

During explicit allocation (explained in detail in Section 6.4.2), this data is used to determine whether the NCT-end can be allocated or an error needs to be returned. When the NCT-manager receives an explicit allocation request, it searches for the given NCT-name in its database. If the name is not found, the data of the new NCT

is stored at the end of the arrays, and a new NCT-handler is forked off. Then the new NCT-ID is returned to the requesting node.

If the NCT-name already exists in the NCT-manager's database, the data coming with the allocation request is verified against the data in the database. If an error is encountered (see Section 3.1.1 for a detailed description of possible errors), the error is returned to the requesting node. Otherwise, the client- and server-states are updated, and the NCT-ID of the existing NCT is returned to the requesting node.

It is significantly simpler for the NCT-manager to handle implicit allocation requests coming from a decode-handler which is making a previously non-networked CTB networked. Details about this are given in Section 7.7.2. No data except the NCT-name needs to be stored for implicitly allocated NCTs. The NCT-name stored is an empty string. Since empty strings are not valid as NCT-names for pony's allocation processes, the NCT-manager cannot accidentally find an implicitly allocated NCT when searching through its database during an explicit allocation request.

During the shutdown of the pony kernel, the NCT-manager is responsible for shutting down the NCT-handlers. Please refer to Section 10.5 for a detailed description of pony's shutdown mechanism.

Chapter 9

The Network Drivers

Figure 9.1 shows pony's network drivers, which are discussed in this chapter.

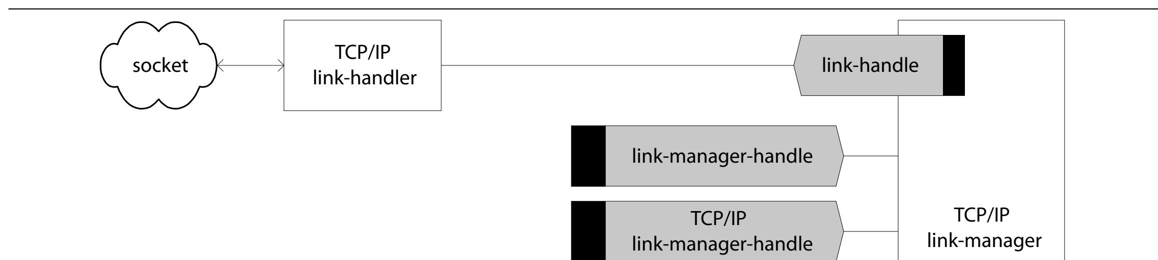

Figure 9.1: pony's network drivers

9.1 The TCP/IP Link-handler

The link-handler is responsible for the communication over network links. Links are two-way asynchronous buffered connections between two nodes; they are preserving the order of messages. All network communication between a given pair of nodes is multiplexed over the link between them. The practical implementation of links depends on the network-type used. This chapter discusses the link-handler and the link-manager for TCP/IP, which is currently the only supported network-type in pony. TCP/IP links are implemented as sockets.

Every slave of a pony application has a link to the master. This is used to communicate with the NCT-manager and the NCT-handlers. Additionally, there are links between every pair of slaves that have or had a session established between two of their CTB-handlers. This means that if between a given pair of slave nodes there was never a session established so far, there is also no link between them. Of course, there may also be sessions between the master and a slave, but as just mentioned, every slave has a link to the master anyway; that link is used for sessions as well. The creation of new links between slave nodes is discussed in Section 9.2. The link between a slave and the master is created during the startup of the slave node; this is described in Section 10.4.

The link-handler consists of two parallel sub-processes — the reader and the writer. The names 'reader' and 'writer' are to be seen from the point of view of the socket. This means that the reader reads network-messages from the socket and forwards them to the target component. The writer reads messages which pony components have sent over the link-handle, and sends them over the socket to the target node.

As discussed earlier, if a networking error occurs during the operation of the link-handler or the link-manager, the error-handler is notified about it. For the implementation of pony's error-handling mechanism, please refer to Section 10.1.

Since communication between slaves only happens during *external* sessions, there is no need for a link-handler between a slave node and itself. This is different for the master, since also components on the master node need to communicate with the NCT-manager and the NCT-handlers. Therefore, one of the link-handlers on the master node is a *dummy link-handler* which simulates a 'loop' link between the master and itself. This dummy link-handler only ever handles messages related to the master, such as claim and release requests for NCT-handlers. It never needs to handle session-related messages, since, just like on a slave node, sessions between the master and itself are always internal and therefore do not involve the pony infrastructure.

The reader is inactive in the dummy link-handler — since there is no socket to read from. The writer does not write messages to a socket, but forwards them to the target components on the master node itself. In order to preserve the semantics of links, being asynchronous and buffered, it is not possible for the writer to send the message directly via the handles to the target components, since communication over occam-π channels is synchronous and unbuffered. Instead, a special *link-outputter* is forked off by the writer, which outputs the message when the target component is ready to take it.

This gives us asynchronism and buffering. However, the other important characteristic of links, the fact that the order of messages must be preserved, has to be taken into account as well. This means that in order to simulate a link correctly, the link-outputters are not allowed to output their respective message until the 'previous' outputter has done so. This behaviour is implemented by using a chain of *link-outputter-chain-handles* between the individual link-outputters. These are channel-types containing a single channel whose client-end is held by one link-outputter, and whose server-end is held by the link-outputter that was started next.

A link-outputter waits for a signal coming in over the 'previous' chain-handle before outputting its message to the target component. When the message has been taken, the link-outputter sends a signal over the 'next' chain-handle. The very first signal is sent by a small sub-process running in parallel with the main loop of the writer. The very last signal is taken by the writer after its main loop has finished (i.e. when the link-handler is shut down).

9.2 The TCP/IP Link-manager

The link-manager keeps the link-handles for all link-handlers on the node and passes them to other pony components on request. When a session is established between two slaves for the first time, the link-manager creates a new link and forks off a new link-handler. TCP/IP socket links are created by either connecting to the remote

node, or accepting an incoming socket connection. The rule for this is that the link-manager on the node with the greater node-ID always contacts the link-manager on the node with the lesser ID. This conforms with the fact that the master (whose node-ID is 0) is being contacted by slaves (whose node-ID is greater than 0) during the startup of slave nodes. For details on pony's startup mechanism see Section 10.4.

When a link-handler writer on the master node gets a 'new-other-end' or a 'claim-confirm-otherendclaimed' message from an NCT-handler, it examines the node-ID carried by that message. If it is greater than 0 but less than the node-ID of the target node of the message, the writer contacts the link-manager and requests the network location of the node with the ID carried by the message. The request is made via the *TCP/IP link-manager-handle*, which is different from the normal link-manager-handle.

The special handle is used because the network location of a node is network-type-specific (IP address and port number in case of TCP/IP) and should therefore not be exchanged over the normal link-manager-handle. The link-manager on the master node definitely knows the location of the node with the given ID because all slave nodes in the application have contacted the master during their startup. When the writer gets the location for the given node-ID, it stores it in the outgoing network-message.

When the reader on the target node gets the 'new-other-end' or 'claim-confirm-otherendclaimed' message, it checks the node-ID that comes with it. If it is greater than 0 but less than the ID of the own node, the reader knows that the network-message contains the location of the node whose ID was carried by the message. In this case, the reader extracts the location and passes the node-ID and the location from the network-message to the link-manager via the TCP/IP link-manager-handle. The link-manager stores the location for the given node-ID because it knows that it will soon get a request (from a CTB-handler) for the relevant link-handle.

When the link-manager gets a request for a link-handle for which there is no link-handler yet[1], the link-manager compares the node-ID of the request with the ID of its own node. If the own node-ID is greater, the link-manager connects to the other node. The location of the remote node is definitely stored in the link-manager because the CTB-handler has received a 'new-other-end' or a 'claim-confirm-otherendclaimed' message beforehand — so that the link-handler has stored the location in the link-manager. When the connection to the other node has been established, the link-manager forks off a new link-handler, giving it the socket that was just created, and returns the new link-handle to the requesting CTB-handler.

If the own node-ID is less than the node-ID of the request from the CTB-handler, this means that the other node will connect to us. In this case, the link-manager allocates a new link-handle and returns the client-end to the requesting CTB-handler *without* forking off a new link-handler. Instead, the server-end of the link-handle is kept by the link-manager, together with a flag that the respective link is *pending*, which means that the remote node will contact us soon.

When the link-manager on a slave node accepts an incoming socket connection from another slave[2], a new link-handler is forked off. If the respective link was pending, no new link-handle is allocated, but the one that was previously allocated is used. With this mechanism, it is possible to answer requests from CTB-handlers for link-handles before the actual link is established. When the CTB-handler sends a message over the link-handle while the link is pending, the message is simply not taken. When the socket connection from the remote node has been accepted, the then forked off link-handler will start reading messages from the link-handle as usual.

The link-manager uses dynamic mobile arrays to store the link-handles and the data associated with a particular link. The index in the arrays corresponds to the node-ID of the remote node. On slave nodes, the same mechanism is used for the

[1]This can only be the case on slave nodes; such a request can only come from a CTB-handler.

[2]The case when the link-manager on the *master* node accepts an incoming socket connection from a slave is discussed in Section 10.4, which explains pony's startup mechanism.

arrays as in the CTB-manager. That is, some indices may be unused if there is currently no link to the respective node. When a new link is established to a node with a node-ID greater or equal to the current size of the arrays, the arrays are extended in the same way as in the CTB-manager. On the master node, the array mechanism is like in the NCT-manager. When a new slave connects to the master, the next free index at the end of the arrays is used as the node-ID of the new slave. When the arrays are full, they are doubled.

During the shutdown of a slave node, the link-manager is responsible for shutting down the link-handlers. This is explained in detail in Section 10.5, where pony's shutdown mechanism is discussed.

9.3 Optimising TCP/IP Network Performance

pony uses Barnes' socket library [Bar00a] to communicate over links. Some auxiliary processes were implemented specifically for pony; these are based on the socket library as well. The Nagle algorithm[3] [Wik06] is turned off for sockets used by pony in order to avoid delays when sending relatively small network-messages (which will be the bulk of network communication in a typical pony application).

As discussed earlier, when a ULC is sent over a networked channel, the first CLC or the 'remaining' CLCs of the ULC are sent over the link in a single network-message. This significantly improves network latency compared to previous versions of pony, where every NLC was sent and acknowledged separately. In order to be able to send network-messages that contain CLC-packets, two special processes are used by the link-handler:

[3]The Nagle algorithm aims to improve the efficiency of TCP/IP networks by reducing the number of packets that are to be sent. It coalesces several small outgoing messages and sends them all together once a certain packet size has been reached or a timeout has occurred. In pony, this is counterproductive, however, since pony awaits acknowledgments for all data messages sent, in order to preserve the handshake semantics of occam-π's channel communication. Therefore, an immediate transport of each message offers better performance in pony.

```
PROC pony.int.tcpip.socket.fullwrite.multi
            (SOCKET sock, []BYTE header,
             VAL []INT addr.array, size.array,
             RESULT INT result)

PROC pony.int.tcpip.socket.fullread.multi
            (SOCKET sock, RESULT []BYTE header,
             RESULT MOBILE []BYTE data.array,
             RESULT MOBILE []INT size.array,
             RESULT INT result)
```

For convenience, their names will be abbreviated 'fullwrite-multi' and 'fullread-multi' in the following. Both processes are based on the socket library and use Barnes' mechanism to make blocking system calls without blocking the occam-π kernel, described in [Bar00b]. The actual socket operations are done by C functions called from within 'fullwrite-multi' and 'fullread-multi'. Both 'fullwrite-multi' and 'fullread-multi' only need to call a 'blocking' C function once.

With these processes, it is possible to send and receive network-messages that consist of a fixed-sized header and any number of variably-sized items. 'fullwrite-multi' expects an address-array and a size-array; 'fullread-multi' returns a data-array and a size-array. This fits the needs of CLC-packets (cf. Section 7.8.2). In the case of CLC-packets, the items are the individual NLCs, and the CLC-descriptor if applicable.

The header of the network-message must be eight bytes (the size of two integers) larger than the actual header data, because the last eight bytes of the header are used by 'fullwrite-multi' and 'fullread-multi' internally. 'fullwrite-multi' stores the number of items in the last four bytes of the header. If there is exactly one item, its size is stored in the penultimate four bytes of the header. If the arrays are empty, 'fullwrite-multi' only sends the header. If there is exactly one item, 'fullwrite-multi' sends the header and the single item. If there is more than one item, 'fullwrite-multi'

sends the header, the size-array and all items. Everything is sent in one go using the '`writev`' function. No copying of the items that are sent is necessary, which improves performance.

'fullread-multi' first reads the header and extracts the number of items. If there are no items, empty arrays are returned. If there is exactly one item, 'fullread-multi' extracts the item's size from the header and reads the item. Then 'fullread-multi' returns the item itself in the data-array, and for the size-array an array with the item's size as the only element. If there is more than one item, 'fullread-multi' reads the size-array (whose size in bytes is four times the number of items). Then it sums up all sizes in the size-array and reads the corresponding number of bytes into the data-array. Using this mechanism, fixed-sized network-messages (most non-CLC-related ones) only require one socket read. Network-messages with only one item in the arrays require two socket reads. (The bulk of CLC-packets in a typical pony application will only contain a single item — either a data-item NLC or a CLC-descriptor.) Network-messages with more than one item in the arrays require three socket reads.

'fullwrite-multi' and 'fullread-multi' are also used by the ANS, the link-manager and the link-handler to allow strings of variable length as part of network-messages. This is used for application-names and NCT-names. In this case, the address-array and the size-array for 'fullwrite-multi' only contain one element each. Hence, the data-array returned by 'fullread-multi' contains the relevant name only.

Chapter 10

Other Implementation Issues

10.1 Implementation of Error-handling

Figure 10.1 shows the error-handler with the internal and the external error-handle.

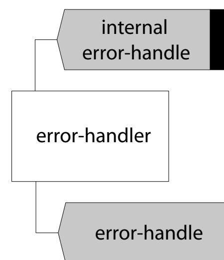

Figure 10.1: The error-handler

If an error-handler has been started on a node (cf. Section 2.3), the link-handler and the link-manager use the *internal error-handle* to report networking errors occurring during their operation. For the storage of errors, the error-handler maintains a ring buffer which is doubled in size whenever it gets too small.

Except the '`pony.err.get.nct.id.*`' processes, all other error-handling processes (cf. Section 4.1) are implemented as a simple request/reply sequence over the (external) error-handle. The error-handling process makes a request to the error-handler, carrying the relevant data from its parameters, and waits for a reply. During its main loop, the error-handler `ALT`s over the internal and the external error-handle.

When a request arrives from one of the error-handling processes over the external error-handle, the error-handler sends a reply to the requesting process. The data coming with the reply is then returned by the error-handling process to the user-level code via the relevant parameters. The only error-handling process that gets no reply from the error-handler is the '`pony.err.shutdown`' process, which just sends a shutdown signal to the error-handler (see Section 10.5 for details about pony's shutdown mechanism).

The '`pony.err.get.nct.id.*`' processes are a special case, since they do not communicate with the error-handler. Instead, they first check whether the channel-type-end parameter is defined or not. If the end is undefined, an error is returned. Otherwise, a small inline assembler routine determines the CTB-pointer of the end. Then an auxiliary C function is called to find out the NCT-ID stored in the CTB. If the CTB is not networked, the auxiliary function returns an error, otherwise it returns the NCT-ID. The '`pony.err.get.nct.id.*`' processes terminate after the auxiliary function has finished, returning the NCT-ID and the result via the relevant parameters.

When the ID of the current remote node for a given NCT-ID is requested over the external error-handle, the error-handler contacts the CTB-manager and requests the CTB-instant-handle for the relevant CTB-handler. If the reply from the CTB-manager says that the NCT-ID is invalid, the error-handler notifies the requesting process about it. Otherwise, the error-handler now requests the ID of the current remote node from the CTB-handler's instant-handler. The reply from the instant-handler (which may either be the requested node-ID, or a notification that there is currently no external session) is then forwarded to the requesting process.

The error-handler associates a unique running number (called the 'position') with each error that gets stored in the buffer. This number is not stored in the array together with the error; instead, the error-handler keeps two special variables. These variables store the position of the first error that is currently stored in the buffer (called 'first running number'), and the position of the next error that will happen

(called 'next running number'). When a new error-point is requested from the error-handler, the 'next running number' will be returned. The error-handler keeps another variable, which stores the number of *pending* error-points at the current 'next running number'. Every time an error-point is requested, this number is increased.

When an error is reported over the internal error-handle, the error-handler discards the error if the buffer is empty *and* the number of pending error-points is zero. Otherwise the error, together with the number of pending error-points[1], is stored in the buffer. Then the 'next running number' is increased and the number of pending error-points is set to zero. In this way, the error-handler knows for every error that is stored in the buffer how many error-points were given to user-level processes while the position associated with the error was the 'next running number'.

When the deletion of an error-point is requested over the external error-handle, the error-handler deletes the error-point as set out in Algorithm 10.1.

When the error-handler receives a request for all errors after a given error-point over the external error-handle, it does a similar check as in Algorithm 10.1. If the error-point is at the 'pending' position and the number of pending error-points is greater than zero, an OK and an empty array are returned, together with the unchanged error-point. If the error-point is out of range, or if there are no error-points stored in the buffer at the given position, the requesting process is notified that the error-point is invalid.

Otherwise, the error-handler now goes through the buffer, starting with the error at the position of the given error-point, and assembles an array with all errors that fit the criteria which came with the request. Please note that the assembled array may be empty if none of the errors that happened after the given error-point fits the criteria. Then the error-point at the given position is deleted in the same way as in Algorithm 10.1. This involves deleting errors without an error-point from the start of the buffer. When this is done, the number of pending error-points is increased.

[1] which may be zero if there are errors in the buffer already

Algorithm 10.1: Deleting an error-point

```
SEQ
  ...  We just received a request to delete an error-point
  ...    (‘err.point’ is the error-point to be deleted)
  ...  Set ‘rel.err.point’ to the index of the error-point in the buffer
  IF
    -- Error-point at ‘pending’ position
    (err.point = next.running.num) AND (pending.err.points > 0)
      SEQ
        pending.err.points := pending.err.points - 1
        ...  Return OK
    -- Error-point out of range
    (err.point < first.running.num) OR (err.point >= next.running.num)
      ...  Return ‘invalid error-point’ error
    -- No error-point at given position
    err.data.array[rel.err.point][num.err.points] = 0
      ...  Return ‘invalid error-point’ error
    TRUE
      SEQ
        -- Delete error-point at given position
        err.data.array[rel.err.point][num.err.points] :=
          err.data.array[rel.err.point][num.err.points] - 1
        IF
          ...  Deleted error-point was at start of buffer and
          ...      was the only error-point stored at this position
            ...  Delete all errors with no error-point from start of buffer
            ...    (afterwards buffer is either empty, or new first element
            ...    of the buffer has a stored ‘num.err.points’ > 0)
          TRUE
            SKIP
        ...  Return OK
```

Finally, the error-handler returns an OK to the requesting process, together with the assembled array, and the current ‘next running number’ as the updated error-point.

10.2 Implementation of Message-handling

Figure 10.2 shows the pony components that are related to message-handling.

