A Simple Protocol to Communicate Channels over Channels

Henk L. Muller and David May

Department of Computer Science, University of Bristol, UK
henkm@cs.bris.ac.uk, dave@cs.bris.ac.uk

Abstract. In this paper we present the communication protocol that we use to implement first class channels. Ordinary channels allow data communication (like CSP/Occam); first class channels allow communicating channel ends over a channel. This enables processes to exchange communication capabilities, making the communication graph highly flexible. In this paper we present a simple protocol to communicate channels over channels, and we show that we can implement this protocol cheaply and safely. The implementation is going to be embedded in, amongst others, ultra mobile computer systems. We envisage that the protocol is so simple that it can be implemented at hardware level.

1 Introduction

The traditional approach to communication in concurrent and parallel programming languages is either very flexible, or very restricted. A language like Occam [1], based on CSP [2], has a static communication graph. Processes are connected by means of channels. Data is transported over channels synchronously, meaning that input and output operations wait for each other before data can be communicated. In contrast, communication in concurrent object oriented languages is carried out via object identifiers. Once a process obtains an identifier of another object, it can communicate with that object, and any object can communicate with that object. Similarly, languages based on the π-calculus [3], use many-to-one communications.

We have recently developed a form of communication that lies between the purely static and very dynamic communication constructs. We enforce communication over point-to-point channels, just like Occam, but instead of having a fixed communication graph we allow channels to be passed between processes. This gives us a high degree of flexibility. A channel can be seen as a communication capability to or from a process. This is close to the NIL approach [4].

These flexible channels can be extremely useful in many application areas. Allowing channel ends to be passed like ordinary data gives us an elegant paradigm to encode the (mobile) software required for wearable computers. Multi-user games often require communication capabilities to be exchanged between nodes in the system. A similar type of channels has been successfully used for audio processing [5], and mobile channels appear to be a natural communication medium for continuous media.

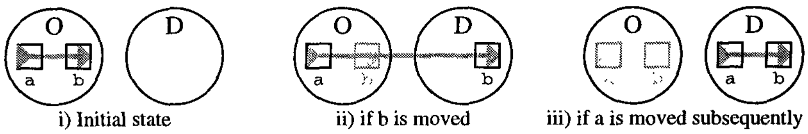


Fig. 1. A system, with two nodes 'D' and 'O', and a channel connecting ports a and b. (i) the channel is local on 'O'; ii) port b is passed to node 'D', stretching the channel; iii) port a is transferred to 'D', the channel snaps back.

In all of these applications, one wants to be able to pass around a communication capability from one part of a system to another. Having one-to-one communication channels as opposed to one-to-many communication models (which many concurrent OO languages offer as their default mechanism) makes it easier for the programmer to reason about the program behaviour, while it simplifies and speeds up the implementation. This is especially important when we are going to implement these protocols at hardware level.

In this paper we discuss the implementation of the mobile channel model. In Section 2 we first give a brief overview of the high level paradigm that we use to model these flexible channels. After that, we discuss the protocols that we have developed to implement the channels in Section 3. In Section 4 we present some performance figures of our implementation.

2 Moving Ports Paradigm

The programming paradigm that we have developed, Icarus, is designed to enable the development of mobile distributed software. In this section we give a brief description of the Icarus programming model, concentrating on the communication aspects; for an in-depth discussion of Icarus we refer to a companion paper [6]. Icarus processes communicate by means of channels. Like CSP and Occam channels, Icarus channels are point-to-point, and uni-directional. That means, at any moment in time, a channel has exactly one process that may input from the channel, and one process that may output to the channel. We call these channel ends *ports*, so each channel has an associated *input port* and *output port*. The two ports that are connected by a channel are called *companion* ports.

In contrast with CSP and Occam, Icarus ports are first-class. This means that one can declare a variable of the type "input port of integers", and assign it a channel end (the type system requires this to be the input-end of an integer channel). Alternatively one can declare a "channel of input port of integers", and communicate a port over this channel. Because an Icarus port can be passed around between processes, a flexible communication structure can be created.

