Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

pony – The occam-π Network Environment

Mario SCHWEIGLER
ms44@kent.ac.uk / research@informatico.de

Adam SAMPSON
ats1@kent.ac.uk / ats@offog.org

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, UK

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 paper 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 first part of this paper discusses the philosophy behind pony, explains how it is used, and gives a brief overview of its implementation. The second part evaluates pony's performance by presenting a number of benchmarks.

Keywords. pony, occam-pi, KRoC, CSP, concurrency, networking, unified model, inter-processor, intra-processor, benchmarks

Introduction

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 [1] and the π-calculus [2], with their notion of processes and channels, are particularly suited to model the 'real world', especially since there is a programming language available that is based on those formal calculi, but still easy to understand and to use. This language is occam-π, the new dynamic version of the classical occam[1] [3]. Originally targeted at transputer [4] 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 the advent of

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

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 [5,6,7] 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 pony[2] project [8] 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. Background and Motivation

1.1. 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] [9]. 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.

1.2. 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.

1.2.1. 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

[2]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.

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

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 [10], which was the last transputer being developed and which went out of production shortly after having been introduced by Inmos, additionally offered a *Virtual Channel Processor (VCP)* [11] 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' [12] 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 [13]. 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 [14,9,15]. 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 are preserved as well, both when mobile data [14] 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 like on the transputer.

1.2.2. 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.

1.3. 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 [16]. In autumn 2001, the first major version was released as part of an MSc dissertation under the name 'Distributed occam Protocol' [17]. 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 a PhD[4] [18,19,8]. During this time, the occam language was extended significantly[5], adding many dynamic features. This 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.

The mobility of ends of network-channel-types was inspired by the mobile channels in Muller and May's Icarus language [20]. 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.

1.4. Structure of This Paper

Section 2 introduces the terminology used in this paper and presents the architecture of the pony environment. Sections 3 through 5 discuss the characteristics of pony nodes, their startup, and the startup of the Application Name Server. The allocation of NCTs is covered in Section 6, the shutdown of pony in Section 7. Section 8 is concerned with the configuration of the pony environment.

Section 9 outlines a sample pony application. A brief overview of the implementation of pony is given in Section 10. Section 11 presents a number of benchmarks that were carried out to examine pony's performance. Section 12 concludes with a discussion of the work presented in this paper, along with an outline of possible future research.

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

2. Architecture and Terminology

2.1. Applications and Nodes

A group of occam-π programs which are interconnected by the pony infrastructure is called a pony *application*. 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).

2.2. 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.

2.3. 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 [21,22], 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.

2.4. 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.

2.5. Variants of Channel-types and Their Graphical Representation

For the remainder of this paper, 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. Please note that the x2x-type is a property of concrete instances of a channel-type, not of its type declaration.

Figure 1 shows how channel-types are depicted in this paper. 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.

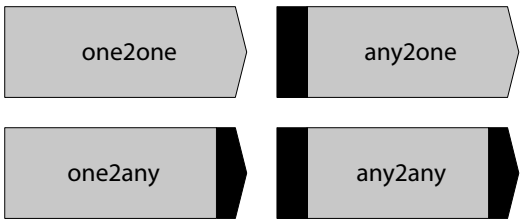

Figure 1. Channel-type variants

3. 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.

3.1. 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. 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 no 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 [8].

3.2. 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.[6] pony uses the C Interface (CIF) library [23] for its protocol-converters, and KRoC's course, socket, file and process libraries [24] for calling various routines that are needed for its functionality.

3.3. 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 outputting status and error messages. In order to prevent this paper from getting too large, error- and message-handling will not be discussed here, since they are not part of pony's basic functionality. Details about pony's error- and message-handling are given in [8].

The startup process will return a given set of *handles*. A handle is the client-end of a channel-type[7] 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.

[6] 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.

[7] Please note that in this paper, 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'.

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 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.

4. 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.

4.1. 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[8] 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 6 and 7. The other parts of the name of the startup process are optional and used for error- and message-handling, for which the startup process returns an *error-handle* and/or a *message-handle* if required.

4.2. 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.

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

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.
- ‘ans.name’ is the name of the ANS. The ANS-name determines which ANS is contacted by the node. Details about this are given in Section 8.2.
- ‘app.name’ is the name of the pony application to which the node belongs. Under this name, the application is administrated by the ANS.
- ‘node.name’ is the name of the node. The node-name determines which configuration file is used by pony to resolve the network location of the node. Details are given in Section 8.1.
- ‘node.type’ is the type of the node, i.e. whether it is the master or a slave.
- ‘result’ is the result returned by the startup process upon completion. If the startup process completes successfully, the ‘result’ parameter will return an OK, otherwise it will return an error. Possible errors that can occur during startup are discussed in detail in [8].
- 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 by the pony environment. 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 other parameters of the startup process are used for error- and message-handling. They are only part of the parameter list of those startup processes whose names contain the relevant options, see above.

4.3. 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.[9] 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 would require extra careful programming. It would need to start a pony environment separately for each application, and use separate handles for the different applications. In such a ‘bridging node’ it would be vital not to mix up NCTs of separate applications. That is, no NCT-ends of one application may be 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 7.

[9] 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.

5. Starting the ANS

As discussed in Section 2.3, 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 8.3 for details.

6. 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*.