If a message-handler is active on a node, the pony components use the *internal message-handle* to report status messages to the message-handler. The link-handler

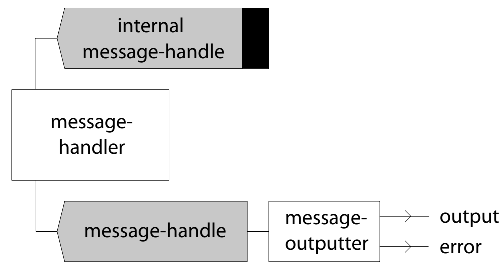

Figure 10.2: pony components related to message-handling

and the link-manager may report error messages as well. Which messages are reported depends on the message-type given to startup process (cf. Section 2.3).

As described in Section 4.2, there are different message-outputters using various combinations of output and error channels. The correct message-outputter should be chosen according to the message-type used. A message-outputter with an output channel should be chosen if and only if status messages are reported by the pony kernel; the same applies to an error channel and error messages accordingly. Nevertheless, it *is* possible to use any combination of message-type and message-outputter. If there are misfits between the channels and the message-type, warnings are displayed by the message-outputter.

When the message-outputter is started, it requests the ID of its own node and the message-type from the message-handler. If the message-outputter has a channel for a type of message that is not reported by the pony kernel, it displays a warning. The same applies if a type of message is reported by the pony kernel for which the message-outputter has no channel. In the latter case, the corresponding messages are discarded by the message-outputter. The own node-ID is displayed when messages are output.

Then the message-outputter starts its main loop, which consists of an auto-prompter requesting the next message from the message-handler. Whenever a message arrives, it is output to the correct channel (unless it needs to be discarded; see above). When the message-outputter receives a shutdown signal from the message-handler, it leaves its main loop and terminates.

The message-handler maintains a ring buffer to store messages from the pony components until they have been taken by the message-outputters. The size of the buffer is doubled whenever it gets too small. During its main loop, the message-handler `ALT`s over the internal and the external message-handle. Each time a new message arrives over the internal message-handle, it is stored in the buffer. When a shutdown signal arrives over the internal message-handle, the message-handler sets a special flag to store this fact.

Requests over the external message-handle are only accepted by the message-handler if the message-outputter has not requested initialisation yet, if there is at least one message in the buffer, or if a shutdown signal was received from the pony kernel earlier. If the message-outputter requests initialisation, the message-handler replies with the own node-ID and the message-type. If the message-outputter requests a message, the message-handler returns either a message from the buffer if the buffer is not empty; otherwise it sends a shutdown signal to the message-outputter.

10.3 The Application Name Server

The ANS is not part of the pony library as such, but a stand-alone occam-π program. It displays status and error messages on the screen and allows the user to terminate the program via the keyboard. The ANS for TCP/IP uses Barnes' socket, file and process libraries [Bar00a] for calling various routines that are needed for its functionality. When the ANS is started, it first tries to find its ANS-configuration file (cf. Section 3.3.3) and to extract the port number from it. If the file is not found or if it contains no port number, the default ANS port number is used. Then the ANS creates a listening socket and starts its main loop. During the main loop, it accepts incoming connections from master or slave nodes of pony applications.

The ANS keeps a buffer of all applications that are currently active or pending. A pending application is one where one or more slave nodes have contacted the ANS using 'slave/wait' as their node-type (cf. Section 2.3.2) and are currently waiting for

the master to contact the ANS. When an application is shut down, the index of the application in the buffer is put on a free-list so that the index can be re-used. If the buffer is full, it gets doubled in size. In order to avoid mix-ups between nodes of old and new applications using the same index, each application is assigned a unique code, which is implemented as a running number. For each application, the ANS also keeps a count of how many slave nodes have requested the master's network location so far. This information is needed during the shutdown of applications, cf. Section 10.5.

When the ANS is contacted by a node, it looks up its database for an application with the application-name that came with the request. If the application-name is new and the node-type is 'master' or 'master/reset', the new application is added to the database and the location of the master is stored. Then the ANS sends a reply to the master node, containing the application-index and the application-code. If the node-type is 'slave', an error is returned because there is no master yet. If the node-type is 'slave/wait', a new pending application is added to the database. For pending applications, the ANS keeps a buffer of slaves that are currently waiting for the master.

If an active application with the given application-name already exists and the node-type is 'master', an error is returned. If the node-type is 'master/reset', the existing application is removed from the database and the new application (getting a new application-code) is stored instead. Then the new master node is notified about the application-index and the application-code. If the node-type is 'slave' or 'slave/wait', the location of the master is returned to the slave node by the ANS.

If a pending application with the given application-name already exists and the node-type is 'master' or 'master/reset', the state of the application is changed to 'active' and all waiting slaves are notified by the ANS about the location of the newly arrived master. Then the master is notified about the application-index and the application-code. If the node-type is 'slave', an error is returned because there is

no master yet for the application. If the node-type is 'slave/wait', the slave node is added to the buffer of waiting slaves.

When the ANS gets a shutdown request from the master of an application, this request will contain the application-index and the application-code. The ANS checks whether the application-code is correct and sends a reply to the master node, containing the number of slaves that have requested the location of the master so far. Then the application is removed from the buffer; the application-index is put on the free-list. If the application-code is wrong, an error is returned to the requesting master node, and the application is not removed from the database.

During its operation, the ANS displays status messages about incoming requests and how they are handled. After each request, the current number of active and pending applications is displayed. If networking errors happen during the handling of requests, the ANS closes the current socket and displays a warning that the state of the database might be inconsistent. If the listening socket fails, the ANS terminates.

The ANS can be shut down by pressing 'q'. If there are currently active or pending applications stored in the ANS, a warning is displayed and the shutdown must be confirmed by pressing 'q' a second time.

10.4 The Startup Mechanism

As for the ANS, KRoC's socket, file and process libraries are also used by the pony library itself for the startup of a pony node. When the startup process (cf. Section 2.3) is called, it starts a special process whose task it is to connect to the ANS, set up the link to the master if necessary, and start the link-manager. Since this process is network-type-specific, it is kept separate from the rest of the startup mechanism.

The special process for TCP/IP tries to find the ANS-file and the node-file, and to determine the network location of the ANS and of the own node. This is done as set out in Section 3.3. If the settings are not found, defaults are used where applicable. If the lookup fails, an error is returned. Otherwise, the special process now creates

the listening socket for the node. If this is successful, the ANS is contacted. The message to the ANS contains the node-type, the application-name and the location of the node.

As discussed in Section 10.3, upon success, the ANS will return the application-index and the application-code if the node is the master, or the location of the master if the node is a slave. If the node-type is 'slave/wait', the reply from the ANS may tell the slave to wait because there is no master yet. In this case, the slave will close the connection to the ANS and accept a new incoming connection from the ANS — which will arrive as soon as a master for the application has contacted the ANS. When the connection from the ANS has been accepted, the slave node is notified by the ANS about the location of the new master. Then the connection with the ANS is closed.

If the node is a slave, it will now connect to the master and notify the master about its own location. The link-manager on the master node accepts the incoming connection. If the master is in the process of shutting down, it will notify the slave about it. Otherwise, it checks whether the location sent by the slave is used by another slave already, in which case an error is returned to the slave, so that the slave's startup would fail. Otherwise, the master notifies the slave about its node-ID (which is the next free node-ID in the application), and forks off a link-handler for the new link.

If the reply from the master was that the master is currently shutting down, and the node-type is 'slave', the slave's startup would fail. If the node-type is 'slave/wait', the special process will start over and connect to the ANS again — and (if necessary) wait for the next master that will start an application of the given name.

If the above sequence of events has been completed successfully, the special process forks off the link-manager. The link-manager starts its acceptor sub-process, which starts accepting incoming connections. Then the link-manager forks off the master-link-handler. If our node is a slave, the master-link-handler uses the socket which was just established to the master. If our node is the master, obviously, no such socket

exists. Instead, the master-link-handler will be a dummy (cf. Section 9.1). Then the link-manager starts its main loop.

If the special process has finished without errors, the startup process can now fork off the main kernel. The main kernel then forks off its subcomponents. First the message-handler is forked off if the startup process is meant to return a message-handle, then the error-handler is forked off if the startup process is meant to return an error-handle. Then the main kernel requests the link-handle to the master-link-handler from the link-manager. Now the CTB-manager is forked off. On the master node, the last component to be forked off is the NCT-manager. Then the main kernel starts its main loop.

10.5 The Shutdown Mechanism

When pony's shutdown process (cf. Section 3.2) is called, it sends a shutdown request to the main kernel via the network-handle. The main kernel then shuts down all its internal components in the following order:

- CTB-manager and CTB-handlers
- link-manager and link-handlers
- On the master node: NCT-manager and NCT-handlers
- If applicable: error-handler and message-handler

CTB-manager and CTB-handlers

The main kernel first sends a shutdown signal to the CTB-manager via the CTB-manager-handle. In the CTB-manager, a small sub-process then sends internal release signals to both CTB-claim-handles of all CTB-handlers that are currently known by the CTB-manager, in order to release unshared NCT-ends that are located on our

node. All those release signals are sent in parallel. While they are being processed, the CTB-manager continues accepting and handling incoming requests normally.

For the sake of simplicity, release signals to the CTB-handlers are sent for all ends, no matter whether the ends are unshared or shared[2], and whether they are currently claimed or not. Ends that are shared, as well as unshared ends that are currently not on our node, are not in a claimed state in their respective CTB-handlers when the release signals are sent. As discussed in Section 8.1.5, however, this presents no problem, since CTB-handlers accept release signals for ends that are already released when the session-state of the CTB-handler is 'no session'. This is the reason for sending the release signals in parallel. If there is currently a session in the CTB-handler, this means that at least one end is currently claimed. The internal release signal for a claimed end will be accepted in any case. When the end has been released, there is definitely no more session in the CTB-handler, so the internal release signal for the other end will then be accepted as well, even if it is already released.

When all ends have been released, the CTB-manager fills an array with the CTB-pointers of all networked CTBs and shuts down all CTB-handlers on the node. Please note that the number of CTB-handlers that are on our node now may be higher than the number of CTB-handlers whose ends were just internally released. The reason for this is that between the shutdown signal from the main kernel, which initiated the internal releasing of all ends, and the completion of the internal releasing of these ends, channel-type-end NLCs may still have been arriving in encoders on our node. Although none of these NCT-ends were output to user-level processes, new CTBs for them may still have been established before the pending CLCs were cancelled due to the internal releasing of all ends. This was the reason for still letting the CTB-manager handle requests as usual — otherwise a deadlock may have occurred between the CTB-manager trying to release an end, the relevant CTB-handler trying to 'cancel-encode' a pending outgoing CLC, and the encoder dealing with that CLC

[2]In fact, the release signal during shutdown is the only case where the CTB-claim-handle is used for shared ends.

trying to contact the CTB-manager in order to allocate a new NCT-end for a pending channel-type-end NLC.

Even though there may be new CTB-handlers now, there is no need to internally release *their* ends, since those ends were not internally claimed before — because the relevant channel-type-end NLCs were never output to user-level processes. The CTB-manager now sends a shutdown signal to all CTB-handlers that are currently on our node. When the CTB-handlers are shut down, they shut down all their decode- and encode-handlers, which in turn shut down their respective decoders and encoders. When all this is done, the CTB-manager sends a shutdown confirmation to the main kernel via the kernel-reply-handle, together with the array of CTB-pointers. Then the CTB-manager terminates.

Link-manager and Link-handlers

Now the main kernel sends a shutdown signal to the link-manager. The link-manager reacts differently to the shutdown signal depending whether our node is the master or a slave. If our node is a slave, the link-manager will send a shutdown signal to all its link-handlers. This is safe to do now, since once the CTB-handlers have all been shut down, it is not possible anymore for link-handlers to contact the link-manager[3] — therefore there is no danger of deadlock here.

When a link-handler on a slave node gets a shutdown signal from the link-manager, it checks whether the link still exists. If it does, the link-handler sends a shutdown message over the link to the remote link-handler. Then it waits for a shutdown message from the remote link-handler. When that arrives, the link-handler closes the socket, sends a shutdown confirmation to the link-manager over the link-manager-handle, and terminates.

[3]The only case where link-handlers on slave nodes may contact the link-manager is in order to store the network location of a node in the link-manager, cf. Section 9.2.

When a link-handler on a slave node gets a shutdown message from the remote link-handler *before* getting the shutdown signal from the link-manager, it sends a shutdown message back to the remote link-handler and closes the socket. Then it waits for the shutdown signal from the link-manager. When that arrives, the link-handler sends a shutdown confirmation to the link-manager over the link-manager-handle, and terminates.

On the master node, this is different. The link-manager there does not send a shutdown signal to its link-handlers, but waits for them to be shut down by a shutdown message from the respective slaves. Therefore, link-handlers on the master node never initiate the shutdown of a link, but only react to a shutdown message from the remote link-handler. If this arrives, a shutdown message is returned to the remote link-handler and the socket is closed. The shutdown confirmation is then sent immediately to the link-manager, and the link-handler is shut down. This means that the link-manager on the master node can get shutdown confirmations from its link-handlers at any given time, whereas on slave nodes, the link-manager would need to send the shutdown signal to the link-handlers first.

The reason for implementing the shutdown in this way is that, as pointed out in Section 3.2, on slave nodes, the pony kernel is meant to terminate immediately when the pony shutdown process is called, whereas on the master node, the pony kernel continues serving the slaves (which may still be using the NCT-manager and the NCT-handlers on the master node) until all slaves have been shut down themselves.

On slave nodes, the link-manager shuts down its acceptor immediately when it receives the shutdown signal from the main kernel. When the link-manager on a slave node has received the shutdown confirmations from all its link-handlers, it sends a shutdown confirmation to the main kernel via the kernel-reply-handle, and terminates itself.

When the link-manager on the master node gets the shutdown signal from the main kernel, it contacts the ANS to inform it about the shutdown of the application. As soon as the master has notified the ANS about the shutdown, no more requests

from slave nodes will be accepted by the ANS for this application. However, some slaves may have got the location of the master from the ANS before the ANS was notified about the shutdown, but not connected to the master yet. Therefore, when the ANS gets the shutdown notification from the master, it returns the number of slaves that have requested the location of the master so far. The link-manager on the master node keeps a count of how many slaves have connected to it so far. If the two counts are not equal, the link-manager will not shut down its acceptor until the remaining slaves have connected to it. When this happens, the slave nodes are notified that the master is in the process of shutting down, and the accepted connections from the respective slaves are closed immediately.

The link-manager on the master node waits until *both* all remaining slaves have connected to it (whereupon the acceptor is shut down) *and* the shutdown confirmations from all link-handlers have arrived. Once both conditions are met, it sends a shutdown confirmation to the main kernel via the kernel-reply-handle, and terminates itself.

NCT-manager and NCT-handlers

If our node is the master, the main kernel now sends a shutdown signal to the NCT-manager — which we know will not be used anymore since all links to other nodes are gone. The NCT-manager sends a shutdown signal to all NCT-handlers, which will confirm the shutdown and terminate straight away. Then the NCT-manager confirms its own shutdown to the main kernel, and terminates itself.

Terminating the Shutdown Process

Now the main kernel sends a reply to the shutdown process, containing the array of CTB-pointers it had received from the CTB-manager earlier. The shutdown process then calls an auxiliary C function which will shut down all previously networked

CTBs. This mechanism was described in detail in Section 6.3.3. When the auxiliary function has finished, the shutdown process terminates.

Error-handler and Message-handler

For the main kernel, there is now only one thing left to do, namely notifying the error-handler and the message-handler about the shutdown — if they are running on our node. If the error-handler is running on our node, the main kernel sends a shutdown signal to the error-handler via the internal error-handle. In this case, the error-handler is responsible for sending the shutdown signal to the message-handler via the internal message-handle (if there is a message-handler on our node) when the error-handler has finished itself. If there is no error-handler on our node, but there is a message-handler on our node, the main kernel itself sends the shutdown signal to the message-handler via the internal message-handle. When all this is done, the main kernel leaves its main loop and terminates.

The error-handler only terminates after having received *both* a shutdown signal via the internal error-handle *and* one via the external error-handle. When getting one of the two shutdown signals, the error-handler stores this fact in a special flag. After having received the internal shutdown signal, it will stop taking requests from the internal error-handle; after having received the external shutdown signal, it will stop taking requests from the external error-handle. As soon as the second shutdown signal arrives, the error-handler sends a shutdown signal to the message-handler (if there is one on our node), and terminates.

When the message-handler gets the shutdown signal via the internal message-handle, it stores this fact in a special flag. After having received the shutdown signal, the message-handler will stop taking requests from the internal message-handle. When all remaining messages (including the shutdown signal at the very end) have been taken by the message-outputter, the message-handler terminates.

Part III

Performance of pony, Evaluation and Conclusions

This part of the thesis evaluates pony's performance and contains the final conclusions. Chapter 11 presents a number of tests that were carried out to examine the performance of the pony environment. Chapter 12 concludes with a discussion of the work presented in this thesis, along with an outline of possible future research.

Chapter 11

Performance Evaluation

To examine the performance of the pony environment, we ran a number of tests, the results of which are presented in this chapter. The performance tests were carried out jointly with Adam Sampson from the University of Kent and Dyke Stiles from Utah State University [Uta06].

The author particularly wishes to thank Adam Sampson for implementing the '`bmthroughput`' benchmark program, as well as the '`mandelbauer`' demo and the distributed pony version of the classical `occam` '`commstime`' benchmark [SS06]. All these programs, together with the '`bmpingpongtime`' program also presented in this chapter, are included in the KRoC distribution. Thanks also go to Dyke Stiles for implementing a pony version of his Distributed Robust Annealing package [Sti05].

11.1 Basic Considerations

All tests were run on the TUNA cluster at the University of Kent. This cluster was funded by EPSRC as part of the TUNA project [SWP+05], a joint project of the University of Kent and other universities. The cluster consists of 30 PCs with 3.2 GHz Intel Pentium IV processors, running Linux 2.6.8, linked by a reliable switched gigabit Ethernet network.

The test programs, as well as the pony library itself, were compiled using KRoC's highest optimisation options. Some random checks, however, showed that the compiler optimisation had no measurable effect on the test results.

For the duration of the tests, no other tasks were performed on the machines used. Memory usage was watched carefully to avoid going into swap. Each pony node was run on a dedicated host, unless stated otherwise (see Section 11.7). The ANS was also given a dedicated host. This was done primarily for ease of management, since the ANS is not performance-critical.

All tests (except the annealing tests, which are a special case; see Section 11.7) aim to be 'steady-state' measurements. The loops are started and allowed to run for at least two seconds before the timer is started, in order to avoid CPU caching effects; the performance of the loop is then measured over a period of ten seconds.

Each measurement was repeated three times (five times for the annealing tests) and the average of the results taken. We have omitted error bars for clarity; the error was within 1% on all tests.

It is worth noting that, to date, the main concern in the implementation of pony was its proper function according to the objectives set out in Section 1.3. Only little time has been invested in performance optimisation so far. Still, the results of our tests are very encouraging, and should only improve when pony's performance is further optimised in the future. We have already built several distributed applications using pony which perform well on PC clusters.