There are two ways to look at mobile channels and ports. The easiest way to visualise mobile channels is to view a channel as a rubber pipe connecting two ports. Wherever the ports are moved, the channel connects them and transports data from the input port to the output port when required. The grey line in

```
{ chan !int d ; chan ?int c ;
  par { { chan z ;
          c ! z ; d ! z ;
        }
     || { ?int x ; int b ;
          c ? x ;
          x ? b ;
        }
     || { !int y ;
          d ? y ;
          y ! 1 ;
        }
  }
}
```

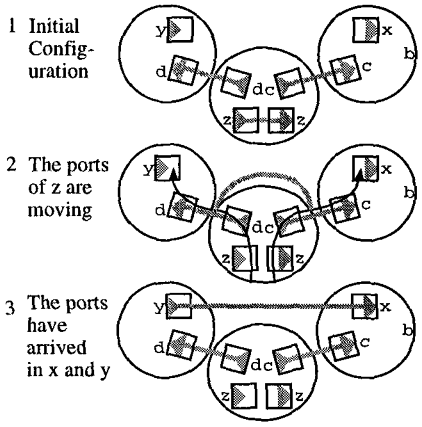

1 Initial Configuration
2 The ports of z are moving
3 The ports have arrived in x and y
y
d
dc
z
z
x
b
c

Fig. 2. Example Icarus program. Left: the code. Right: the execution, the processes before, during and after communication over c and d.

Figure 1(i) shows the channel between the two ports labelled a and b. Another way to view channels and ports is to enumerate them. If we assign a unique even numbers to each channel, say C, then we can define the ports of that channel to have numbers C and $C+1$. Data output over port C will always arrive at its companion port $C+1$; regardless the physical locations of the ports.

In order to preserve the point-to-point property of channels, ports are not *copied* when assigned or sent over a channel. Instead, they are *moved*. When reading a port variable, the variable is not just read, but read-and-destroyed in one atomic operation. These moving semantics guarantee that at any time each channel connects exactly two ports. A port-variable can thus either contain a port, that is, it is connected via a channel to a companion port, or it is unconnected, which we can represent by having no value in the port.

The syntax of Icarus allow us to declare an output port of something by using an exclamation mark: $!T$ is the type of a port over which values of type T can be output. The type $?T$ denotes the ports over which values of T can be input. So the type `!?int` is the type of port over which one can output a port over which one can input an integer.

Icarus has output (`!`) and input (`?`) operators, and a *select* statement to allow a choice between multiple possible inputs. Icarus communication is synchronous, which, as is shown later in this paper, simplifies the implementation dramatically. The example program in Figure 2 creates three processes. Process 1 sends an input port to process 2 and an output port to process 3. This establishes a channel between processes 2 and 3, over which they can exchange data.

3 The Protocol for Moving Ports

The protocol used to communicate ports over channels is simple. We present the protocol bottom up: we start with the basic communication of simple values, then

Port index	Companion port index	Remote Address	Port-state	More
12:	13	-	IDLE	...
13:	12	-	IDLE	...

Fig. 3. Part of the port data structure, with two companion ports.

we discuss sending ports itself, and we finally consider the difficult cases such as sending two companion ports simultaneously. We assume that the protocol is implemented on top of a reliable in-order packet delivery mechanism.

3.1 Communicating Single Values

The port data structure consists of an integer that denotes the index of the companion port, an optional field to denote the remote address of the companion port (not used for a port with a local companion), a field that gives the *state* of the port, and some additional fields for storing process identifications. Two companion ports are shown in Figure 3.

Communicating single values is implemented conventionally, using the same algorithms as used in, for example, the Transputer family [7]. The channel can be in one of three states, IDLE, INPUTTING and OUTPUTTING. This uniquely defines the state of both inputting and outputting processes.