6.1. 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. Please note that 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.[10] In pony, this characteristic has been 'moved up' to the application level and now applies to the slaves and to the master. That is, the master node must be up and running before slave nodes can connect to it (although pony provides a mechanism to start a slave node before the master; it just waits in this case, see [8] for details).

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:

[10] 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.

```
(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.
- ‘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. This parameter declares whether the other end is meant to be unshared, shared, or whether we do not know or do not care about the other end’s share-type. Any mismatches with previously allocated ends of the NCT will cause the allocation process to return an error and fail.
- ‘chan.type.end’ is the variable that is to be allocated. ‘<alloc-type>’ is a wild-card 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.[11]
- ‘result’ is the result returned by the allocation process. If the allocation is successful, the ‘result’ parameter will return an OK, otherwise it will return an error. Possible errors are mismatches in the x2x-type of the NCT as declared during previous allocations of NCT-ends of the same NCT-name. A detailed discussion of possible errors is given in [8].

6.2. 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 to send 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

[11] ‘MOBILE.CHAN’ parameters have recently been added to occam-π; any channel-type-end that fits the specified client/server direction and share-type may be passed as an argument.

the pony infrastructure.[12] 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 of them as well.

7. 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)
```

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, possibly except for tasks related to error- and message-handling — which do not involve *networked* channel-type-ends.

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.

8. 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 com-

[12] 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 [8].

puter 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.

8.1. The Node-file

During startup, a node-name must be supplied to the startup process (cf. Section 4.2). 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 check the node-file for the IP address and 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.

8.2. The ANS-file

Similarly to the node-name, the name of the ANS must be given to the pony startup process. 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`' or '`.pony.tcpip.ans.<ans-name>`', depending on whether the ANS-name is an empty string or not — this naming scheme is the same as for the node-file.

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 check the ANS-file for the location (hostname or IP address, and port number) of the ANS.

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.

8.3. 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.[13] 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 the file is found, the ANS looks for the port number under which it is supposed to listen for connections. If the file or the setting are not found, the default ANS port of 7400 is used.

9. A Sample Application

This section 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[14] 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 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 2 through 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.

Figure 2 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.[15] 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,

[13] 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.

[14] Lines starting with '. . .' denote parts of the code that have been folded. This notation is used by origami and other folding editors.

[15] 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.

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.

```
-- 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 1. Sample application: Declarations

```
#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 2. Sample application: The broker

10. Implementation Overview

10.1. NCTs and CTBs

There are two important terms related to pOny which are 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 [15].

```
#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.SLAVE,
                      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 3. Sample application: The worker

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 6.2.

In pony-enabled programs, the memory layout of CTBs is slightly larger than in traditional occam-π programs. This is necessary in order to accommodate the needs of networked CTBs (as well as of non-networked CTBs that may become networked). As discussed in Section 3.1, the pony-specific compiler support, which includes the modified CTB layout, is enabled in KRoC if it is built with the '`--with-pony`' option. The pony-specific CTB layout, as well as the compiler support for pony, are explained in detail in [8].

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

PROC customer (CHAN BYTE key?, scr!, err!)
  INT own.node.id, result:
  PONY.NETHANDLE! net.handle:
  SHARED BROKERHANDLE! broker.handle:
  SEQ
    -- Start pony
    pony.startup.unh (PONYC.NETTYPE.TCPIP, "", "sample-app",
                      "", PONYC.NODETYPE.SLAVE,
                      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)
    IF
      result <> PONYC.RESULT.ALLOC.OK
        ...  Deal with allocation error
      TRUE
        BOOL worker.available:
        WORKERHANDLE! worker.handle:
        SEQ
          -- Get worker-handle from broker
          CLAIM broker.handle
            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
    -- Shut down pony kernel
    pony.shutdown (net.handle)
:
```

Algorithm 4. Sample application: The customer

10.2. Structure of pony

Apart from the compiler support for pony-enabled CTBs, 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 [23] processes. Figure 3 shows the layout of the the pony environment with its various