11.2 Communication Time

'`commstime`' is a standard benchmark that has traditionally been used with various incarnations of `occam` and similar CSP-based platforms. Its process layout is shown in Figure 11.1.

The '`commstime`' benchmark consists of four parallel processes, three of which are running in a loop. The processes are connected by channels carrying `INT`s. The

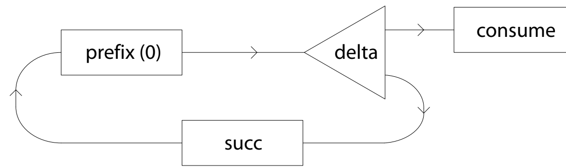

Figure 11.1: The 'commstime' benchmark

'prefix' process first outputs a pre-defined number. Then it inputs incoming INTs and passes them on. The 'delta' process inputs INTs and passes them on via two output channels. The 'succ' process inputs INTs and outputs their successors. Finally, the 'consume' process inputs the INTs from the above circuit and acts as a monitoring process. Since the processes are effectively only doing communications, the cycle rate of the network (i.e. how long it takes for a piece of data to travel around the loop) can be used to estimate the overhead of a single communication.[1]

The pony version of the 'commstime' benchmark modifies the standard program so that each of the four processes runs on a separate node. The channels between processes become NCTs containing a single INT channel. Appendix C compares the traditional and the distributed implementation of 'commstime' in order to give a practical example of how to use pony to make an existing application distributed.

Since pony sends INTs as single NLCs, the communication time measured is the time for a basic *network* communication — which includes not just several occam-π context-switches, but also eight pthreads context-switches, four system calls into the kernel, and two TCP round-trips across the network.

The standard 'commstime' was compiled using the same KRoC version and options as the other test programs, and reported a communication time of 19 ns with CPU usage at 100%. The pony 'commstime' reported a communication time of 66 μs with CPU usage on each node at 3% — approximately fifteen thousand communications per second.

[1] There is also a 'parallel delta' version of the original benchmark which is used to measure process startup time; the benchmark used here is the 'sequential delta' version in which no processes are created or destroyed while the benchmark is running.

These results met our expectations. They are also in line with the throughput measurements below. The measurement in Figure 11.3 gave a throughput of 21.6 KB/s[2] for a single node running 50 worker processes each outputting messages of 1 B size, i.e. about 21.6 thousand messages per second arriving at the measuring process. As expected, the fact that 50 workers were running in parallel moderately increased the communication rate compared to the fifteen thousand communications per second measured in the '`commstime`' benchmark above.

11.3 Throughput

'`bmthroughput`' is a benchmark program intended to measure the aggregate data rate available across a group of networked channels. We will refer to that rate as 'throughput' in the following. The '`bmthroughput`' program is designed to be a flexible tool for throughput measurement, allowing variations of several parameters of the distributed benchmark application.

A collection of worker processes — distributed across a number of slave nodes — sends '`MOBILE []BYTE`' arrays to a master process (on the master node); the master measures the rate at which it is receiving data from the collection of workers. The number of slave nodes, number of workers per slave node, range of message sizes (fixed or randomly distributed) and transmission rate (in messages per second, or simply 'as fast as possible') can be varied. In this set of benchmarks, the code generating messages is trivial, and there are no other occam-π processes running to compete with pony for CPU time.

Using 100 KB — a message size typical for applications rendering real-time graphics — the saturation point of the network can be reached with relatively few sending processes. Figure 11.2 shows the throughput available with one to 25 slave nodes, each running two workers; network saturation is just reached at 25 slave nodes (i.e.

[2]All byte prefixes used in this chapter are decimal, e.g. 1 KB = 1000 B.

50 workers). The peak at two slave nodes was persistent through all measurements; we have no explanation for it at the present stage.

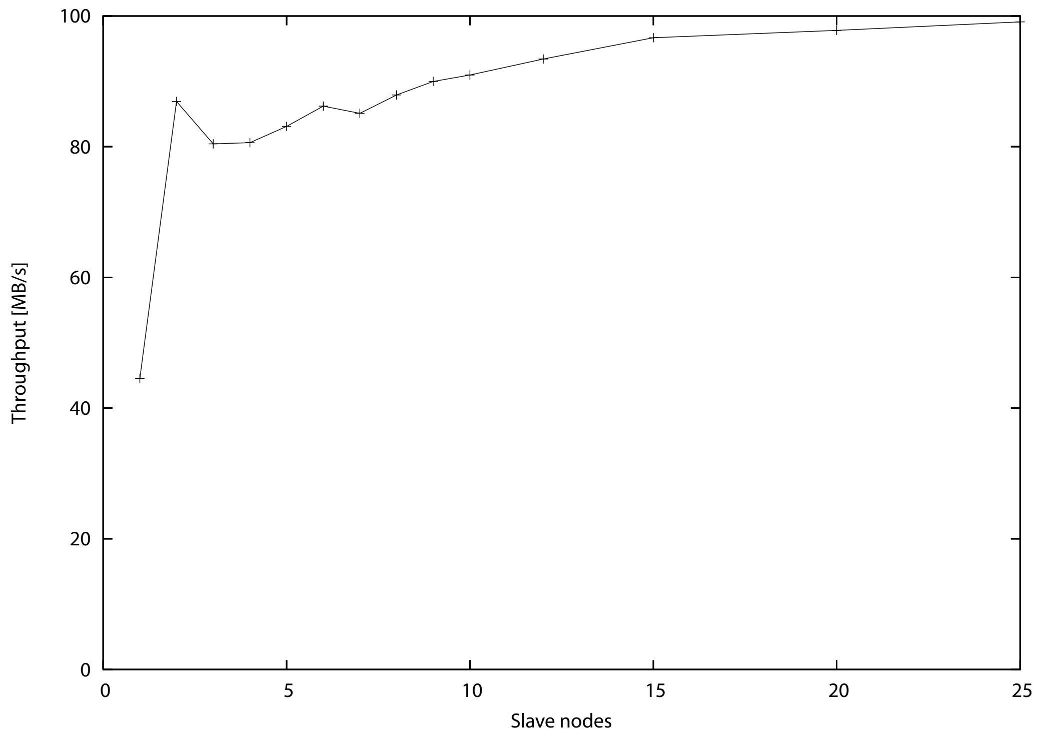

Figure 11.2: Throughput: 100 KB messages, two workers per slave

Figure 11.3 shows the throughput available from one slave node running 50 workers as the message size is varied between 1 B and 1 MB. Since pony (like occam-π internally) does approximately the same amount of work per communication regardless of the size of the message, there is an obvious advantage in using larger messages if your application is optimised for throughput.

Figure 11.4 shows the throughput available from one slave node using 50 KB messages as the number of workers is varied between 1 and 500. pony uses blocking system calls, so other occam-π processes can execute while pony is waiting for network operations to complete; throughput-sensitive applications should therefore use multiple processes per node, or have internal buffering, to ensure that the networked channels always have data available to send.

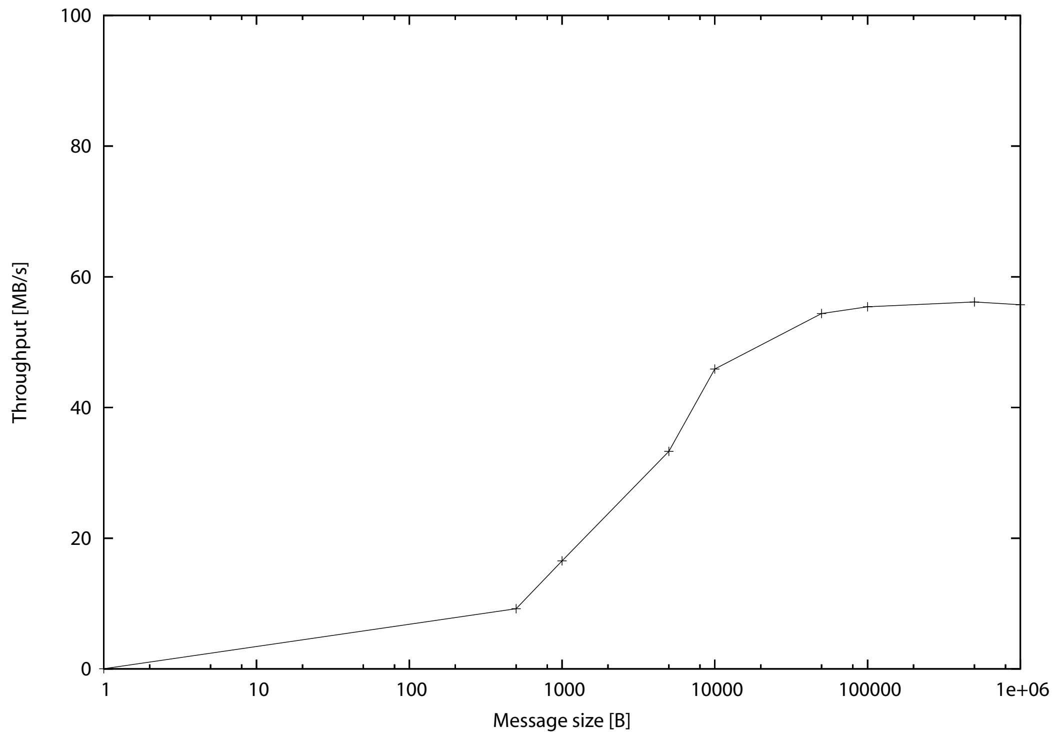

Figure 11.3: Throughput: Varying message size, one slave with 50 workers

11.4 Network Overhead

An analysis of the results of 'bmthroughput' gives an idea of the network overhead created by the pony environment. The throughput measured by the master process only includes the data actually being sent by the application (that is, the 'MOBILE []BYTE' arrays); the network overhead due to the pony and TCP/IP protocols is not included. It can be estimated by comparing the measurement with the network data rate reported by the operating system.

The rightmost data point in figure 11.2 is 99.1 MB/s. At the same point, the network usage measured (using the Linux 'saidar' tool) was 104.9 MB/s. The network overhead was thus approximately 5.8%, or 5.8 KB for every 100 KB array of data. Since each 1.5 KB Ethernet frame will contain approximately 60 B of Ethernet, IP and TCP headers, the network overhead can be split up into some 4% which are due

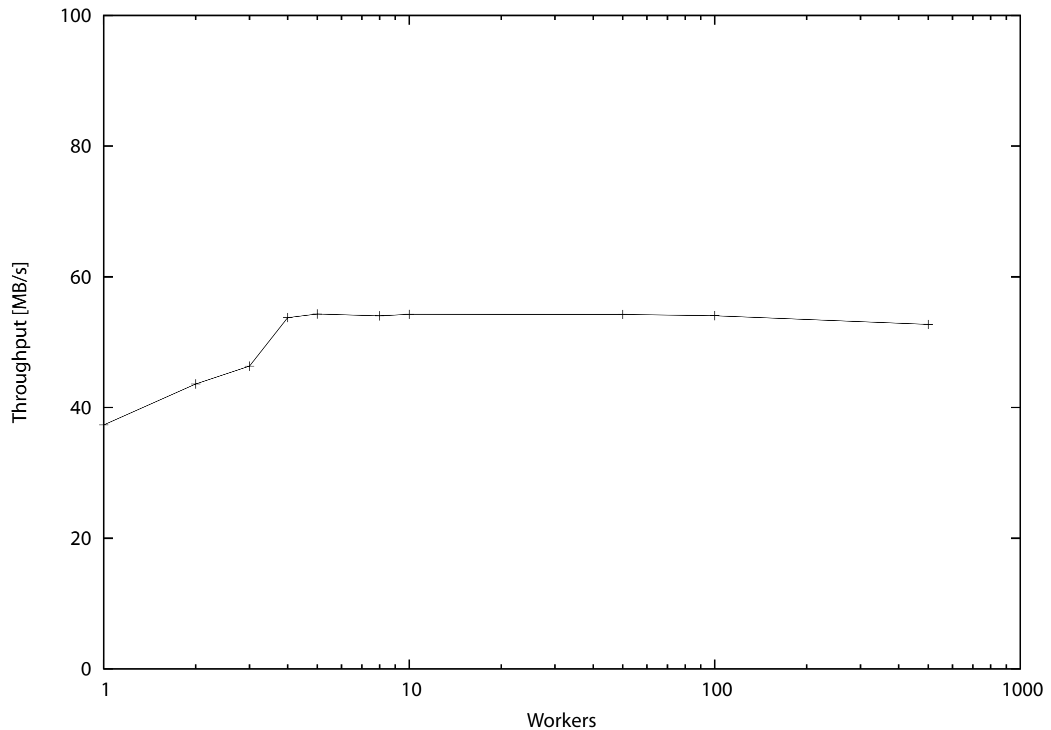

Figure 11.4: Throughput: 50 KB messages, one slave, varying number of workers

to the network protocols in use, and 1.8% due to pony itself. This shows the low impact the pony protocol has on the network traffic.

11.5 CPU Overhead

The computational overhead introduced by the pony environment can be evaluated by measuring the CPU time per ULC. This is the time between starting to send a ULC via a networked channel and receiving the acknowledgement for the ULC's last CLC (at which time the user-level process is released from the extended rendezvous), specifically excluding the network latency from this measurement. The time measured reflects the CPU overhead on the sending node.

The '`bmpingpongtime`' benchmark measures the time needed by the CLCs of a ULC to travel through the protocol-decoder, into the pony kernel, and then all the way through the pony kernel until the point where they would have to be output to

the network. At this point, nothing is sent to the network, but the acknowledgement for the relevant CLC is sent to the CTB-handler as if it had just been received from the network. The measurement includes the ping-pong times for all CLCs of the ULC. After the last acknowledgement has been returned, the sending operation finishes as usual, with the decoder assuming that the remote node has received the data, and therefore releasing the user-level channel.

'`bmpingpongtime`' sends regular byte arrays in order to exclude any dynamic memory allocation (for instance of '`MOBILE []BYTE`' arrays) from the figures. Figure 11.5 shows the CPU time per ULC for single byte arrays of varying size. As expected, the CPU time is fairly constant. This is because the pony infrastructure does not copy the user-level data, but uses its address and size (cf. Chapter 7).

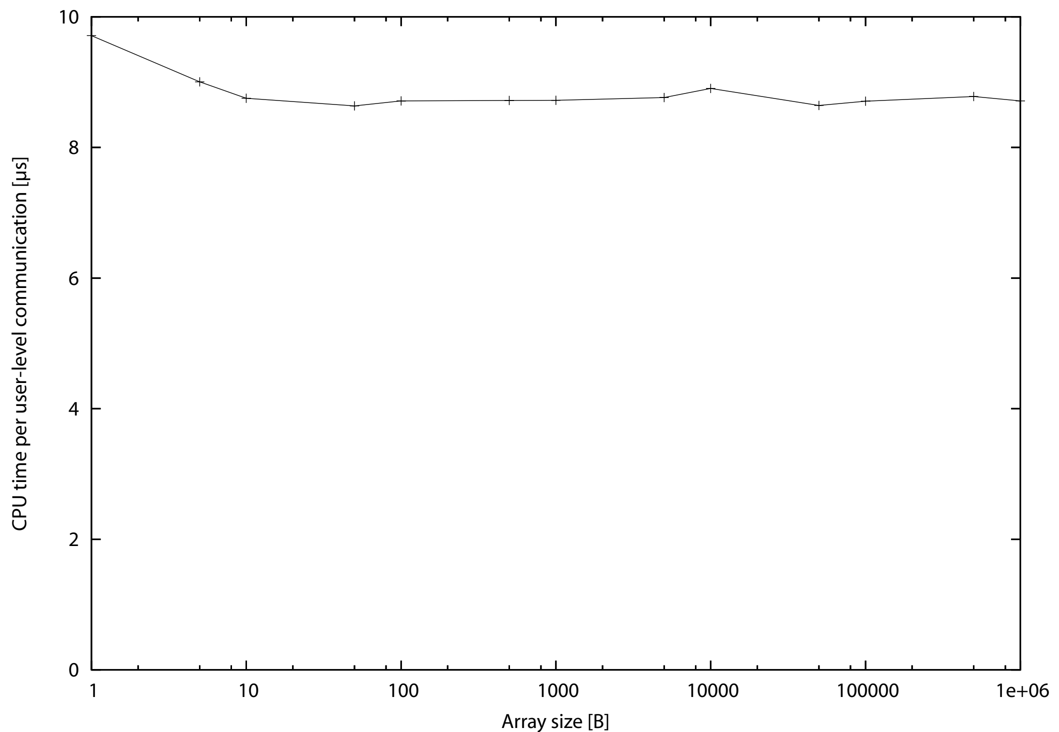

Figure 11.5: CPU overhead: Single byte array of varying size

An interesting phenomenon is that sending one byte of regular data is slower than sending 10 or 1000. An analysis of the bytecode generated by the compiler shows

that KRoC uses the '`OUT8`' instruction for the single byte and '`OUT`' for the rest, so presumably those have different performance characteristics.

Another test measures the CPU time per ULC for sequential protocols with a varying number of items. In each sequential protocol, all items are regular byte arrays of the same size; we have carried out measurements for array sizes of 1 B, 1 KB and 1 MB.

Figure 11.6 shows the CPU time per ULC for sequential protocols of 1 B arrays. The jump between the results for one and two items is rather big, since the second item is sent as part of the 'remaining' CLCs, whereas for a single item, there is just the first CLC to be sent. The CPU time per ULC then gradually increases due to the fact that the individual items of the sequential protocol, except the first and the last, are copied internally by the decoder, and then the address/size pair of the *copy* is passed on (cf. Section 7.7.1); the copying takes more time the more items there are in the protocol.

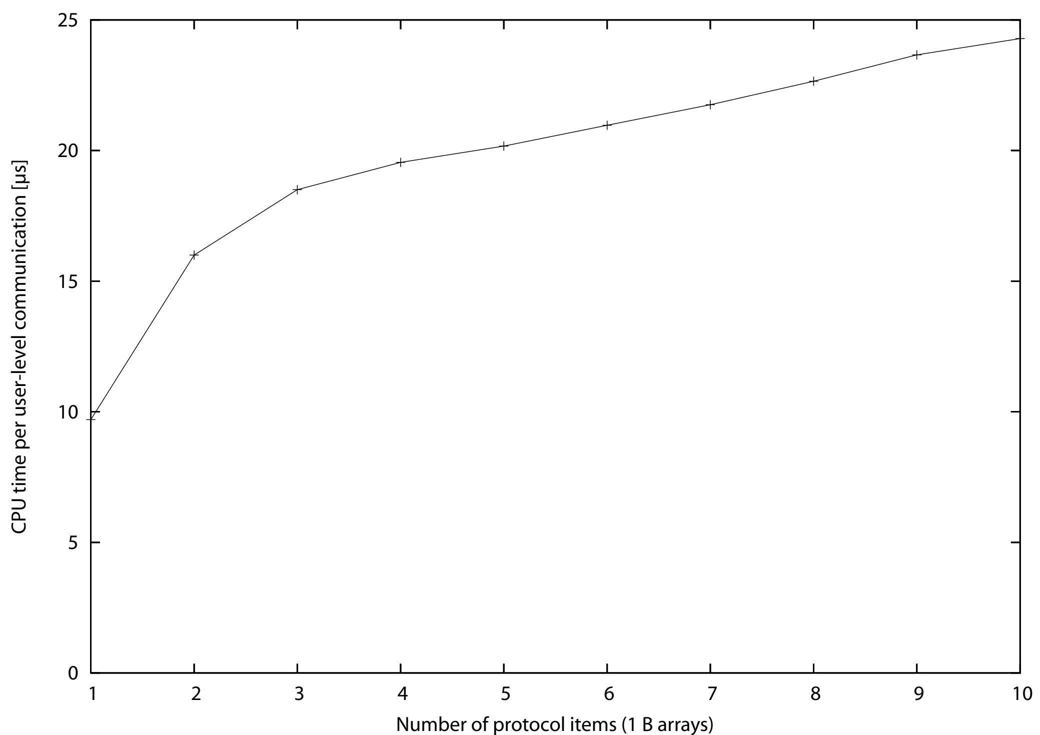

Figure 11.6: CPU overhead: Sequential protocol, 1 B arrays

For sequential protocols with two or more items, there are always two ping-pongs, since the first CLC and the 'remaining' CLCs need to be sent, the latter always in one go. Hence, only the copying of the NLCs of the 'remaining' CLCs causes the gradual increase, with *(n - 2)* copy operations for protocols with n items.