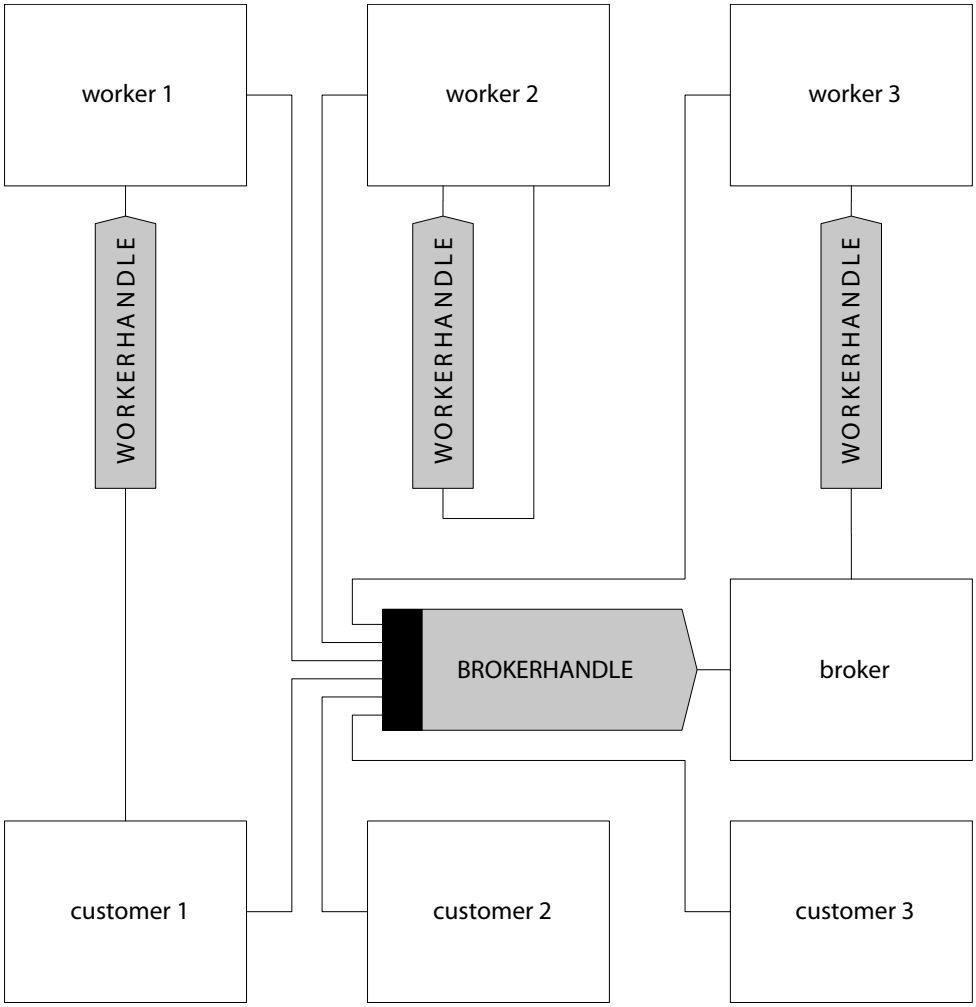

Figure 2. Sample application: Possible dynamic layout

components and the external and internal handles[16] 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. Please note that 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.[17] 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.

The communication between the individual components of the pony environment follows the principle of cycle-free client/server communication as set out in [25]. 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.

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

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

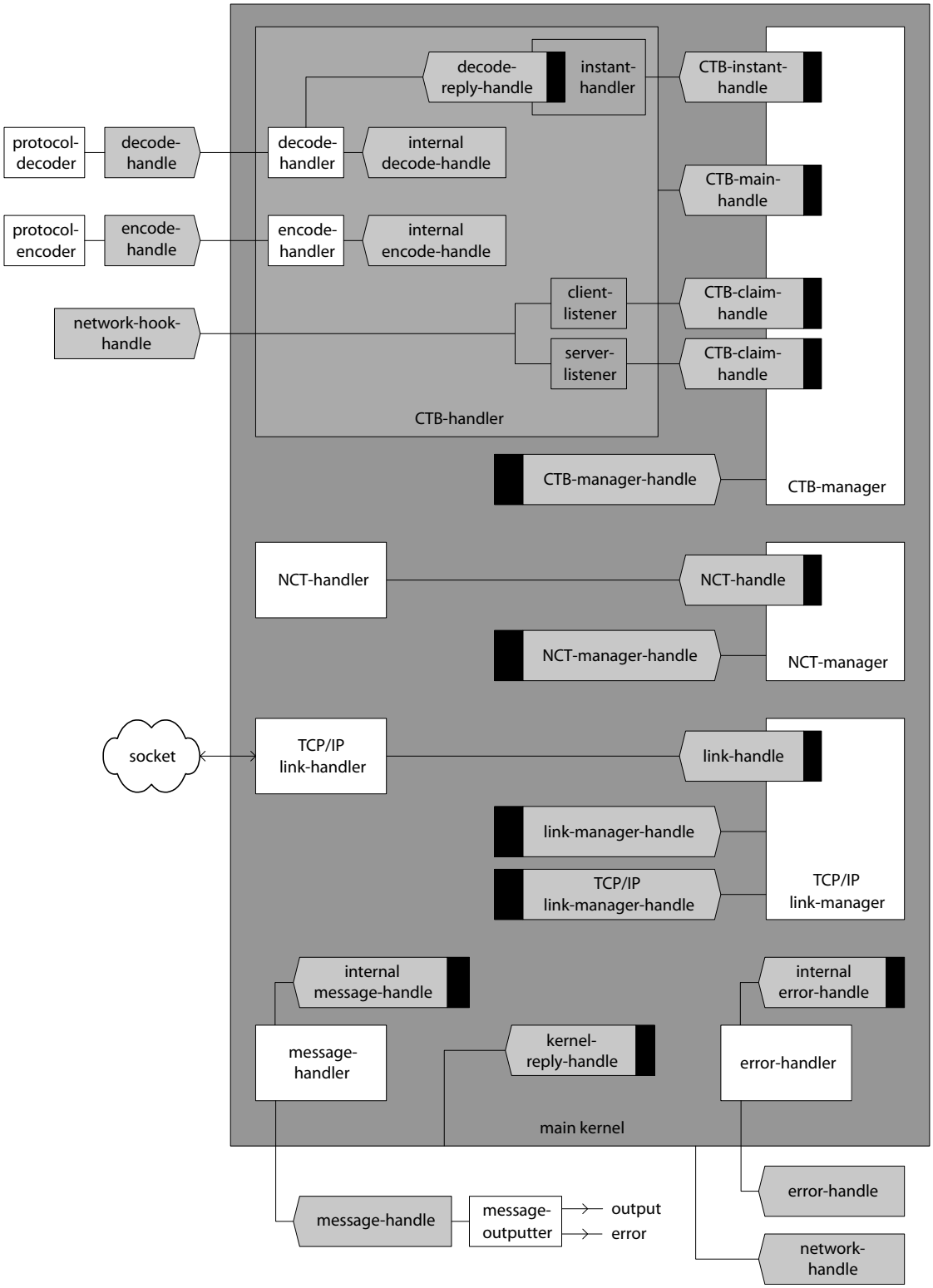

Figure 3. Layout of the pony environment

10.3. Internal 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, is given in [8].

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.

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.

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 3) are no actual components of pony but just simple sub-processes of the CTB-handler.

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 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.

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[18].

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

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.

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 the *link-handles* for existing link-handlers and passes them (via the *link-manager-handle*) to requesting pony components.

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 [10], 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 [21].

Error-handler and Message-handler

The error-handler and the message-handler are used for error- and message-handling. They are only active if this has been requested from the startup process when the node was started.

10.4. 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.

11. Benchmarks

These benchmarks were conducted on the TUNA [26] cluster at the University of Kent, which 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 machines were otherwise idle; memory usage was watched carefully to avoid going into swap. The benchmark programs — which are included in the KRoC distribution — were compiled using KRoC's highest optimisation options, as was the pony library. Each pony node was run on a dedicated host; the ANS was also given a dedicated host (for ease of management; the ANS is not performance-critical).

All the benchmarks 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 such measurement was repeated three times and the mean of the results taken. We have omitted error bars for clarity; the error was within 1% on all benchmarks.

We emphasise that, to date, very little 'tuning' work has been done on pony; these results should only improve with time. That said, the present results are extremely encouraging, and

we have already built several distributed applications using pony which perform well on PC clusters.

11.1. 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 4.

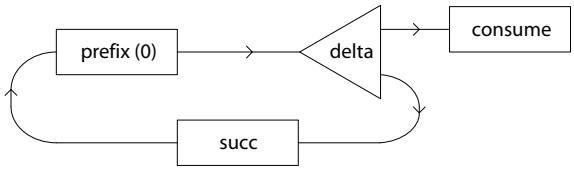

Figure 4. The 'commstime' benchmark

The 'commstime' benchmark consists of four parallel processes, three of which are running in a loop. The processes are connected by channels carrying INTs. The '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. For conventional occam-π programs, the communication time is the context-switch time of the KRoC scheduler.[19]

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. Thus, 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 benchmarks, 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.

11.2. Throughput

The 'bmthroughput' program is intended to measure the aggregate data rate available across a group of networked channels. 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.

[19] 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.

Using 100 KB[20] — 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 5 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. 50 workers).

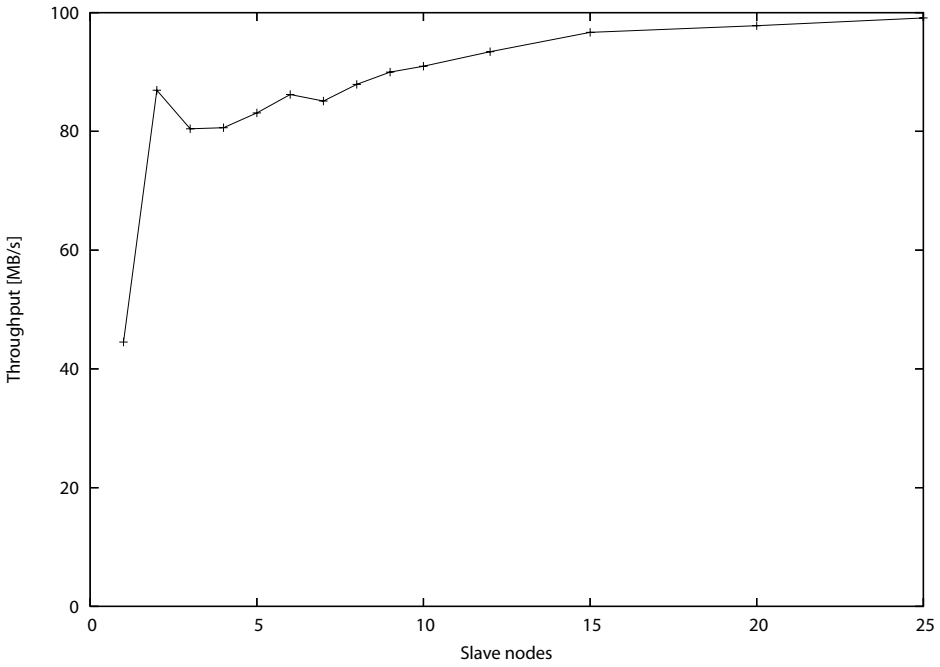

Figure 5. Throughput: 100 KB messages, two workers per slave

Figure 6 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 7 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.

11.3. Network Overhead

In the previous benchmark, the master process's throughput measurement 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 5 is 99.1 MB/s; the network usage measured at the same point 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 to the network protocols in use, and 1.8% due to pony itself.