Figure 11.7 shows the CPU time per ULC for sequential protocols of 1 B, 1 KB and 1 MB arrays. For sequential protocols with one and two items, the CPU time per ULC is nearly identical for all three array sizes, since no copying is involved. As expected, from three items onwards, the results diverge, because the aggregate amount of data that needs to be copied depends on the length of the protocol and the size of the individual arrays.

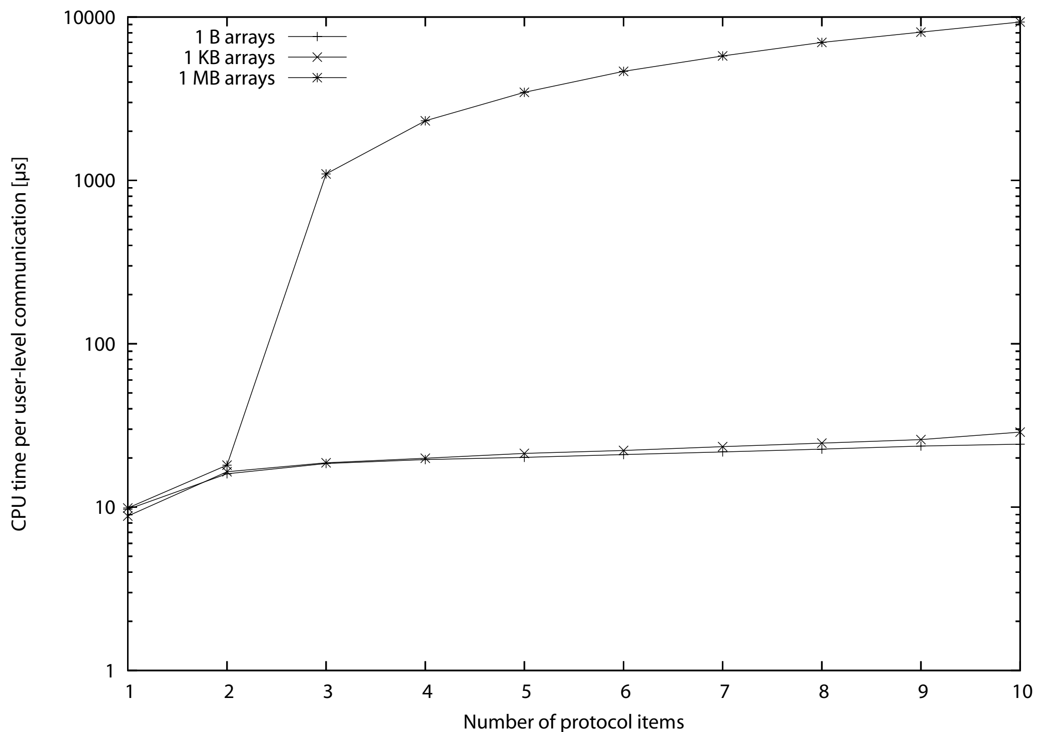

Figure 11.7: CPU overhead: Sequential protocol, several array sizes

Particularly notable is the big jump between two and three items for the 1 MB arrays. This shows the impact of copying large amounts of data — and the advantage of not having to copy non-sequential regular data or mobile data, which will be the bulk of communication in a typical pony application.

Nevertheless, network latency always outweighs local copying. Therefore, copying items of a sequential protocol locally and then sending them in a single network operation is still better than not copying them and sending each item over the network separately.

11.6 Application Scalability

'`mandelbauer`' is an example of using pony to make an existing application distributed; in this case, the original program computes a region of the Mandelbrot set. The approach taken is 'farming': the master node generates work requests for rectangular sections of the region being computed; a number of slave nodes read the requests, do the appropriate computation and send the results back to the master; the master then collects and displays the results. For the purposes of this test, the display has been disabled; the master just measures the rate at which pixels are being computed.

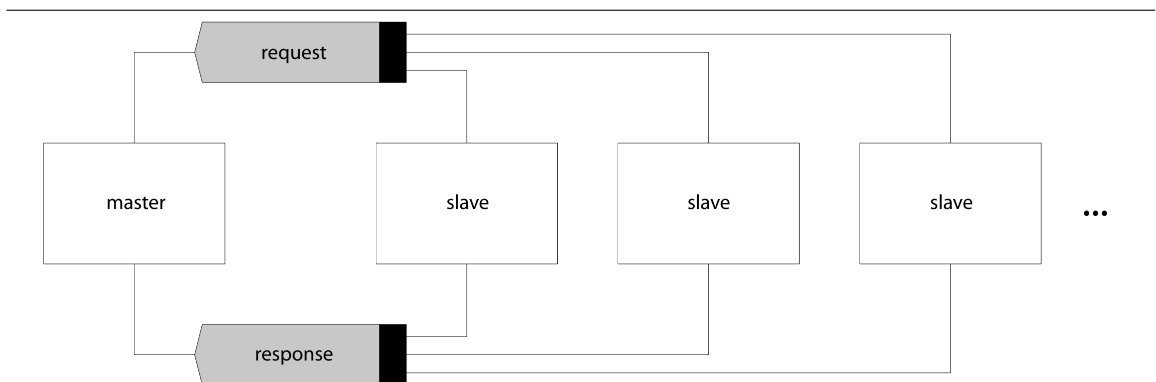

Figure 11.8: The '`mandelbauer`' application: Shared mode

The '`mandelbauer`' application can be run in two modes. In shared mode (see Figure 11.8), there is a single pair of shared networked request/response channels (in two separate NCTs) used by all the slaves. In multiplexing mode (see Figure 11.9), each slave has its own pair of networked request/response channels (in a single NCT), connected to a handler process on the master node. When a slave is started up, it

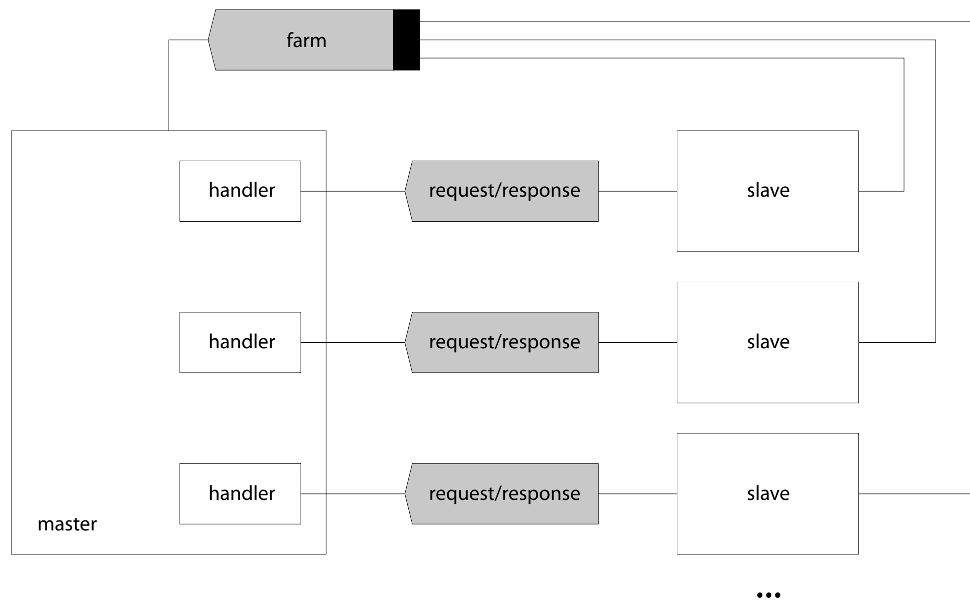

Figure 11.9: The '`mandelbauer`' application: Multiplexing mode

sends the server-end of its 'request/response' NCT to the master over the 'farm' NCT; the master will then set up a new handler process for the slave. The master uses local shared channels to distribute work to and collect results from the handler processes. The slaves have small internal buffers to hold incoming and outgoing messages.

Figure 11.10 shows the rendering performance of '`mandelbauer`' in both modes. Network saturation is reached at 25 slaves in multiplexing mode, at which point CPU utilisation on the slaves in multiplexing mode is approximately 85%; in shared mode it is approximately 30%.

The scaling performance in multiplexing mode is significantly better than in shared mode. Since shared NCT-ends must be explicitly claimed over the network, in shared mode the master is frequently blocked waiting for one of the workers to claim the request channel. Future research will have to look into ways to improve the mechanism for claiming NCT-ends — which would narrow the gap between shared mode and multiplexing mode.

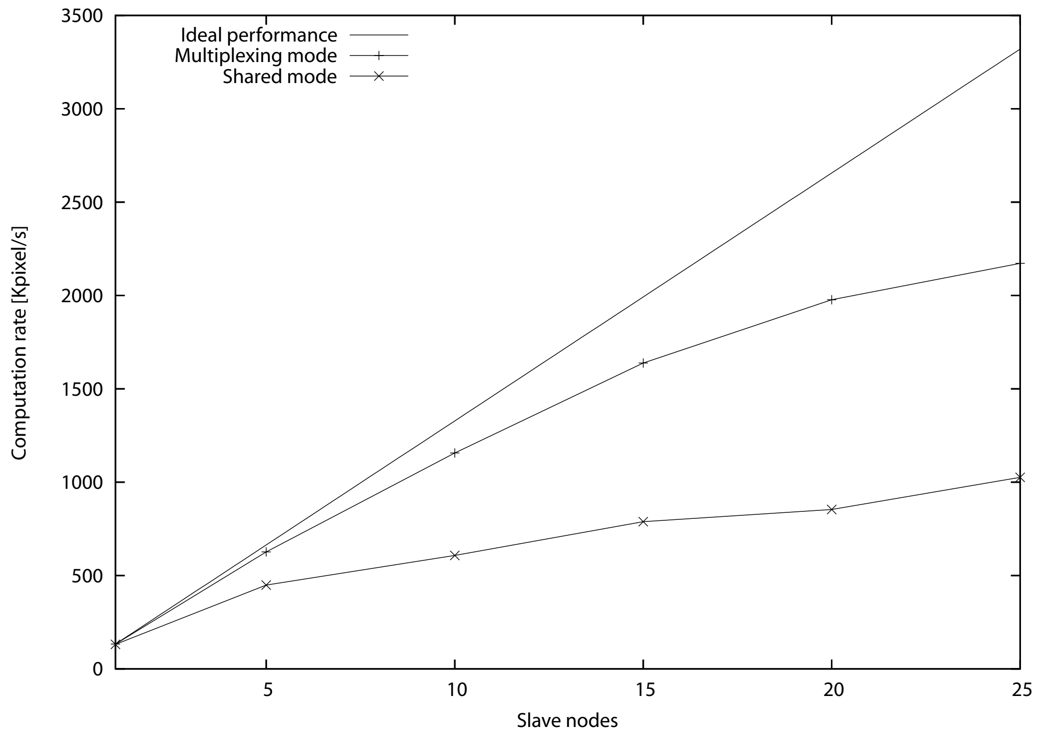

Figure 11.10: Scalability of a distributed application

It is usually considered good practice to run network-bound processes at a higher priority than compute-bound processes, in order to reduce latency for network responses. However, we tried both with and without explicit priorities in this application, and there was no measurable difference — perhaps because, as there is only one calculation process running at a time on each slave node, the pony processes will never be blocked for longer than the time it takes to process one work request.

11.7 Distributed Robust Annealing Case Study

Simulated annealing [SLGS96, SLG97] is a stochastic optimisation algorithm that is used to search for minima (or maxima) of complex problems. A classical example would be the Travelling Salesman, where the goal is to design the shortest route that covers all of the salesman's customers. Simulated annealing begins by randomly

creating a solution to the particular problem, and calculating the 'cost' of the configuration. In the Travelling Salesman, the cost would be the length of the salesman's complete closed route.

The algorithm then searches for a better solution by generating random changes to the original configuration. If the change results in an improvement, it is accepted and serves as the starting point for the next move. If the change yields a poorer solution, it is accepted at a specified probability, which decreases over the life of the algorithm. Thus, eventually only changes which improve the system are likely to be accepted. Under the appropriate conditions, this approach will lead to a result that is probably very close to the true minimum.

Distributed Robust Annealing [Sti05] is a distributed version of the annealing algorithm. There are one master and several worker processes. At the beginning, the master sends the initial configuration to the workers. The workers then independently do a sequence of standard annealing runs, beginning with very short runs and increasing the length by a factor of about 1.2 on each successive run. At the end of each run, the result is returned to the master. The master stores all results it receives from the workers in a large database. When a pre-defined stopping condition is met, each worker terminates independently. The master then selects the best of all results that have been returned by the workers.

The particular annealing problem in this exercise is the Quadratic Assignment Problem (QAP), which deals with the allocation of resources to consumers under certain constraints. This is a classical NP-complete problem. The stopping condition is a minimum allowed rate of improvement of the cost at the end of a run.

In the tests presented here, we measure the 'effective time per iteration', which is calculated by dividing the total time that has passed by the total number of moves performed by all workers during that time. A move consists of randomly selecting one parameter of the system, randomly perturbing that parameter, evaluating the cost of the configuration, then deciding whether to accept the change based on the criteria described above.

During our tests, the typical number of moves was around one million. Due to the large number of moves, and the fact that all workers perform their runs independently without having to interact with each other, the cost of computation is far more relevant than the cost of communication. As can be seen below, this fact had an impact on the outcome of our tests, which led to some interesting results that were not anticipated.

We performed our tests with five different versions of the Distributed Robust Annealing algorithm:

- A non-networked occam-π version running as a single occam-π program. This version did not use pony, and (obviously) ran on a single machine.
- A pony version where the master and all workers were separate nodes and all nodes ran on a single machine.
- A distributed pony version where each node ran on its own dedicated machine.
- A JCSP.net [WAF02, WV02] version with all nodes running on a single machine.
- A distributed JCSP.net version.

Figure 11.11 compares the three occam-π versions. The results for the non-pony version are more or less constant for an increasing number of workers. The distributed pony version is getting faster by a factor roughly proportional to the number of workers. That is, the effective time per iteration for four workers is about a quarter of the time for one worker. These results conform with what was expected.

Somewhat surprising is the result for the pony version running on a single machine. It is getting faster as more nodes are added. So, using the pony version on a single machine creates an advantage compared to the non-pony version, although the overall computational capability of the hardware is the same. Further spot checks with higher numbers of workers showed that a saturation is reached at about ten workers; from

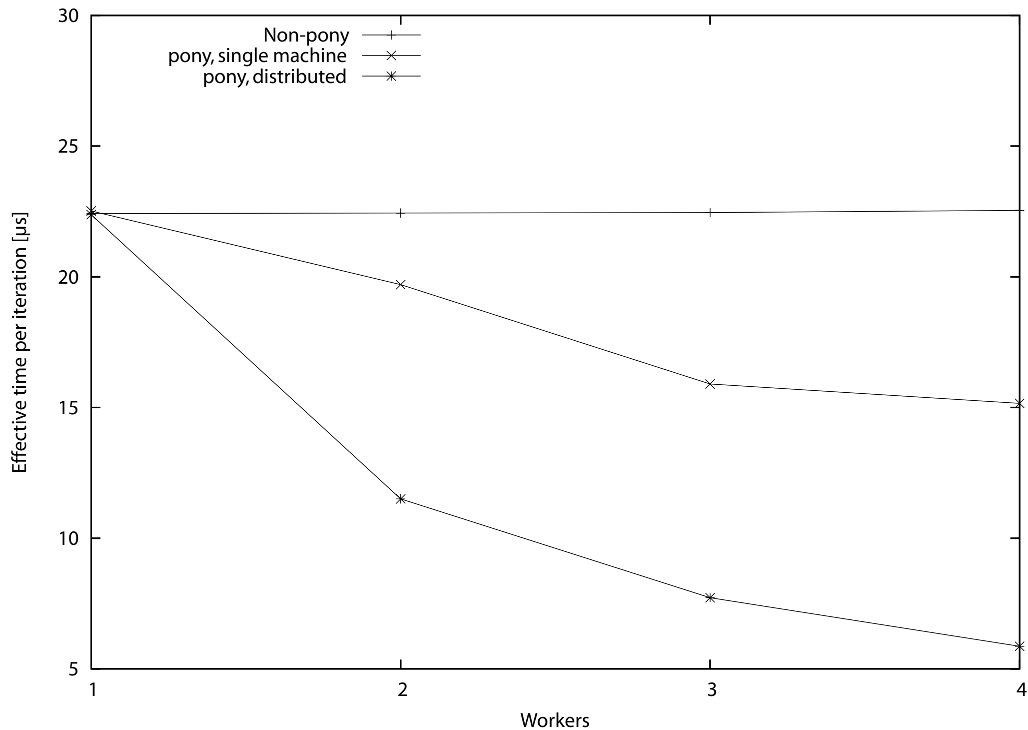

Figure 11.11: Annealing: occam-π versions

then onwards, adding further nodes to the pony version on a single machine does not further decrease the effective time per iteration.

Communication between pony nodes (even on the same machine) is obviously slower than between occam-π processes in a single occam-π program. But, as discussed above, for the annealing algorithm computation outweighs communication as the length of the run increases. The most likely explanation for the advantage of the pony version on a single machine compared to the non-pony version are caching effects.

In the non-pony version, the entire annealing algorithm runs in a single OS-level process. Inside an occam-π program, the KRoC scheduler does not do any time-slicing but schedules the individual occam-π processes according to when/how they interact. In the pony version, there is OS-level scheduling (i.e. time-slicing) between the individual nodes (which are all separate OS-level processes).

The non-pony program is obviously larger than the individual nodes of the pony version. Linux schedules its OS-level processes less frequently than KRoC schedules its occam-π processes. Therefore, the smaller worker nodes of the pony version make fair use of their caches in each time-slice (and pay the cache miss penalties when first scheduled). Within a large KRoC program, the caching behaviour is likely to be unpredictable.

Figure 11.12 compares the pony and JCSP.net versions on a single machine. Figure 11.13 compares the distributed pony and JCSP.net versions. In both cases, the curves of the JCSP.net versions show similar characteristics as in the pony versions, with the JCSP.net versions being a bit faster.