[20] All byte prefixes used in this paper are decimal, e.g. 1 KB = 1000 B.

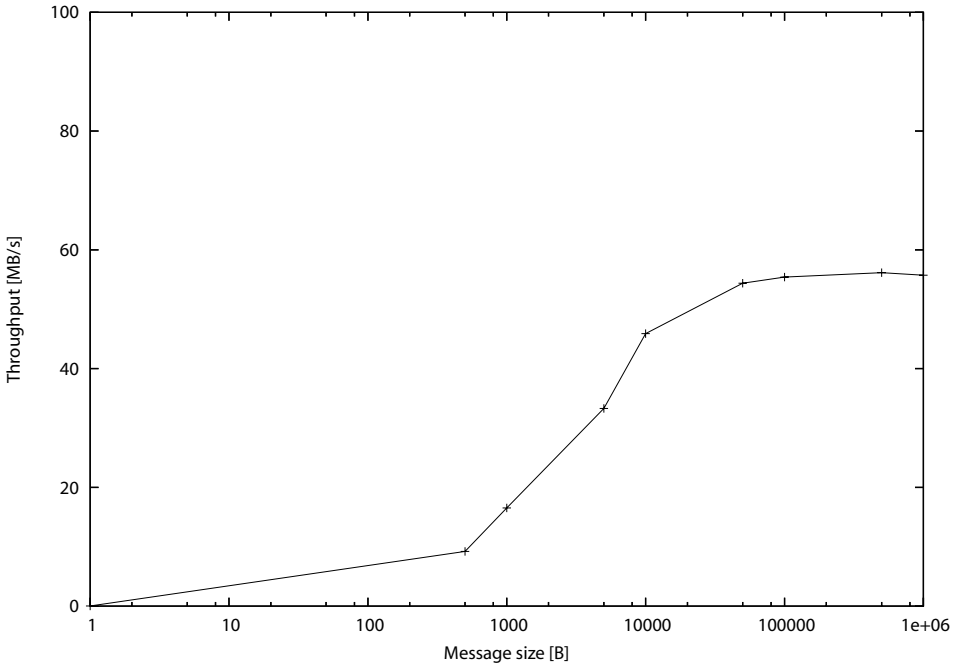

Figure 6. Throughput: Varying message size, one slave with 50 workers

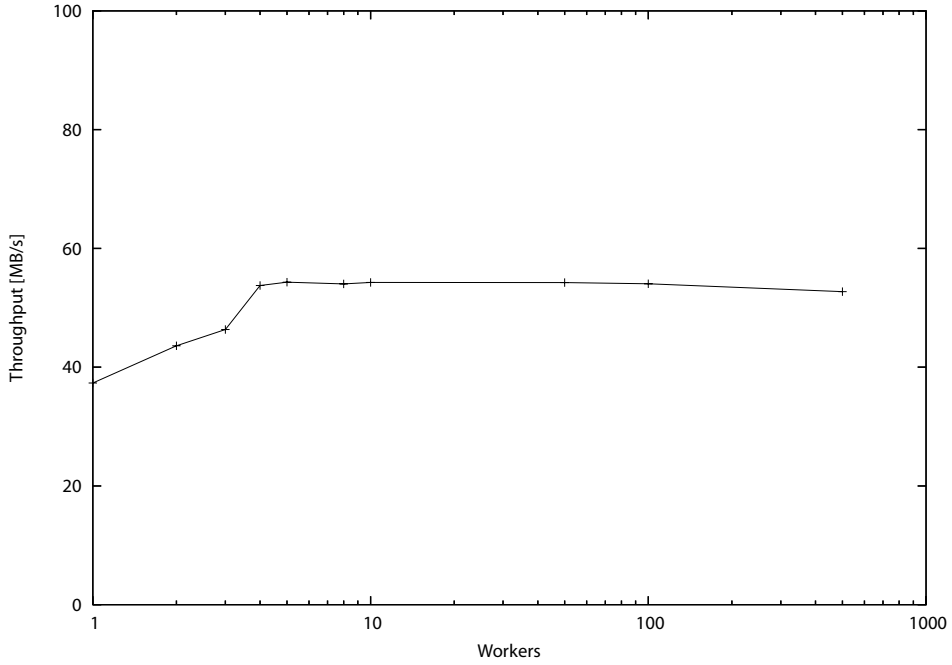

Figure 7. Throughput: 50 KB messages, one slave, varying number of workers

11.4. CPU Overhead

The computational overhead introduced by the pony environment can be evaluated by measuring the CPU time per *user-level communication* (ULC). A 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. The CPU time per ULC is the time between starting to send a ULC via a networked channel and receiving the acknowledgement that the entire ULC has been received by the remote user-level process, 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 ULC, in form of the special protocol created by the protocol-decoder, to travel through the decoder into the pony kernel, and then all the way through the pony kernel until the point where it would have to be outputted to the network. At this point, nothing is sent to the network, but a dummy acknowledgement is sent to the CTB-handler as if an acknowledgement from the remote pony kernel had just been received from the network. Please note that, depending on the user-level protocol, the measurement for a ULC may include one or two such ping-pong times. Details can be found in [8]. 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 8 shows the CPU time per ULC for single byte arrays of varying size. As expected, the CPU time is fairly constant. This is so because the pony infrastructure does not copy the user-level data, but only passes around its address and size.

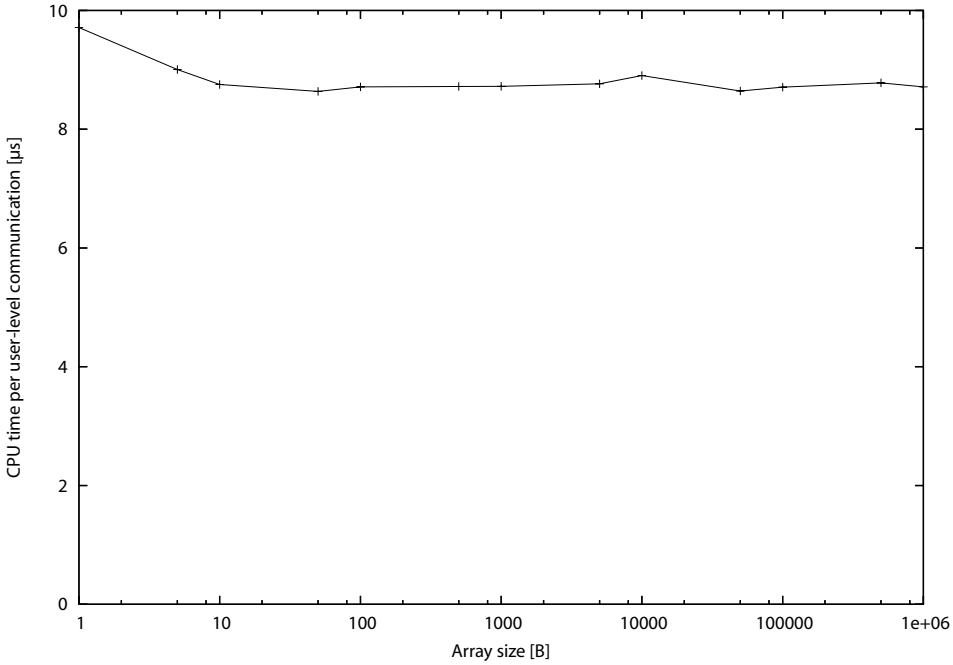

Figure 8. CPU overhead: Single byte array of varying size

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 9 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, because sequential protocols with two or more items require two ping-pongs, whereas non-sequential data (i.e. one item) only

requires a single ping-pong. 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; the copying takes more time the more items there are in the protocol. The copying is necessary because of the way the KRoC compiler evaluates expressions in non-mobile variables (since each item of the sequential protocol may have been an evaluated expression), so that all items from the second item onwards can be sent over the network at once; see [8] for details.

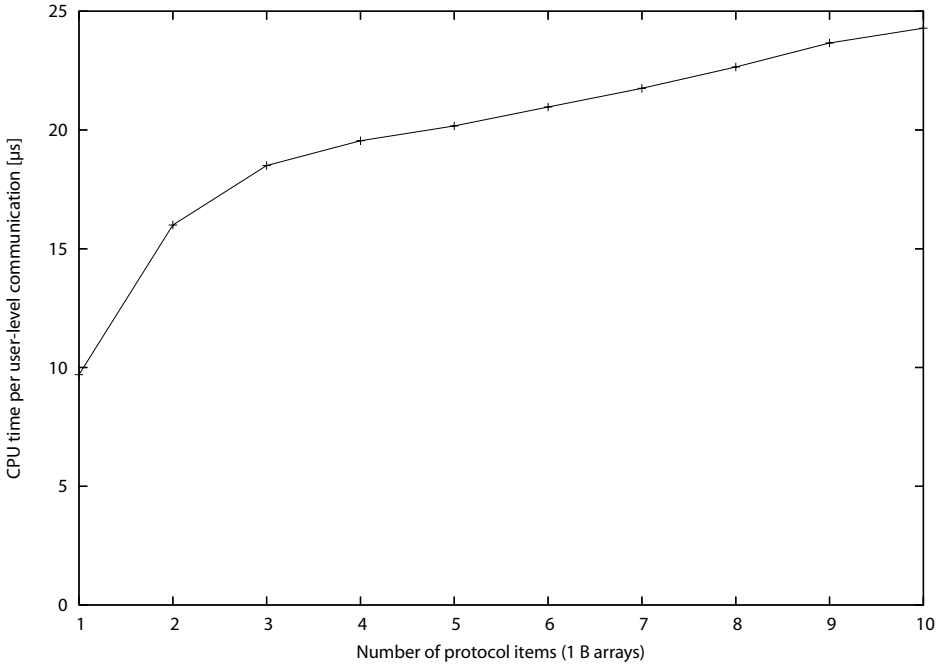

Figure 9. CPU overhead: Sequential protocol, 1 B arrays

As mentioned above, there are always two ping-pongs for sequential protocols with two or more items. Hence, only the copying of protocol items causes the gradual increase, with (n - 2) copy operations for protocols with n items.

Figure 10 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.

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.