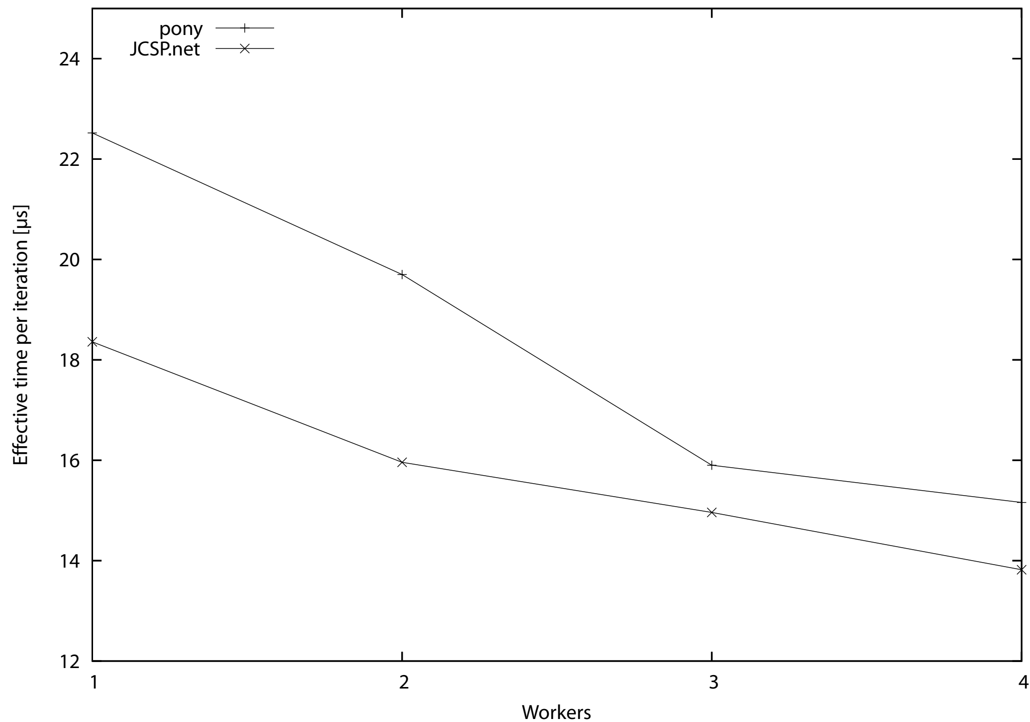

Figure 11.12: Annealing: Single machine versions

Spot checks with higher numbers of workers showed that at about ten workers, the JCSP.net single-machine version reaches a saturation similar to the pony single-machine version. Both versions' results are then within one standard deviation. The

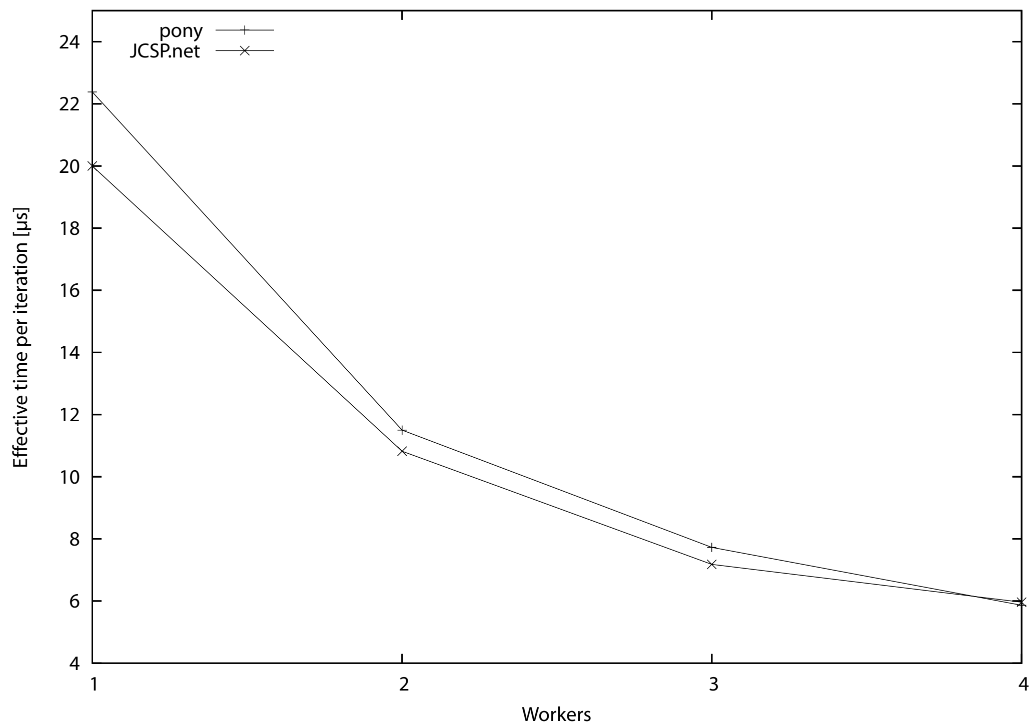

Figure 11.13: Annealing: Distributed versions

same applies to the distributed versions — with an increasing number of workers, the differences between pony and JCSP.net are no longer measurable.

What is surprising is the fact that the JCSP.net versions are faster than the corresponding pony versions. Previous experiences have always shown that JCSP, being built on Java, was slower than occam-π. A likely cause of the opposite effect in our tests is again the specific nature of the annealing, where computation is far more relevant than communication.

A separate test, which only involved the mathematical calculations relevant to our annealing problem (manipulation of elements in large matrices), showed that the occam-π program doing the calculations was slightly faster than the Java program doing the same calculations. The difference was hardly measurable, however, being within the standard deviation.

This leaves three possibilities. Either the Java compiler is a bit more optimised than KRoC in terms of performing the mathematical tasks relevant for our annealing

problem, or the speed advantage of the Java version is due to magnified caching effects (perhaps the JVM is optimised to use the cache more effectively than KRoC), or, as our tests suggest, with a large enough number of workers (and runs long enough), the Java and occam-π versions perform nearly identically. Comparing the mathematical and caching behaviour of Java and occam-π remains an interesting aspect for future research.

Chapter 12

Conclusions and Future Work

12.1 Evaluation of What Has Been Achieved

This thesis has succeeded in developing a unified model for inter- and intra-processor concurrency. The pony environment has become a robust and scalable platform for the development of distributed applications. The author's contribution to the project was the specification of the pony system as presented in this thesis, as well as about 90% of the implementation.

pony expands occam-π's concurrency model into the networked world and achieves semantic and pragmatic transparency according to our objectives in Section 1.3. Semantic transparency has been achieved by consequently ensuring that the semantic behaviour of NCTs — which are the core of pony's distribution model — matches the semantic behaviour of conventional channel-types. Particularly the mobility of NCT-ends allows pony applications to be as flexible and dynamic as their traditional occam-π counterparts. Another important fact is that NCTs may be shared at both ends, and the semantics of shared ends, including the mechanism for claiming and releasing them, is the same as for non-networked channel-types.

pony's protocol-conversion allows all common occam-π protocols to be carried by networked channels. Since the protocol-converters are separate from the main pony kernel, support for future occam-π types and protocols could be added to pony

with comparatively low effort. The internal function of the main pony kernel would not be affected, since the ULC/CLC/NLC system would be used for new protocols as well. A special case might need to be made for protocols that are not 'plain data' but constitute a new communication primitive. In the near future, this applies particularly to mobile barriers, see Section 12.2.2.

Pragmatic transparency has been achieved by a variety of measures. Most importantly, communication over a channel-type whose ends are on the same node is done in the traditional occam-π way by only accessing the channel-word. This means that plain communication does not involve additional overheads if performed locally. Another aspect of pragmatic transparency is that communicating ULCs between nodes has been optimised to require as few network-messages as possible. Also, the amount of copying data-items has been kept low. On the sending node, no copying is necessary at all except for non-mobile data-items in sequential protocols with more than two items.

A small extra cost had to be introduced to accommodate the needs of networkable CTBs. The overhead of the larger CTB memory layout in a pony-enabled KRoC build only has a minor impact in most applications and configurations. An exception may be applications with millions of (possibly non-networked) channel-types. The small extra runtime cost related to the claiming and releasing of shared NCT-ends should also typically present no problem. A small inconvenience at the moment is the incompatibility of pony-enabled and non-pony-enabled KRoC builds. Possible ways to overcome this incompatibility are discussed in Section 12.3.2. This becomes particularly relevant in applications like the ones mentioned above, where there may be a potentially huge number of channel-types that are never going to be networked — according to the proposal in Section 12.3.2, the CTBs of those channel-types would not need to contain the pony-specific overheads.

In the pony kernel, there is a clean separation between the logical function and the networking function. This allows a better abstraction, especially when it comes to extending pony in the future. This applies to adding new features (such as support

for new protocols) to the 'front-end' of the pony kernel as well as to adding new network drivers in order to support additional network-types.

Our tests have shown that pony already has acceptable performance for distributed occam-π applications, and that existing occam-π applications can easily be adapted to take advantage of pony (cf. also Appendix C). Both the networking overhead and the CPU overhead of pony are very small, and are therefore more than outweighed by the ease of use due to pony's semantic transparency.

The scalability of pony applications is acceptable as well, especially considering the fact that very little 'tuning' work has been done on pony so far. We have also identified areas where future work on the pony implementation may improve the performance of distributed applications. We hope to test pony in a Grid environment in the future to identify any scaling problems with larger systems.

The handling of pony for the occam-π programmer is simple and straightforward. As far as the basic functionality is concerned, only the startup and shutdown of pony, as well as the explicit allocation of NCT-ends, require a specific consideration of pony by the programmer. This is done through a minimum number of public processes which provide the interface between pony and the user-level code.

Startup and shutdown only need to be done once per node, the explicit allocation only once for each (explicitly allocated) NCT-end. Apart from this, all interaction between the user-level code and pony is implicit, which includes the implicit allocation of NCTs. All runtime operations are handled automatically and transparently by the pony kernel.

Error-handling and message-handling are useful 'add-ons' to the basic functionality of pony. While the error-handling is not transparent (since occam-π lacks an underlying fault-tolerance mechanism), it is nevertheless a useful means for detecting errors, allowing the programmer to write more robust pony applications. pony's message-handling is a useful debugging tool, especially when it comes to extending pony in the future.

The configuration of pony is easy to understand and minimises the complexity of setting up a distributed application. Most settings can be determined automatically; the only setting that is typically required to start up a node is the location of the ANS. This simple configuration mechanism allows programmers to write 'plug and play' pony applications that will work straight away without requiring a complex setup procedure first.

Naturally, this thesis could only break the ground for the new concurrency model, and the potential for further development never ceases. The following sections give an overview of what could (and should) be done in the future to further improve the pony environment and to keep up with the current developments in the occam-π world.

12.2 Adding Support for New/Future occam-π Features

12.2.1 Supporting Mobile Processes

Mobile processes [BW04] have been added to occam-π recently and are still largely experimental. Adding support for mobile processes in pony will not require many changes in the internal structure of the pony environment since they do not introduce a new communication primitive (unlike channel-types). What has to be dealt with are items *inside* a mobile process that are carried with it when it moves over a networked channel to another node, such as mobile data and channel-type-ends. This can be done by the protocol-converters.

Mobile data-items inside the mobile process require the adaptation of their pointers on the receiving node. This can be done by the protocol-encoder on the receiving node; the internal function of pony would not be affected. Dealing with channel-type-end variables stored in the workspace of mobile processes would involve a channel-type-end NLC for each of them in the usual way (which may involve an

implicit allocation), plus the adaptation of the pointers by the encoder (as for mobile data-items).

The code for a mobile process could be stored in a special repository, for instance on the master node or in the ANS. If the protocol-encoder on the receiving node does not yet have the code for an arriving mobile process available, it would request it from the repository. In this way, large code segments would only need to be sent over the network when needed. In order to avoid naming conflicts when requesting the code for a mobile process from the repository, a hash value could be used, similar to the type-hash for channel-types. Apart from adding the code repository, only the protocol-converters would need to be extended to accommodate mobile processes.

12.2.2 Supporting Mobile Barriers

Mobile barriers [WB05] are currently being added to occam-π and are still at an experimental stage. Unlike a mobile process, a mobile barrier is a new communication/synchronisation primitive. Therefore, support for mobile barriers in pony will require some additions to the internal layout of the pony infrastructure, such as 'barrier-handlers' on the master node keeping track of which processes are currently enrolled on the barrier. The role of such a barrier-handler for a networked barrier would be similar to that of an NCT-handler for an NCT.

12.2.3 Supporting RMoX

Integrating pony into the RMoX operating system [BJV03] should be relatively straightforward. The internal structure of pony does not need to be changed; just the network-specific code, i.e. the link-handler, the link-manager and the ANS, would need to be adapted in order to use the RMoX built-in network drivers.

12.2.4 Supporting Buffered Channels

The proposed 'BUFFERED' channels [Wel04a] for occam-π provide the opportunity for reducing network latency. pony support for 'BUFFERED' channels may be used to reduce the number of network acknowledgements needed when sending streams of buffered data.

12.2.5 Supporting Behaviour Patterns

Within the Concurrency Research Group at the University of Kent, the idea of meta 'behaviour patterns' for channel-types has been discussed. These patterns could be declared in a BNF-style syntax along with the type declaration of the channel-type itself and specify valid sequences of communication along the channels inside the channel-type. This notion is similar to Boosten's Formal Contracts [Boo03] or the contracts in Microsoft's Singularity project [Mic05].

pony may utilise behaviour patterns to reduce network latency. For instance, if a channel-type contains a request and a reply channel that are communicating (by behaviour pattern) in a ping-pong style, pony could use that information to save unnecessary network acknowledgements (since each communication could be used as an acknowledgement for the previous one). At the moment these ideas are purely theoretical, however. Since adding pony support for ping-pong style communication would be non-trivial, it makes sense not to do it until the occam-π language itself provides a suitable syntax for expressing it — particularly so since the key objective of pony is transparency, and the usage of pony should interfere as little as possible with the usual occam-π way of programming.

12.2.6 Supporting the Proposed 'GATE'/'HOLE' Mechanism

Supporting the proposed 'GATE'/'HOLE' mechanism [Sch04, Wel04b] (or similar approaches) in occam-π could further enhance pony's performance. The 'GATE'/'HOLE'

mechanism was proposed in order to tackle a drawback arising from the new dynamic features in occam-π. The classical static occam fixed the design of its process networks (and the channels between the processes) at compile time. It was always obvious over which 'interface' a process would communicate with its environment. By introducing mobile channel-types, we have unfortunately also introduced the possibility of hidden communication routes that are not declared in the interface (i.e. the header) of the process.

Previously, the ways in which a process interacted with its environment (e.g. through channels and barriers) could be statically and explicitly listed in the process header. Introducing mobile channel-types means that the set of possible interactions for any process can grow at runtime, so that interactions can take place that were not declared by its interface. This raises specification and security issues that are similar to those found in common OO languages (where aliasing is endemic and the opportunities for object interaction exceed those declared by their public interfaces [Loc01, Wel00]).

At the Concurrency Research Group at Kent, the following rules have been proposed to ensure that there are no hidden interactions between a process and its environment — a property which we call 'structural integrity' — despite the mobility of channel-types:

Definition:

(a) Channel-type-end parameters *may* be qualified as being '`GATE`' or '`HOLE`'. '`GATE`' and '`HOLE`' parameters are *live*.

(b) All other channel-type-ends are *dead* (i.e. locally declared channel-type-end-variables and parameters not qualified as '`GATE`' or '`HOLE`').[1]

[1] This property of channel-type-ends is *static* — each variable is either *always* '`GATE`' or *always* '`HOLE`' or *always* dead.

Usage:

(c) A process may not communicate over a dead channel-type-end.

Assignment/Communication:

(d) '`GATE`' channel-type-end parameters have '`VAL`' semantics — they may not be changed inside the process in whose header they are declared.

(e) '`GATE`', '`HOLE`' and dead channel-type-ends may be freely assigned/communicated to each other as long as this does not break Rule (d).[2]

Parameter-Passing:

(f) Arguments for '`GATE`' parameters may only be live variables — unless the process is being forked. If the process is being forked, both live and dead arguments are allowed, as long as this does not break Rule (d).[3]

(g) Inside the scope of a '`CLAIM`', a claimed shared channel-type-end may be passed as an argument *only* to an unshared '`GATE`' parameter of a process that is not being forked.[4]

(h) '`HOLE`' parameters are initially undefined when a process starts. Arguments for '`HOLE`' parameters may be outer '`HOLE`' parameters of matching type which must be currently undefined, or the keyword '`HOLE`'. The latter may only be supplied to forked processes. '`HOLE`' parameters have no return value (i.e. for the calling process they are still undefined when the called process terminates).

[2] It would, for instance, be possible to assign a '`SHARED GATE`' parameter to a dead variable, but it would not be possible to assign an unshared '`GATE`' parameter to another variable because this would leave the '`GATE`' parameter undefined, which is not allowed. Nothing can be assigned/communicated to a '`GATE`'.

[3] Note that the semantics of passing arguments to parameters of forked processes is anyway that of communication. So, this clause conforms with Rule (e).

[4] This forces conformity to the existing rule that inside a `CLAIM`, a live parameter may only be used for communication; it technically becomes unshared and its value frozen. Please note that the possibility of passing a claimed shared channel-type-end to an unshared parameter, provided that the value of that parameter is not changed inside the process, has already been incorporated in KRoC (cf. the possibility to use shared handles for calling `pony`'s public processes — they have to be claimed beforehand as well).

(i) Arguments for dead parameters may only be dead or '`HOLE`' variables — unless the process is being forked. If the process is being forked, '`GATE`' variables are also allowed as arguments, as long as this does not break Rule (d).

The aim of these rules is that processes only interact with their environment through formally declared live parameters. In the case of '`HOLE`' parameters, what they are bound to may change dynamically, but only by explicit action of the processes themselves (by internally assigning or communicating a newly acquired channel-type-end to one of its '`HOLE`' parameters). But the external shape of a process does not change — we have structural integrity. There are no undeclared routes into or out of the process.

Additional issues arise from forking. It is proposed to restrict forking so that a process cannot fork off another process without the calling process being aware of it. This could be implemented by introducing a '`FORKS`' keyword after which a process would declare all possible processes that it (or any subsequently called processes) might fork off (similarly to exceptions and the '`throws`' keyword in Java). Since this might be a rather heavy burden for the programmer, a lighter approach could be a marker by which a process may be marked as a '`FORKING PROC`'. Only '`FORKING PROC`s' would then be allowed to fork off other processes or subsequently call other '`FORKING PROC`s'.

Since with the '`GATE`'/'`HOLE`' approach, there would be a clear distinction between 'live' and 'dead' channel-type-end variables, i.e. between variables that are used for communication and those that are not, pony could exploit this information by not setting up the internal pony components for a networked CTB until at least one of its ends is used for communication. In this way, NCT-ends that are just 'passing through' a node would not require setting up the internal pony infrastructure, thus saving resources.