11.5. 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

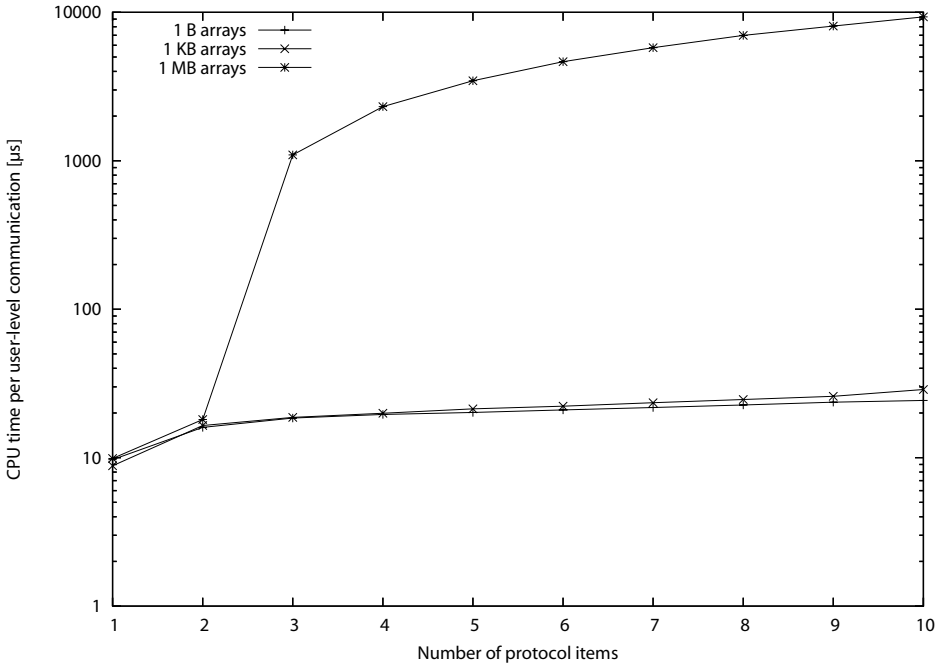

Figure 10. CPU overhead: Sequential protocol, several array sizes

and send the results back to the master; the master then collects and displays the results. For the purposes of this benchmark, the display has been disabled; the master just measures the rate at which pixels are being computed.

The '`mandelbauer`' application can be run in two modes. In shared mode (see Figure 11), 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 12), 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 sends the server-end of its request/response NCT to the master, which will then set up a new handler process. 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.