12.2.7 Supporting New Fault Tolerance Mechanisms

As discussed in Section 4.1.1, the only part of pony that is not transparent at the moment is the error-handling. There have been various proposals to add fault-tolerance to occam-π, for instance Barnes' 'TRY'/'CATCH' approach in [Bar03]. When such a fault-tolerance mechanism is added to occam-π, pony could use it to improve its own error-handling in order to make it more transparent to the occam-π programmer.

12.3 Other Things

12.3.1 Adding Support for Networked Plain Channels

Plain network-channels — i.e. the networked version of plain (classical occam) channels — can be implemented as 'anonymous' NCTs that contain exactly one channel. This approach is similar to the implementation of the anonymous 'SHARED' channels mentioned in Section 1.5.5 (discussed in detail in [Bar03]). The following declaration:

```
NET CHAN INT iw!:                -- These network-channel-ends
NET CHAN BOOL br?:               --   must be allocated before
SHARED NET CHAN BOOL sbw!:       --   we can communicate over them!
SHARED NET CHAN INT sir?:
SEQ
  ...  Allocate iw!, br?, sbw!, sir?
  ...  Use iw!, br?, sbw!, sir?
```

would have the semantics of:

```
CHAN TYPE $anon.INT              -- Compiler-generated type
  MOBILE RECORD
    CHAN INT x?:                 -- Server-end holds reading-end
:
```

```
CHAN TYPE $anon.BOOL            -- Compiler-generated type
  MOBILE RECORD
    CHAN BOOL x?:               -- Server-end holds reading-end
:

$anon.INT! iw$cli:
$anon.BOOL? br$svr:
SHARED $anon.BOOL! sbw$cli:
SHARED $anon.INT? sir$svr:
SEQ
  ...  Allocate iw$cli, br$svr, sbw$cli, sir$svr
  ...  Use iw$cli, br$svr, sbw$cli, sir$svr
  ...    resp. iw$cli[x], br$svr[x], sbw$cli[x], sir$svr[x]
```

where the server-end of the compiler-generated channel-type by definition holds the reading-end of the channel. Before a process can communicate over a network-channel-end, that end would need to be allocated. This would be done using allocation processes similar to those for NCT-ends.

The compiler would replace any occurrences of network-channel-ends in the user-level code by the appropriate generated variables. In parameters, network-channel-ends would be replaced by the generated channel-type-end (e.g. '`br?`' would be replaced by '`br$svr`'). When used for communication, they would be replaced by the actual channel-field (e.g. '`iw ! 5`' would be replaced by '`iw$cli[x] ! 5`').

12.3.2 Adding Support for '`LOCAL`' High Performance Channels

As described in Section 6.1, in a pony-enabled KRoC build, CTBs have a larger memory footprint than 'traditional' CTBs. Furthermore, the special claim/release mechanism with network-hook, state-field and state-semaphore consumes additional runtime. Although the extra cost is small and inevitable in order to provide full

transparency, there may be situations where the programmer might want to eliminate those overheads. This is of course only possible for channel-types that are not networked and will never become networked in their lifetime.

Currently, the programmer only has the choice to use either a traditional KRoC build or a pony-enabled one. This means that either networked channel-types are impossible across the entire application or all channel-types are network*able* — with the known overheads. A 'mix and match' solution is not possible at the moment.

This could be achieved by adding an optional 'LOCAL' keyword to occam-π. Channel-type-end variables could be declared 'LOCAL'; this would restrict them to be used in a non-networked ('traditional') way. Only pairs of 'LOCAL' or pairs of networkable channel-type-ends could be allocated together. The 'LOCAL' ones would be allocated with the traditional CTB layout and without the special claim/release mechanism. For 'LOCAL' channel-type-ends, it would only be allowed to send them over channels that are specifically declared to carry 'LOCAL' ends, or over channels inside other 'LOCAL' channel-types. In this way, 'LOCAL' and networkable channel-types would not get mixed up, and accidentally moving a 'LOCAL' channel-type-end to another node would be impossible (since pony's allocation processes would require non-'LOCAL' channel-type-ends).

With this approach, it would still be possible to distinguish between pony-enabled and traditional KRoC builds; however, they could be made more compatible than before. The compiler would need to treat channel-types in programs compiled by a traditional KRoC build as implicitly 'LOCAL'. If a pony-enabled program wanted to use an occam-π library that was compiled with a traditional KRoC build, it would simply need to treat all channel-types in the library as 'LOCAL', conforming with the above rules. Or put another way, the library would need to be used as if it were a pony-enabled one where every single channel-type-end was declared 'LOCAL'.

A further enhancement could be a compiler analysis to identify NCTs whose ends are declared locally and never leave the node on which they are declared. These NCT-ends could then be tagged 'LOCAL' by the compiler automatically. Implementing

such an analysis would be non-trivial, however, and require complex changes in the compiler.

12.3.3 Supporting Different Architectures

At the moment, pony only runs on Linux/x86 platforms. KRoC/occam-π (with its new dynamic features) is currently being ported to many different platforms besides x86. Supporting different architectures in pony would involve adapting the compiler-specific code (the built-in compiler support and the low-level drivers) and re-compiling the library.

Ultimately, we want to be able to bridge different platforms across the same pony application. This raises the question of how to cope with different endianisms on the different platforms. For this, the pony infrastructure on each node must be aware of the endianism of each node to which it is connected. This could be done by adding a flag to each network-message that indicates the endianism of the node from which the message comes. Alternatively, each node could be notified about the endianism of a remote node by the time a new link to that remote node is established, so that the 'endianism flag' in every single network-message would not be necessary.

When network-messages are exchanged between nodes with different endianisms, the relevant parts of the message must be converted to the endianism of the target node, which could be done either on the sending or on the receiving node. This conversion would happen at different places in the pony environment. Data-item NLCs would need to be converted in the protocol-converters. The conversion of CLC-descriptors would be done in the decode- and encode-handlers. All other parts of a network-message could be converted in the network drivers directly. An alternative might be to send all network-messages in plain-text XML or a similar protocol. From a performance point of view, this seems hardly viable, however.

Bridging several platforms across the same pony application would also affect a possible code repository for mobile processes as mentioned in Section 12.2.1. Rather

than storing the native code of several platforms for the same mobile process, the code repository could store ETC code [Poo98]. Inside the protocol-encoder on the receiving node, a just-in-time compiler could then compile the mobile process into native code of the target platform.

12.3.4 Security and Reliability

An area that has not been explored yet with respect to the pony environment is security. In order to make pony applicable in wider markets, an encryption model for pony needs to be developed. A possible approach could be public/private keys similar to the ssh model.

There may be fields of application where encryption cannot be used, for instance if a server running in a pony application provides services to the world. In such a scenario, everyone may start a node that joins the application in order to use the service provided. This raises the possibility of Denial of Service attacks. Future research may investigate the detection and defense against such attacks in pony applications, as well as other well-known security implications that networked applications face today.

In order for pony to become more reliable, an improvement for the ANS could be to make its state persistent. Currently, if the ANS crashes, all information stored in its database is lost. It would be useful to be able to restart the ANS and restore its state after a failure happened.

12.3.5 Simplifying the Setup

Although setting up a node is simple and straightforward in pony, there may be ways to improve it. This concerns for instance the way pony is configured. Possible improvements could be supporting a search path for the configuration files in an environment variable, or the possibility to configure settings directly via environment

variables. It would also be interesting to look into ways to find an ANS dynamically through a web page. All these things should be alternatives to the current configuration files rather than replacing them.

In certain scenarios, for instance when running a pony application on a large cluster, it may be desirable not to have to start each node separately. Future research might look into possibilities to automate the startup of pony applications by launching nodes automatically, for instance by using Grid-style cluster management.

Another field to investigate in the future could be support for Zeroconf-style [Zer06a, Zer06b] setup of pony applications on local clusters. Similar to Multicast DNS, a kind of 'Multicast ANS' mechanism might be developed to enable nodes of an application to find each other without the need for an ANS. For applications spread over big or even global networks such as the internet, the ANS would still be a necessary and sensible means for interconnecting nodes, but for local clusters, a Zeroconf-style setup mechanism could further simplify the use of the pony environment.

Bibliography

[Bar00a] Fred Barnes. *Socket, File and Process Libraries for occam*. Computing Laboratory, University of Kent at Canterbury, June 2000. Available at: `http://www.cs.kent.ac.uk/people/staff/frmb/documents/`.

[Bar00b] F.R.M. Barnes. Blocking System Calls in KRoC/Linux. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures*, volume 58 of *Concurrent Systems Engineering*, pages 155–178, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[Bar03] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent, June 2003.

[Bar05] F.R.M. Barnes. Interfacing C and occam-pi. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *WoTUG-28, Concurrent Systems Engineering, ISSN 1383-7575*, pages 249–260, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN: 1-58603-561-4.

[BDv99] M. Boosten, R.W. Dobinson, and P.D.V. van der Stok. MESH: MEssaging and ScHeduling for Fine-Grain Parallel Processing on Commodity Platforms. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*

(PDPTA'1999), Las Vegas, Nevada, June 1999. CSREA press. ISBN: 1-892512-15-7.

[BJV03] F.R.M. Barnes, C.L. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 269–288, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN: 1-58603-381-6.

[Boo03] M. Boosten. Formal Contracts: Enabling Component Composition. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 185–197, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN: 1-58603-381-6.

[Bro04] N.C.C Brown. C++CSP Networked. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, volume 62 of *WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575*, pages 185–200, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

[BW01] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 243–264, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[BW02] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part I. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 331–361, IOS

Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[BW03] N.C.C. Brown and P.H. Welch. An Introduction to the Kent C++CSP Library. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 139–156, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN: 1-58603-381-6.

[BW04] F.R.M. Barnes and P.H. Welch. Communicating Mobile Processes. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, volume 62 of *WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575*, pages 201–218, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

[CG89] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–459, April 1989.

[CO03] Vincent Cremet and Martin Odersky. PiLib: A Hosted Language for Pi-Calculus Style Concurrency. In *Domain-Specific Program Generation*, pages 180–195, 2003.

[FK97] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 1997. Available at: `ftp://ftp.globus.org/pub/globus/papers/globus.pdf`.

[FKNT02] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. The Psychology of the Grid: An Open Grid Services Architechture for Distributed Systems Integration. *Global Grid Forum*, June 2002. Available at: `http://www.globus.org/research/papers/ogsa.pdf`.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 2001. Available at: `http://www.globus.org/research/papers/anatomy.pdf`.

[FKT02] I. Foster, C. Kesselman, and S. Tuecke. What is the Grid? A Three Point Checklist. *GRIDToday*, July 2002. Available at: `http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf`.

[Goo01] I.N. Goodacre. occam NetChans, March 2001. Project report, Computing Laboratory, University of Kent at Canterbury.

[Hil05] G.H. Hilderink. Exception Handling Mechanism in Communicating Threads for Java. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *WoTUG-28, Concurrent Systems Engineering, ISSN 1383-7575*, pages 317–334, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN: 1-58603-561-4.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.

[IBM03] IBM Corporation. Globus Toolkit 3.0 Quickstart, Redpaper. Technical report, IBM Corporation, 2003. Available at: `http://www.redbooks.ibm.com/redpapers/pdfs/redp3697.pdf`.

[Ind04] Indiana University LAM Team. LAM/MPI User's Guide. Technical report, Indiana University, May 2004. Available at: `http://www.lam-mpi.org/download/files/7.0.6-user.pdf`.

[Inm88] Inmos Limited. *Transputer Reference Manual*. Prentice Hall, March 1988. ISBN: 0-13-929001-X.

[Inm93] Inmos Limited. *The T9000 Transputer Instruction Set Manual*. SGS-Thompson Microelectronics, 1993. Document number: 72 TRN 240 01.

[Inm95] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: `http://wotug.org/occam/`.

[Loc01] T.S. Locke. Towards a Viable Alternative to OO – extending the occam/CSP programming model. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 329–349, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[Mic05] Microsoft Research. An Overview of the Singularity Project, 2005. Microsoft Research Technical Report MSR-TR-2005-135. Available at: `http://research.microsoft.com/os/singularity/`.

[Mil99] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.

[MM98] Henk L. Muller and David May. A simple protocol to communicate channels over channels. In *EURO-PAR '98 Parallel Processing, LNCS 1470*, pages 591–600, Southampton, UK, September 1998. Springer Verlag.

[Moo99] J. Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, The Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[MPI97] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, MPI Forum, July 1997. Available at: `http://www.mpi-forum.org/docs/mpi-20.ps`.

[MTW93] M.D. May, P.W. Thompson, and P.H. Welch. *Networks, Routers and Transputers*, volume 32 of *Transputer and occam Engineering Series*. IOS Press, 1993.

[MW96] J.M.R. Martin and P.H. Welch. A Design Strategy for Deadlock-free Concurrent Systems. In *Transputer Communications*, volume 3 (4), pages 215–232. Wiley and Sons Ltd., UK, October 1996.

[OAC+05] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The Scala Language Specification Version 1.0. `http://scala.epfl.ch/docu/files/ScalaReference.pdf`, October 2005.

[Obj93] Object Management Group. The Common Object Request Broker: Architecture and Specification (CORBA). Technical report, Object Management Group, December 1993. Available at: `ftp://ftp.omg.org/`.

[Poo96] M.D. Poole. Occam for all – two approaches to retargetting the INMOS compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 167–178, Amsterdam, The Netherlands, March 1996. World occam and Transputer User Group, IOS Press. ISBN: 90-5199-261-0.

[Poo98] M.D. Poole. Extended Transputer Code - a Target-Independent Representation of Parallel Programs. In P.H. Welch and A.W.P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed*

Applications, Proceedings of WoTUG 21, volume 52 of *Concurrent Systems Engineering*, pages 187–198, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language Based on the Pi-Calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[PV02] K.S. Pedersen and B. Vinter. Java PastSet: A Structured Distributed Shared Memory System. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 97–108, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[SBW03] M. Schweigler, F.R.M. Barnes, and P.H. Welch. Flexible, Transparent and Dynamic occam Networking With KRoC.net. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 199–224, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN: 1-58603-381-6.

[Sch01] M. Schweigler. The Distributed occam Protocol – A New Layer On Top Of TCP/IP To Serve occam Channels Over The Internet. Master's thesis, Computing Laboratory, University of Kent at Canterbury, September 2001. MSc Dissertation.

[Sch04] M. Schweigler. Adding Mobility to Networked Channel-Types. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, WoTUG-27, Concurrent Systems

Engineering, ISSN 1383-7575, pages 107–126, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

[SLG97] G.S. Stiles, F.H. Lee, and C. Gyulai. Power-Law Convergence of Stochastic Optimization Algorithms. In H.R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 1158–1566, Las Vegas, Nevada, USA, June 1997. CSREA Press. ISBN: 0-9648666-8-4.

[SLGS96] G.S. Stiles, F.H. Lee, C. Gyulai, and V. Swaminathan. The Speedup of Parallel Randomized Approximation Algorithms. In H.R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, pages 1284–1295, Sunnyvale, California, USA, August 1996. CSREA Press. ISBN: 0-9648666-4-1.

[SS06] M. Schweigler and A.T. Sampson. pony – The occam-π Network Environment. In Peter Welch, Jon Kerridge, and Fred Barnes, editors, *Communicating Process Architectures 2006*, WoTUG-29, Concurrent Systems Engineering, ISSN 1383-7575, pages 77–108, Amsterdam, The Netherlands, September 2006. IOS Press. ISBN: 1-58603-671-8.

[Sti05] G.S. Stiles. An occam-pi Implementation of a Verified Distributed Robust Annealing Algorithm. In H.R. Arabnia, editor, *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2005)*, volume 1, pages 208–214. CSREA Press, June 2005.

[Sun90] V.S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[Sun05] Sun Microsystems. Application Isolation API Specification, July 2005. Available at: http://jcp.org/aboutJava/communityprocess/pr/jsr121/.

[SWP+05] S. Stepney, P.H. Welch, F.A.C. Pollack, J.C.P. Woodcock, S. Schneider, H.E. Treharne, and A.L.C. Cavalcanti. TUNA: Theory Underpinning Nanotech Assemblers (Feasibility Study), January 2005. EPSRC grant EP/C516966/1. Available from: `http://www.cs.york.ac.uk/nature/tuna/index.htm`.

[Uta06] Utah State University. The Utah State University Website, 2006. Available at: `http://www.usu.edu/`.

[Vel98] Kevin Vella. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations*. PhD thesis, The University of Kent at Canterbury, Canterbury, Kent. CT2 7NF, December 1998.

[Vin05] B. Vinter. The Architecture of the Minimum intrusion Grid (MiG). In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *WoTUG-28, Concurrent Systems Engineering, ISSN 1383-7575*, pages 189–201, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN: 1-58603-561-4.

[vN93] John von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[VW99] K. Vella and P.H. Welch. CSP/occam on Shared Memory Multiprocessor Workstations. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, pages 87–119, Amsterdam, The Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[WAF02] 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, editors, *Computational Science - ICCS 2002,*

volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002. ISBN: 3-540-43593-X. See also: `http://www.cs.kent.ac.uk/pubs/2002/1382`.

[WB05] P.H. Welch and F.R.M. Barnes. Mobile Barriers for occam-pi: Semntics, Implementation and Application. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *WoTUG-28, Concurrent Systems Engineering, ISSN 1383-7575*, pages 289–316, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN: 1-58603-561-4.

[Wel89] P.H. Welch. Graceful Termination – Graceful Resetting. In *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10*, pages 310–317, Enschede, Netherlands, April 1989. Occam User Group, IOS Press, Netherlands. ISBN 90 5199 007 3.

[Wel99] P.H. Welch. CSP for Java (JCSP), 1999. Available at: `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[Wel00] P.H. Welch. Process Oriented Design for Java – Concurrency for All. In *PDPTA 2000*, volume 1, pages 51–57. CSREA Press, June 2000. ISBN: 1-892512-52-1.

[Wel04a] P.H. Welch. Buffered Channels (and `PRI ALT`s). Technical Report UKC-CRG-15-04-2004, Computing Laboratory, University of Kent, Canterbury, UK, April 2004.

[Wel04b] P.H. Welch. Maintaining Structural Integrity in Dynamic Systems. Technical Report UKC-CRG-11-03-2004, Computing Laboratory, University of Kent, Canterbury, UK, March 2004.

[Wik06] Wikipedia. The Free Encyclopedia. Nagle's algorithm, 2006. Available at: `http://en.wikipedia.org/wiki/Nagle's_algorithm`.

[WMBW06] P.H. Welch, J. Moores, F.R.M. Barnes, and D.C. Wood. The KRoC Home Page, 2006. Available at: http://www.cs.kent.ac.uk/projects/ofa/kroc/.

[WV02] P.H. Welch and B. Vinter. Cluster Computing and JCSP Networking. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 213–232, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[WW96] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166, Amsterdam, The Netherlands, March 1996. World occam and Transputer User Group, IOS Press. ISBN: 90-5199-261-0.

[Zer06a] Zeroconf. Zero Configuration Networking (Zeroconf), 2006. Available at: http://www.zeroconf.org/.

[Zer06b] Zeroconf and DNS Extensions. Multicast DNS, 2006. Available at: http://www.multicastdns.org/.

Part IV

Appendices

The appendices provide supplementary information that is not part of the main thesis. Appendix A contains a list of abbreviations and acronyms used in this thesis. Appendix B gives a comprehensive reference of the public interface of the pony environment, consisting of the public processes, data-types and constants. Appendix C compares the traditional implementation of the '`commstime`' benchmark with a distributed one using pony, in order to give a practical example of how to use pony to make an existing application distributed. Finally, Appendix D lists the author's publications that are related to pony and its development.

Appendix A

Abbreviations and Acronyms

The following list contains the pony-specific abbreviations and acronyms used in this thesis in alphabetical order.

Abbreviation or acronym	**Meaning**	**Where explained?**
ANS	Application Name Server	Section 2.1.3
CLC	compiler-level communication	Section 7.2
CTB	channel-type-block	Section 6.1
NCT	network-channel-type	Sections 2.1.2 and 6.1
NLC	network-level communication	Section 7.2
pony	occam-π Network Environment	Chapter 1
ULC	user-level communication	Section 7.2

Appendix B

The Public pony Interface

B.1 Public pony Processes

This section contains a reference of all public processes of the pony environment, complete with a description of their parameters. Developers of distributed occam-π applications would call these processes in order to access pony's functionality.

B.1.1 Startup Processes

pony.startup.unh

Process Header

```
PROC pony.startup.unh
            (VAL INT net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT PONY.NETHANDLE! net.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns an unshared network-handle, as well as the ID of the own node.

Parameters

Parameter	**Description**
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (unshared).
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.snh

Process Header

```
PROC pony.startup.snh
            (VAL INT net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT SHARED PONY.NETHANDLE! net.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns a shared network-handle, as well as the ID of the own node.

Parameters

Parameter	**Description**
net.type	Network-type. For possible values see Appendix B.2.3.
ans.name	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
app.name	Name of the application. Empty string not allowed.
node.name	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
node.type	Node-type. For possible values see Appendix B.2.4.
own.node.id	Node-ID of the own node.
net.handle	Network-handle (shared).
result	Result. For possible values see Appendix B.2.6.

pony.startup.unh.ueh

Process Header

```
PROC pony.startup.unh.ueh
            (VAL INT net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT PONY.NETHANDLE! net.handle,
             RESULT PONY.ERRHANDLE! err.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns an unshared network-handle and an unshared error-handle, as well as the ID of the own node.

Parameters

Parameter	Description
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (unshared).
`err.handle`	Error-handle (unshared).
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.unh.ueh.iep

Process Header

```
PROC pony.startup.unh.ueh.iep
          (VAL INT net.type,
           VAL []BYTE ans.name, app.name, node.name,
           VAL INT node.type,
           RESULT INT own.node.id,
           RESULT PONY.NETHANDLE! net.handle,
           RESULT PONY.ERRHANDLE! err.handle,
           RESULT INT err.point,
           RESULT INT result)
```

Description

Starts the pony environment. On success, returns an unshared network-handle, an unshared error-handle and an initial error-point, as well as the ID of the own node.

Parameters

Parameter	Description
net.type	Network-type. For possible values see Appendix B.2.3.
ans.name	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
app.name	Name of the application. Empty string not allowed.
node.name	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
node.type	Node-type. For possible values see Appendix B.2.4.
own.node.id	Node-ID of the own node.
net.handle	Network-handle (unshared).
err.handle	Error-handle (unshared).
err.point	Initial error-point.
result	Result. For possible values see Appendix B.2.6.

pony.startup.unh.seh

Process Header

```
PROC pony.startup.unh.seh
            (VAL INT net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT PONY.NETHANDLE! net.handle,
             RESULT SHARED PONY.ERRHANDLE! err.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns an unshared network-handle and a shared error-handle, as well as the ID of the own node.

Parameters

Parameter	Description
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (unshared).
`err.handle`	Error-handle (shared).
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.unh.seh.iep

Process Header

```
PROC pony.startup.unh.seh.iep
          (VAL INT net.type,
           VAL []BYTE ans.name, app.name, node.name,
           VAL INT node.type,
           RESULT INT own.node.id,
           RESULT PONY.NETHANDLE! net.handle,
           RESULT SHARED PONY.ERRHANDLE! err.handle,
           RESULT INT err.point,
           RESULT INT result)
```

Description

Starts the pony environment. On success, returns an unshared network-handle, a shared error-handle and an initial error-point, as well as the ID of the own node.

Parameters

Parameter	Description
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (unshared).
`err.handle`	Error-handle (shared).
`err.point`	Initial error-point.
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.snh.ueh

Process Header

```
PROC pony.startup.snh.ueh
            (VAL INT net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT SHARED PONY.NETHANDLE! net.handle,
             RESULT PONY.ERRHANDLE! err.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns a shared network-handle and an unshared error-handle, as well as the ID of the own node.

Parameters

Parameter	**Description**
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (shared).
`err.handle`	Error-handle (unshared).
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.snh.ueh.iep

Process Header

```
PROC pony.startup.snh.ueh.iep
          (VAL INT net.type,
           VAL []BYTE ans.name, app.name, node.name,
           VAL INT node.type,
           RESULT INT own.node.id,
           RESULT SHARED PONY.NETHANDLE! net.handle,
           RESULT PONY.ERRHANDLE! err.handle,
           RESULT INT err.point,
           RESULT INT result)
```

Description

Starts the pony environment. On success, returns a shared network-handle, an unshared error-handle and an initial error-point, as well as the ID of the own node.

Parameters

Parameter	Description
net.type	Network-type. For possible values see Appendix B.2.3.
ans.name	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
app.name	Name of the application. Empty string not allowed.
node.name	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
node.type	Node-type. For possible values see Appendix B.2.4.
own.node.id	Node-ID of the own node.
net.handle	Network-handle (shared).
err.handle	Error-handle (unshared).
err.point	Initial error-point.
result	Result. For possible values see Appendix B.2.6.

pony.startup.snh.seh

Process Header

```
PROC pony.startup.snh.seh
            (VAL INT net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT SHARED PONY.NETHANDLE! net.handle,
             RESULT SHARED PONY.ERRHANDLE! err.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns a shared network-handle and a shared error-handle, as well as the ID of the own node.

Parameters

Parameter	Description
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (shared).
`err.handle`	Error-handle (shared).
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.snh.seh.iep

Process Header

```
PROC pony.startup.snh.seh.iep
          (VAL INT net.type,
           VAL []BYTE ans.name, app.name, node.name,
           VAL INT node.type,
           RESULT INT own.node.id,
           RESULT SHARED PONY.NETHANDLE! net.handle,
           RESULT SHARED PONY.ERRHANDLE! err.handle,
           RESULT INT err.point,
           RESULT INT result)
```

Description

Starts the pony environment. On success, returns a shared network-handle, a shared error-handle and an initial error-point, as well as the ID of the own node.

Parameters

Parameter	Description
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (shared).
`err.handle`	Error-handle (shared).
`err.point`	Initial error-point.
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.unh.mh

Process Header

```
PROC pony.startup.unh.mh
           (VAL INT msg.type, net.type,
            VAL []BYTE ans.name, app.name, node.name,
            VAL INT node.type,
            RESULT INT own.node.id,
            RESULT PONY.NETHANDLE! net.handle,
            RESULT PONY.MSGHANDLE! msg.handle,
            RESULT INT result)
```

Description

Starts the pony environment. On success, returns an unshared network-handle and a message-handle, as well as the ID of the own node.

Parameters

Parameter	Description
msg.type	Message-type. Determines which messages are output. For possible values see Appendix B.2.2.
net.type	Network-type. For possible values see Appendix B.2.3.
ans.name	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
app.name	Name of the application. Empty string not allowed.
node.name	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
node.type	Node-type. For possible values see Appendix B.2.4.
own.node.id	Node-ID of the own node.
net.handle	Network-handle (unshared).
msg.handle	Message-handle.
result	Result. For possible values see Appendix B.2.6.

pony.startup.snh.mh

Process Header

```
PROC pony.startup.snh.mh
            (VAL INT msg.type, net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT SHARED PONY.NETHANDLE! net.handle,
             RESULT PONY.MSGHANDLE! msg.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns a shared network-handle and a message-handle, as well as the ID of the own node.

Parameters

Parameter	Description
`msg.type`	Message-type. Determines which messages are output. For possible values see Appendix B.2.2.
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (shared).
`msg.handle`	Message-handle.
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.unh.ueh.mh

Process Header

```
PROC pony.startup.unh.ueh.mh
            (VAL INT msg.type, net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT PONY.NETHANDLE! net.handle,
             RESULT PONY.ERRHANDLE! err.handle,
             RESULT PONY.MSGHANDLE! msg.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns an unshared network-handle, an unshared error-handle and a message-handle, as well as the ID of the own node.

Parameters

Parameter	Description
`msg.type`	Message-type. Determines which messages are output. For possible values see Appendix B.2.2.
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (unshared).
`err.handle`	Error-handle (unshared).
`msg.handle`	Message-handle.
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.unh.ueh.iep.mh

Process Header

```
PROC pony.startup.unh.ueh.iep.mh
            (VAL INT msg.type, net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT PONY.NETHANDLE! net.handle,
             RESULT PONY.ERRHANDLE! err.handle,
             RESULT INT err.point,
             RESULT PONY.MSGHANDLE! msg.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns an unshared network-handle, an unshared error-handle, an initial error-point and a message-handle, as well as the ID of the own node.

Parameters

Parameter	Description
msg.type	Message-type. Determines which messages are output. For possible values see Appendix B.2.2.
net.type	Network-type. For possible values see Appendix B.2.3.
ans.name	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
app.name	Name of the application. Empty string not allowed.
node.name	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
node.type	Node-type. For possible values see Appendix B.2.4.
own.node.id	Node-ID of the own node.
net.handle	Network-handle (unshared).
err.handle	Error-handle (unshared).
err.point	Initial error-point.
msg.handle	Message-handle.
result	Result. For possible values see Appendix B.2.6.

pony.startup.unh.seh.mh

Process Header

```
PROC pony.startup.unh.seh.mh
              (VAL INT msg.type, net.type,
               VAL []BYTE ans.name, app.name, node.name,
               VAL INT node.type,
               RESULT INT own.node.id,
               RESULT PONY.NETHANDLE! net.handle,
               RESULT SHARED PONY.ERRHANDLE! err.handle,
               RESULT PONY.MSGHANDLE! msg.handle,
               RESULT INT result)
```

Description

Starts the pony environment. On success, returns an unshared network-handle, a shared error-handle and a message-handle, as well as the ID of the own node.

Parameters

Parameter	**Description**
`msg.type`	Message-type. Determines which messages are output. For possible values see Appendix B.2.2.
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (unshared).
`err.handle`	Error-handle (shared).
`msg.handle`	Message-handle.
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.unh.seh.iep.mh

Process Header

```
PROC pony.startup.unh.seh.iep.mh
            (VAL INT msg.type, net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT PONY.NETHANDLE! net.handle,
             RESULT SHARED PONY.ERRHANDLE! err.handle,
             RESULT INT err.point,
             RESULT PONY.MSGHANDLE! msg.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns an unshared network-handle, a shared error-handle, an initial error-point and a message-handle, as well as the ID of the own node.

Parameters

Parameter	Description
`msg.type`	Message-type. Determines which messages are output. For possible values see Appendix B.2.2.
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (unshared).
`err.handle`	Error-handle (shared).
`err.point`	Initial error-point.
`msg.handle`	Message-handle.
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.snh.ueh.mh

Process Header

```
PROC pony.startup.snh.ueh.mh
            (VAL INT msg.type, net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT SHARED PONY.NETHANDLE! net.handle,
             RESULT PONY.ERRHANDLE! err.handle,
             RESULT PONY.MSGHANDLE! msg.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns a shared network-handle, an unshared error-handle and a message-handle, as well as the ID of the own node.

Parameters

Parameter	Description
msg.type	Message-type. Determines which messages are output. For possible values see Appendix B.2.2.
net.type	Network-type. For possible values see Appendix B.2.3.
ans.name	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
app.name	Name of the application. Empty string not allowed.
node.name	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
node.type	Node-type. For possible values see Appendix B.2.4.
own.node.id	Node-ID of the own node.
net.handle	Network-handle (shared).
err.handle	Error-handle (unshared).
msg.handle	Message-handle.
result	Result. For possible values see Appendix B.2.6.

pony.startup.snh.ueh.iep.mh

Process Header

```
PROC pony.startup.snh.ueh.iep.mh
            (VAL INT msg.type, net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT SHARED PONY.NETHANDLE! net.handle,
             RESULT PONY.ERRHANDLE! err.handle,
             RESULT INT err.point,
             RESULT PONY.MSGHANDLE! msg.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns a shared network-handle, an unshared error-handle, an initial error-point and a message-handle, as well as the ID of the own node.

Parameters

Parameter	**Description**
`msg.type`	Message-type. Determines which messages are output. For possible values see Appendix B.2.2.
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (shared).
`err.handle`	Error-handle (unshared).
`err.point`	Initial error-point.
`msg.handle`	Message-handle.
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.snh.seh.mh

Process Header

```
PROC pony.startup.snh.seh.mh
            (VAL INT msg.type, net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT SHARED PONY.NETHANDLE! net.handle,
             RESULT SHARED PONY.ERRHANDLE! err.handle,
             RESULT PONY.MSGHANDLE! msg.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns a shared network-handle, a shared error-handle and a message-handle, as well as the ID of the own node.

Parameters

Parameter	Description
`msg.type`	Message-type. Determines which messages are output. For possible values see Appendix B.2.2.
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (shared).
`err.handle`	Error-handle (shared).
`msg.handle`	Message-handle.
`result`	Result. For possible values see Appendix B.2.6.

pony.startup.snh.seh.iep.mh

Process Header

```
PROC pony.startup.snh.seh.iep.mh
            (VAL INT msg.type, net.type,
             VAL []BYTE ans.name, app.name, node.name,
             VAL INT node.type,
             RESULT INT own.node.id,
             RESULT SHARED PONY.NETHANDLE! net.handle,
             RESULT SHARED PONY.ERRHANDLE! err.handle,
             RESULT INT err.point,
             RESULT PONY.MSGHANDLE! msg.handle,
             RESULT INT result)
```

Description

Starts the pony environment. On success, returns a shared network-handle, a shared error-handle, an initial error-point and a message-handle, as well as the ID of the own node.

Parameters

Parameter	Description
`msg.type`	Message-type. Determines which messages are output. For possible values see Appendix B.2.2.
`net.type`	Network-type. For possible values see Appendix B.2.3.
`ans.name`	Name of the ANS. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`app.name`	Name of the application. Empty string not allowed.
`node.name`	Name of the node. Allowed characters: letters, digits, dash, dot, underscore (or empty string).
`node.type`	Node-type. For possible values see Appendix B.2.4.
`own.node.id`	Node-ID of the own node.
`net.handle`	Network-handle (shared).
`err.handle`	Error-handle (shared).
`err.point`	Initial error-point.
`msg.handle`	Message-handle.
`result`	Result. For possible values see Appendix B.2.6.

B.1.2 Allocation Processes

pony.alloc.uc

Process Header

```
PROC pony.alloc.uc
           (PONY.NETHANDLE! net.handle,
            VAL []BYTE nct.name, VAL INT svr.type,
            RESULT MOBILE.CHAN! cli, RESULT INT result)
```

Description

Allocates an unshared client-end of an NCT. On success, returns the allocated NCT-end.

Parameters

Parameter	**Description**
`net.handle`	The network-handle.
`nct.name`	Name of the NCT. Empty string not allowed.
`svr.type`	Share-type of the server-end of the NCT. For possible values see Appendix B.2.5.
`cli`	The (unshared) client-end variable to be allocated.
`result`	Result. For possible values see Appendix B.2.7.

pony.alloc.sc

Process Header

```
PROC pony.alloc.sc
            (PONY.NETHANDLE! net.handle,
             VAL []BYTE nct.name, VAL INT svr.type,
             RESULT SHARED MOBILE.CHAN! cli, RESULT INT result)
```

Description

Allocates a shared client-end of an NCT. On success, returns the allocated NCT-end.

Parameters

Parameter	**Description**
`net.handle`	The network-handle.
`nct.name`	Name of the NCT. Empty string not allowed.
`svr.type`	Share-type of the server-end of the NCT. For possible values see Appendix B.2.5.
`cli`	The (shared) client-end variable to be allocated.
`result`	Result. For possible values see Appendix B.2.7.

pony.alloc.us

Process Header

```
PROC pony.alloc.us
            (PONY.NETHANDLE! net.handle,
             VAL []BYTE nct.name, VAL INT cli.type,
             RESULT MOBILE.CHAN? svr, RESULT INT result)
```

Description

Allocates an unshared server-end of an NCT. On success, returns the allocated NCT-end.

Parameters

Parameter	**Description**
`net.handle`	The network-handle.
`nct.name`	Name of the NCT. Empty string not allowed.
`cli.type`	Share-type of the client-end of the NCT. For possible values see Appendix B.2.5.
`svr`	The (unshared) server-end variable to be allocated.
`result`	Result. For possible values see Appendix B.2.7.

pony.alloc.ss

Process Header

```
PROC pony.alloc.ss
            (PONY.NETHANDLE! net.handle,
             VAL []BYTE nct.name, VAL INT cli.type,
             RESULT SHARED MOBILE.CHAN? svr, RESULT INT result)
```

Description

Allocates a shared server-end of an NCT. On success, returns the allocated NCT-end.

Parameters

Parameter	**Description**
`net.handle`	The network-handle.
`nct.name`	Name of the NCT. Empty string not allowed.
`cli.type`	Share-type of the client-end of the NCT. For possible values see Appendix B.2.5.
`svr`	The (shared) server-end variable to be allocated.
`result`	Result. For possible values see Appendix B.2.7.

B.1.3 The Shutdown Process

pony.shutdown

Process Header

```
PROC pony.shutdown (PONY.NETHANDLE! net.handle)
```

Description

Shuts down the pony environment. Must be called after all activity on (possibly) networked channel-types has ceased.

Parameters

Parameter	Description
`net.handle`	The network-handle.

B.1.4 Error-handling Processes

pony.err.get.nct.id.uc

Process Header

```
PROC pony.err.get.nct.id.uc
            (MOBILE.CHAN! cli,
             RESULT INT nct.id, result)
```

Description

Returns the NCT-ID for a given channel-type-end if the channel-type is networked (version for unshared client-ends).

Parameters

Parameter	**Description**
`cli`	The (unshared) client-end to be checked.
`nct.id`	NCT-ID of the channel-type.
`result`	Result. For possible values see Appendix B.2.8.

pony.err.get.nct.id.sc

Process Header

```
PROC pony.err.get.nct.id.sc
            (SHARED MOBILE.CHAN! cli,
             RESULT INT nct.id, result)
```

Description

Returns the NCT-ID for a given channel-type-end if the channel-type is networked (version for shared client-ends).

Parameters

Parameter	**Description**
`cli`	The (shared) client-end to be checked.
`nct.id`	NCT-ID of the channel-type.
`result`	Result. For possible values see Appendix B.2.8.

pony.err.get.nct.id.us

Process Header

```
PROC pony.err.get.nct.id.us
          (MOBILE.CHAN? svr,
           RESULT INT nct.id, result)
```

Description

Returns the NCT-ID for a given channel-type-end if the channel-type is networked (version for unshared server-ends).

Parameters

Parameter	Description
svr	The (unshared) server-end to be checked.
nct.id	NCT-ID of the channel-type.
result	Result. For possible values see Appendix B.2.8.

pony.err.get.nct.id.ss

Process Header