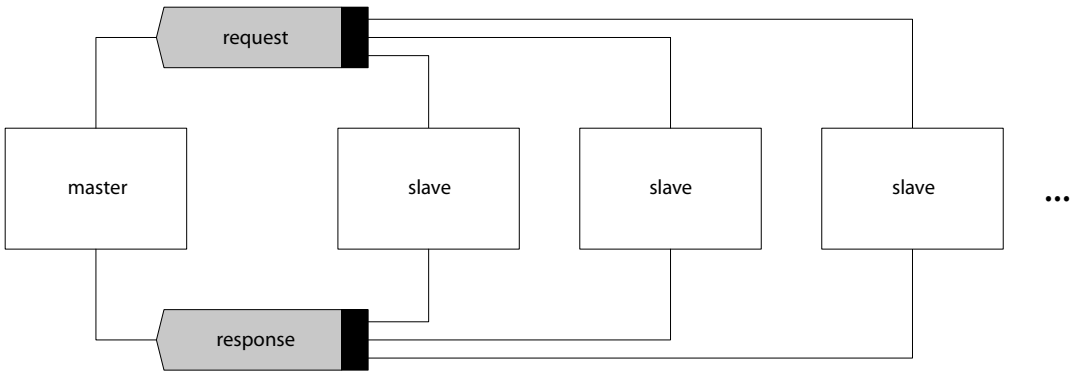

Figure 11. The '`mandelbauer`' application: Shared mode

Figure 13 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

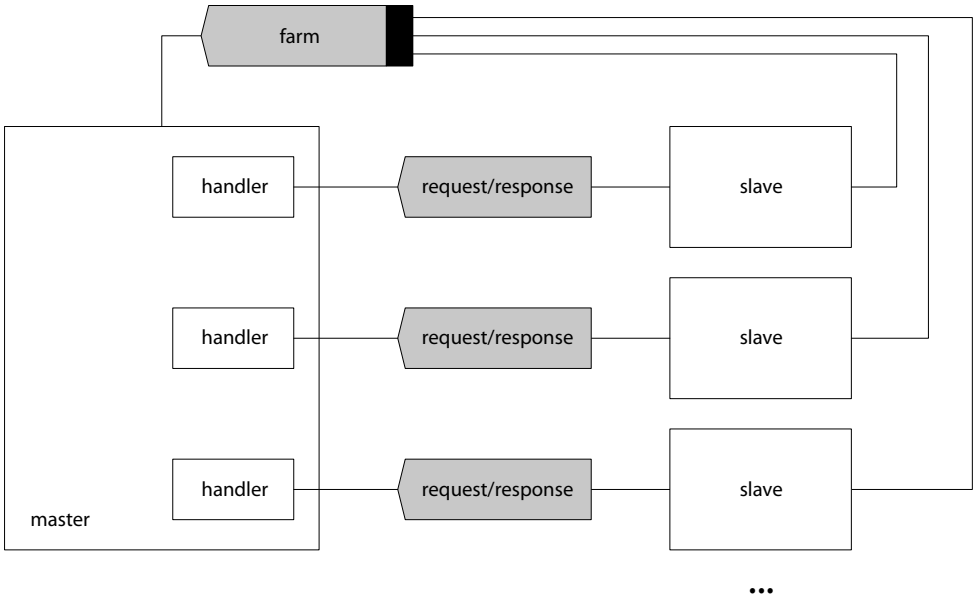

Figure 12. The '`mandelbauer`' application: Multiplexing mode

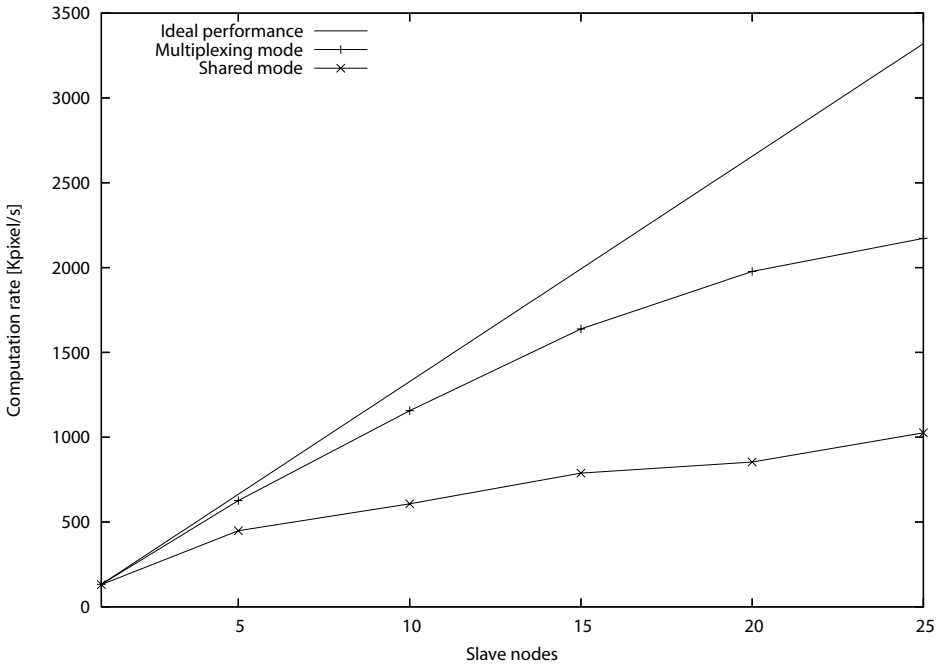

Figure 13. Scalability of a distributed application

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.

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.

12. Conclusions and Future Work

The pony project has succeeded in developing a unified model for inter- and intra-processor concurrency. pony has become a robust and scalable platform for the development of distributed applications. The pony environment expands occam-π's concurrency model into the networked world and achieves semantic and pragmatic transparency according to our objectives that were expressed in Section 1.2.

The handling of pony for the occam-π programmer is simple and straightforward. There is a minimum number of public processes for the basic operations (startup/shutdown, allocation, error-/message-handling), providing the interface between pony and the user-level code. All runtime operations are handled automatically and transparently by the pony kernel. The configuration is easy to understand and minimises the complexity of setting up a distributed application.

By benchmarking, we 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. We have also identified areas where future work on the pony implementation can 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.

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 [27] and mobile barriers [28]. Integrating the pony environment into RMoX, the occam operating system [29], 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.

Other new features added to occam-π in the future would gradually have to be incorporated into pony as well, so that semantic transparency between pony and occam-π would be preserved. Other areas of future work could be the adaption of pony for different architectures (dealing with endianism etc.), a security model for pony (introducing encrypted network communication), as well as alternative startup and configuration models for pony applications distributed on clusters.

For a detailed discussion on possible future work on pony, the reader is referred to [8]. Potential for further development never ceases, just like in the 'real world' — which occam-π and pony are designed to model.

Acknowledgements

The authors are very grateful to Fred Barnes for his work on the KRoC compiler and his helpfulness in discussing the various issues arising from the development of pony and its integration into the KRoC environment.

Thanks also go to Peter Welch and the other members of the University of Kent's Concurrency Research Group, as well as the anonymous reviewers. Their advice in reviewing this paper and their contributions to our many discussions on the project were very valuable.

Finally, the authors would like to acknowledge EPSRC's support for parts of this work through a research studentship (EP/P50029X/1) and the TUNA project (EP/C516966/1), as well as the Computing Laboratory at the University of Kent for supporting parts of this work through a Brian Spratt Bursary.

References

[1] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.

[2] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.

[3] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: `http://wotug.org/occam/`.

[4] Inmos Limited. *Transputer Reference Manual*. Prentice Hall, March 1988. ISBN: 0-13-929001-X.

[5] 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`.

[6] 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`.

[7] 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`.

[8] Mario Schweigler. *A Unified Model for Inter- and Intra-processor Concurrency*. PhD thesis, University of Kent, UK, Canterbury, Kent, CT2 7NF, August 2006.

[9] 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.

[10] Inmos Limited. *The T9000 Transputer Instruction Set Manual*. SGS-Thompson Microelectronics, 1993. Document number: 72 TRN 240 01.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent, June 2003.

[16] I.N. Goodacre. occam NetChans, 2001. Project report.

[17] 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.

[18] 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.

[19] 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*, volume 62 of *WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575*, pages 107–126, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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/.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.