```
PROC pony.err.get.nct.id.ss
            (SHARED MOBILE.CHAN? svr,
             RESULT INT nct.id, result)
```

Description

Returns the NCT-ID for a given channel-type-end if the channel-type is networked (version for shared server-ends).

Parameters

Parameter	Description
svr	The (shared) server-end to be checked.
nct.id	NCT-ID of the channel-type.
result	Result. For possible values see Appendix B.2.8.

pony.err.get.current.remote.node

Process Header

```
PROC pony.err.get.current.remote.node
            (PONY.ERRHANDLE! err.handle,
             VAL INT nct.id,
             RESULT INT remote.node.id, result)
```

Description

Returns the ID of the current remote node for a given NCT-ID if the NCT is currently in a session that involves a remote node.

Parameters

Parameter	Description
`err.handle`	The error-handle.
`nct.id`	NCT-ID to be checked.
`remote.node.id`	ID of current remote node.
`result`	Result. For possible values see Appendix B.2.8.

pony.err.new.err.point

Process Header

```
PROC pony.err.new.err.point
            (PONY.ERRHANDLE! err.handle,
             RESULT INT err.point)
```

Description

Gets a new error-point from the error-handler.

Parameters

Parameter	Description
err.handle	The error-handle.
err.point	New error-point.

pony.err.delete.err.point

Process Header

```
PROC pony.err.delete.err.point
            (PONY.ERRHANDLE! err.handle,
             VAL INT err.point,
             RESULT INT result)
```

Description

Deletes a given error-point.

Parameters

Parameter	Description
err.handle	The error-handle.
err.point	Error-point to be deleted.
result	Result. For possible values see Appendix B.2.8.

pony.err.get.errs.after

Process Header

```
PROC pony.err.get.errs.after
            (PONY.ERRHANDLE! err.handle,
             INT err.point,
             VAL BOOL check.ans, check.master, check.all.nodes,
             VAL []INT nodes.to.check,
             RESULT MOBILE []PONY.ERROR err.array,
             RESULT INT result)
```

Description

Returns all errors that occurred after a given error-point which meet the given criteria.

Parameters

Parameter	Description
err.handle	The error-handle.
err.point	Error-point.
check.ans	'TRUE' in order to return errors that involve the ANS.
check.master	'TRUE' in order to return errors that involve the master node of the application.
check.all.nodes	'TRUE' in order to return errors that involve any remote node.
nodes.to.check	Array of node-IDs. Process returns errors that involve any of the nodes in 'nodes.to.check'.
err.array	Array of returned errors. For details about the 'PONY.ERROR' data-type see Appendix B.2.1.
result	Result. For possible values see Appendix B.2.8.

pony.err.shutdown

Process Header

```
PROC pony.err.shutdown (PONY.ERRHANDLE! err.handle)
```

Description

Shuts down the error-handler.

Parameters

Parameter	Description
err.handle	The error-handle.

B.1.5 The Message-outputters

pony.msg.out.uo

Process Header

```
PROC pony.msg.out.uo
          (PONY.MSGHANDLE! msg.handle,
           CHAN BYTE out!)
```

Description

Outputs status messages from the pony environment (version for an unshared output channel). To be run in parallel with everything else that involves the pony environment.

Parameters

Parameter	Description
`msg.handle`	The message-handle.
`out`	(Unshared) output channel — typically ‘`stdout`’.

pony.msg.out.so

Process Header

```
PROC pony.msg.out.so
          (PONY.MSGHANDLE! msg.handle,
           SHARED CHAN BYTE out!)
```

Description

Outputs status messages from the pony environment (version for a shared output channel). To be run in parallel with everything else that involves the pony environment.

Parameters

Parameter	**Description**
`msg.handle`	The message-handle.
`out`	(Shared) output channel — typically '`stdout`'.

pony.msg.out.ue

Process Header

```
PROC pony.msg.out.ue
            (PONY.MSGHANDLE! msg.handle,
             CHAN BYTE err!)
```

Description

Outputs error messages from the pony environment (version for an unshared error channel). To be run in parallel with everything else that involves the pony environment.

Parameters

Parameter	Description
msg.handle	The message-handle.
err	(Unshared) error channel — typically 'stderr'.

pony.msg.out.se

Process Header

```
PROC pony.msg.out.se
           (PONY.MSGHANDLE! msg.handle,
            SHARED CHAN BYTE err!)
```

Description

Outputs error messages from the pony environment (version for a shared error channel). To be run in parallel with everything else that involves the pony environment.

Parameters

Parameter	**Description**
msg.handle	The message-handle.
err	(Shared) error channel — typically ‘stderr’.

pony.msg.out.uo.ue

Process Header

```
PROC pony.msg.out.uo.ue
          (PONY.MSGHANDLE! msg.handle,
           CHAN BYTE out!, err!)
```

Description

Outputs status and error messages from the pony environment (version for an unshared output channel and an unshared error channel). To be run in parallel with everything else that involves the pony environment.

Parameters

Parameter	**Description**
`msg.handle`	The message-handle.
`out`	(Unshared) output channel — typically '`stdout`'.
`err`	(Unshared) error channel — typically '`stderr`'.

pony.msg.out.so.ue

Process Header

```
PROC pony.msg.out.so.ue
            (PONY.MSGHANDLE! msg.handle,
             SHARED CHAN BYTE out!, CHAN BYTE err!)
```

Description

Outputs status and error messages from the pony environment (version for a shared output channel and an unshared error channel). To be run in parallel with everything else that involves the pony environment.

Parameters

Parameter	Description
msg.handle	The message-handle.
out	(Shared) output channel — typically 'stdout'.
err	(Unshared) error channel — typically 'stderr'.

pony.msg.out.uo.se

Process Header

```
PROC pony.msg.out.uo.se
            (PONY.MSGHANDLE! msg.handle,
             CHAN BYTE out!, SHARED CHAN BYTE err!)
```

Description

Outputs status and error messages from the pony environment (version for an unshared output channel and a shared error channel). To be run in parallel with everything else that involves the pony environment.

Parameters

Parameter	Description
msg.handle	The message-handle.
out	(Unshared) output channel — typically 'stdout'.
err	(Shared) error channel — typically 'stderr'.

pony.msg.out.so.se

Process Header

```
PROC pony.msg.out.so.se
          (PONY.MSGHANDLE! msg.handle,
           SHARED CHAN BYTE out!, err!)
```

Description

Outputs status and error messages from the pony environment (version for a shared output channel and a shared error channel). To be run in parallel with everything else that involves the pony environment.

Parameters

Parameter	**Description**
`msg.handle`	The message-handle.
`out`	(Shared) output channel — typically '`stdout`'.
`err`	(Shared) error channel — typically '`stderr`'.

B.2 Public pony Data-types and Constants

This section contains a reference of all public data-types and constants that the pony environment uses via its public processes.

B.2.1 The Error Record

PONY.ERROR

Declaration

```
DATA TYPE PONY.ERROR
  RECORD
    BOOL ans.concerned:
    BOOL master.concerned:
    BOOL remote.node.concerned:
    INT remote.node.id:
    INT err.code:
:
```

Description

Represents a networking error in pony. The fields contain specific information about the error.

Fields

Field	**Description**
`ans.concerned`	'`TRUE`' if the error involves the ANS.
`master.concerned`	'`TRUE`' if the error involves the master node of the application.
`remote.node.concerned`	'`TRUE`' if the error involves a remote node.
`remote.node.id`	ID of the remote node involved in the error. Only relevant if '`remote.node.concerned`' is '`TRUE`'.
`err.code`	Error-code. Dependent on the network-type. For possible TCP/IP error-codes see Appendix B.2.9.

B.2.2 Message-types

Prefix: `PONYC.MSGTYPE.*`

Suffix	Value	Description
STATUS	0	Display status messages only.
ERR	1	Display error messages only.
STATUSERR	2	Display status and error messages.

B.2.3 Network-types

Prefix: `PONYC.NETTYPE.*`

Suffix	Value	Description
TCPIP	0	TCP/IP. (Currently the only supported network-type.)

B.2.4 Node-types

Prefix: `PONYC.NODETYPE.*`

Suffix	Value	Description
MASTER	0	Master node.
MASTERRESET	1	Master node. Reset application if another master is stored in the ANS already.
SLAVE	2	Slave node. Fails if there is no master stored in the ANS yet.
SLAVEWAIT	3	Slave node. If there is no master yet, wait until one contacts the ANS.

B.2.5 Share-types

Prefix: `PONYC.SHARETYPE.*`

Suffix	Value	Description
UNKNOWN	0	Share-type unknown or not specified.
UNSHARED	1	NCT-end is unshared.
SHARED	2	NCT-end is shared.

B.2.6 Results for Startup Processes

Non-network-type-specific Results

Prefix: PONYC.RESULT.STARTUP.*

Suffix	Value	Description
OK	0	Completed successfully.
ILLEGALMSGTYPE	-1	Illegal message-type.
ILLEGALNETTYPE	-2	Illegal network-type.

Network-type-specific Results for TCP/IP

Prefix: PONYC.RESULT.STARTUP.TCPIP.*

Suffix	Value	Description
ILLEGALANSNAME	-11	Illegal ANS-name.
ILLEGALAPPNAME	-12	Illegal application-name.
ILLEGALNODENAME	-13	Illegal node-name.
ILLEGALNODETYPE	-14	Illegal node-type.
HOMEDIRNOTINENV	-15	Home directory not specified in environment.
ANSFILEACCESSERROR	-16	ANS-file not readable.
ANSFILEINVALIDSETTINGS	-17	ANS-file contains invalid settings.
NODEFILEACCESSERROR	-18	Node-file not readable.
NODEFILEINVALIDSETTINGS	-19	Node-file contains invalid settings.
IPADDRNOTRESOLVED	-20	IP address could not be resolved.
LISTENSOCKFAILURE	-21	Network failure when creating listening socket.
ANSNETFAILURE	-22	Network failure when connecting to ANS.
DUPLICATEMASTER	-23	Node's location is duplicate of master's location stored in ANS.
DUPLICATESLAVE	-24	Node's location is duplicate of a pending slave's location stored in ANS or of another slave's location stored by master.
OTHERMASTER	-25	Master tries to log on to ANS, other master already there.
NOMASTERYET	-26	Slave tries to log on to ANS, no master there yet.
MASTERNETFAILURE	-27	Network failure when slave tries to connect to master.
MASTERSHUTTINGDOWN	-28	Slave tries to log on to master, master is currently shutting down.

B.2.7 Results for Allocation Processes

Prefix: `PONYC.RESULT.ALLOC.*`

Suffix	Value	Description
OK	0	Completed successfully.
ILLEGALNCTNAME	-1	Illegal NCT-name.
CHANTYPEMISMATCH	-2	Type mismatch between NCT-end to be allocated and existing NCT of the same name.
X2XTYPEMISMATCH	-3	Mismatch of x2x-type.
X2XCOUNTMISMATCH	-4	Mismatch of x2x-count (i.e. trying to allocate more than one one2x client-end or more than one x2one server-end).

B.2.8 Results for Error-handling Processes

Results for 'pony.err.get.nct.id.*'

Prefix: PONYC.RESULT.ERR.GNI.*

Suffix	Value	Description
OK	0	Completed successfully.
CTENDUNDEFINED	-1	Channel-type-end undefined.
CTENDNOTNETWORKED	-2	Channel-type-end not networked.

Results for 'pony.err.get.current.remote.node'

Prefix: PONYC.RESULT.ERR.GCRN.*

Suffix	Value	Description
OK	0	Completed successfully.
INVALIDNCTID	-1	Invalid NCT-ID.
NOSESSION	-2	Currently no session.
SAMENODE	-3	Currently both ends on the same node (internal session).

Results for 'pony.err.delete.error.point'

Prefix: PONYC.RESULT.ERR.DEP.*

Suffix	Value	Description
OK	0	Completed successfully.
INVALIDERRPOINT	-1	Invalid error-point.

Results for 'pony.err.get.errors.after'

Prefix: PONYC.RESULT.ERR.GEA.*

Suffix	Value	Description
OK	0	Completed successfully.
INVALIDERRPOINT	-1	Invalid error-point.

B.2.9 Error-codes for TCP/IP

Prefix: `PONYC.ERRCODE.TCPIP.*`

Suffix	Value	Description
ACCEPTFAILURE	-1	Accepting new connection failed.
CONNECTFAILURE	-2	Connecting failed.
SETNODELAYFAILURE	-3	Setting no-delay option (turning off Nagle algorithm) failed.
READFAILURE	-4	Read operation failed.
WRITEFAILURE	-5	Write operation failed.

Appendix C

Different 'commstime' Implementations

C.1 The Traditional 'commstime' Implementation

This section contains the traditional, non-distributed, implementation of the classical occam 'commstime' benchmark.

```
PROC commstime (CHAN BYTE key?, scr!, err!)
  BOOL use.seq.delta:
  INT num.loops:
  SEQ
    ...  Find out whether to use the sequential or the parallel delta
    ...  Find out the number of loops
    -- Channels between the processes
    CHAN INT a, b, c, d:
    -- Run sub-processes in parallel
    PAR
      prefix (0, b?, a!)
      IF
        use.seq.delta
          -- Sequential delta
          seq.delta (a?, c!, d!)
        TRUE
          -- Parallel delta
          delta (a?, c!, d!)
      succ (c?, b!)
      -- Monitoring process
      consume (num.loops, d?, scr!)
:
```

C.2 The Distributed 'commstime' Implementation

This section contains the distributed implementation of the 'commstime' benchmark. There are four different nodes, each of which is running one of the sub-processes. A complete distributed 'commstime' application consists of all four nodes.

C.2.1 The Channel-type Declaration

```
-- Channel-type with one INT channel
CHAN TYPE INT.CT
  MOBILE RECORD
    CHAN INT chan?:
:
```

C.2.2 The 'prefix' Node

```
PROC commstime.prefix (CHAN BYTE key?, scr!, err!)
  -- Network-handle
  PONY.NETHANDLE! net.handle:
  -- NCT-end variables
  INT.CT? b.svr:
  INT.CT! a.cli:
  -- Other variables
  INT own.node.id, result:
  SEQ
    -- Start pony
    pony.startup.unh (PONYC.NETTYPE.TCPIP, "", "commstime",
                      "", PONYC.NODETYPE.SLAVEWAIT,
                      own.node.id, net.handle, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    -- Allocate NCT-ends
    pony.alloc.us (net.handle, "b", PONYC.SHARETYPE.UNSHARED,
                   b.svr, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    pony.alloc.uc (net.handle, "a", PONYC.SHARETYPE.UNSHARED,
                   a.cli, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    -- Start sub-process
    prefix (0, b.svr[chan], a.cli[chan])
    -- No shutdown of pony here
    --   because the sub-process that was started is running infinitely
 :
```

C.2.3 The 'delta' Node

```
PROC commstime.delta (CHAN BYTE key?, scr!, err!)
  -- Network-handle
  PONY.NETHANDLE! net.handle:
  -- NCT-end variables
  INT.CT? a.svr:
  INT.CT! c.cli, d.cli:
  -- Other variables
  BOOL use.seq.delta:
  INT own.node.id, result:
  SEQ
    ...  Find out whether to use the sequential or the parallel delta
    -- Start pony
    pony.startup.unh (PONYC.NETTYPE.TCPIP, "", "commstime",
                      "", PONYC.NODETYPE.SLAVEWAIT,
                      own.node.id, net.handle, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    -- Allocate NCT-ends
    pony.alloc.us (net.handle, "a", PONYC.SHARETYPE.UNSHARED,
                   a.svr, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    pony.alloc.uc (net.handle, "c", PONYC.SHARETYPE.UNSHARED,
                   c.cli, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    pony.alloc.uc (net.handle, "d", PONYC.SHARETYPE.UNSHARED,
                   d.cli, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    -- Start sub-process
    IF
      use.seq.delta
        -- Sequential delta
        seq.delta (a.svr[chan], c.cli[chan], d.cli[chan])
      TRUE
        -- Parallel delta
        delta (a.svr[chan], c.cli[chan], d.cli[chan])
    -- No shutdown of pony here
    --   because the sub-process that was started is running infinitely
 :
```

C.2.4 The 'succ' Node

```
PROC commstime.succ (CHAN BYTE key?, scr!, err!)
  -- Network-handle
  PONY.NETHANDLE! net.handle:
  -- NCT-end variables
  INT.CT? c.svr:
  INT.CT! b.cli:
  -- Other variables
  INT own.node.id, result:
  SEQ
    -- Start pony
    pony.startup.unh (PONYC.NETTYPE.TCPIP, "", "commstime",
                      "", PONYC.NODETYPE.SLAVEWAIT,
                      own.node.id, net.handle, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    -- Allocate NCT-ends
    pony.alloc.us (net.handle, "c", PONYC.SHARETYPE.UNSHARED,
                   c.svr, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    pony.alloc.uc (net.handle, "b", PONYC.SHARETYPE.UNSHARED,
                   b.cli, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    -- Start sub-process
    succ (c.svr[chan], b.cli[chan])
    -- No shutdown of pony here
    --   because the sub-process that was started is running infinitely
:
```

C.2.5 The 'consume' Node

```
PROC commstime.consume (CHAN BYTE key?, scr!, err!)
  -- Network-handle
  PONY.NETHANDLE! net.handle:
  -- NCT-end variable
  INT.CT? d.svr:
  -- Other variables
  INT num.loops, own.node.id, result:
  SEQ
    ...  Find out the number of loops
    -- Start pony
    pony.startup.unh (PONYC.NETTYPE.TCPIP, "", "commstime",
                      "", PONYC.NODETYPE.MASTERRESET,
                      own.node.id, net.handle, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    -- Allocate NCT-end
    pony.alloc.us (net.handle, "d", PONYC.SHARETYPE.UNSHARED,
                   d.svr, result)
    ASSERT (result = PONYC.RESULT.STARTUP.OK)
    -- Start sub-process (monitoring process)
    consume (num.loops, d.svr[chan], scr!)
    -- No shutdown of pony here
    --   because the sub-process that was started is running infinitely
 :
```

Appendix D

Own Publications

The following list contains the author's publications that are related to pony and its development:

- M. Schweigler. The Distributed occam Protocol – A New Layer On Top Of TCP/IP To Serve occam Channels Over The Internet. Master's thesis, Computing Laboratory, University of Kent at Canterbury, September 2001. MSc Dissertation.

- M. Schweigler, F.R.M. Barnes, and P.H. Welch. Flexible, Transparent and Dynamic occam Networking With KRoC.net. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 199–224, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN: 1-58603-381-6.

- M. Schweigler. Adding Mobility to Networked Channel-Types. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575, pages 107–126, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

- M. Schweigler and A.T. Sampson. pony – The occam-π Network Environment. In Peter Welch, Jon Kerridge, and Fred Barnes, editors, *Communicating Process Architectures 2006*, WoTUG-29, Concurrent Systems Engineering, ISSN 1383-7575, pages 77–108, Amsterdam, The Netherlands, September 2006. IOS Press. ISBN: 1-58603-671-8.

All publications are available on-line at:

`http://www.informatico.de/publications/`