The IDLE state denotes that no process wishes to communicate over the channel. The INPUTTING state denotes that a process has stopped for input and is waiting for communication, but that no process is ready for outputting yet. Conversely, the OUTPUTTING state denotes that a process is ready for output but that no process is ready for input. There is no state for two processes being ready for input and output, as that state is resolved immediately when the second process performs its input/output operation.

The state of a channel is represented as the state of its two ports. It is sufficient to store the state INPUTTING on the output port, and to store the state OUTPUTTING on the input port. This ensures that a check of the local port suffices to determine the state of the channel.

If we want to support a *select*-operation, we will need an extra state, SELECTING, which indicates that a process is willing to input. Upon communication the selecting process will perform some extra work to clean other ports which have been placed in the state SELECTING.

3.2 Communicating Simple Values Off-Node

Off-node communication is synchronous, but has been optimised to hide latency as much as possible. A process that can output data will immediately send the data on the underlying network, and then wait for a signal that the data has been consumed before continuing. Note that the process stops after outputting, until the data has been input; this is essential when implementing the full port-moving protocol.

Because the port-state of a remote companion port is not directly accessible, a port-state is stored at both nodes. Ports with a remote companion can be in the states REMOTE_IDLE, REMOTE_INPUTTING, REMOTE_OUTPUTTING and REMOTE_READY. The inputting port can be REMOTE_IDLE (no process willing to input at this moment), REMOTE_INPUTTING (a process is willing to input and waiting) or REMOTE_READY (no process is willing to input, but data has been sent and is waiting to be input). The outputting port can be in the state REMOTE_IDLE (no process is willing to output at this moment) or REMOTE_OUTPUTTING (a process has output data and is waiting for the data to be input).

When implementing a *select*-operation, a port may get into a state REMOTE_SELECTING to denote that a process is waiting for input from more than one source (note that our *select*-operation only allows selective input, we do not support a completely general *select*-operation [8])

3.3 Communicating Ports Off-Node

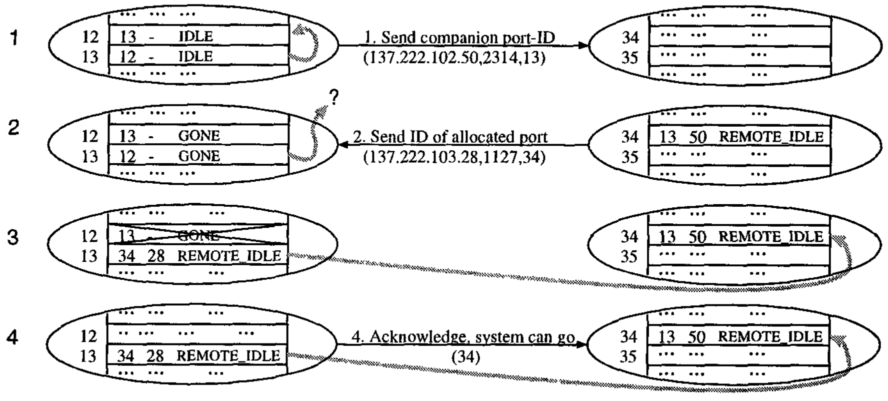
Locally, within a node, a port can be simply an index in an array. Within a node, ports can be freely moved around from one process to another simply by communicating this index value. We discuss transfers of ports to another node in four steps:

1. Transfer of a port from a source node to a destination node, where the companion port is on the source node (channel stretches).
2. Transfer of a port from a source node to a destination node, where the companion port is on a third node.
3. Transfer of a port from a source node to a destination node, where the companion port is on the destination node (channel snaps back)
4. Difficult cases (simultaneous transfers of two companion ports)

1. Transfer of a port from a source node to a destination node, where the companion port is on the source node. In this case the companion port is local, and stays behind. According to Section 3.1, the port that will be sent, must be in the state IDLE, INPUTTING, or OUTPUTTING. Because ports are associated with exactly one process, we can be sure that the port that is going to be sent is not active. That is, no process can be inputting or outputting on the port which is being sent: if a process was inputting or outputting on this port, then the process cannot send the port (it could only attempt to send the port over itself; the type system prevents this from happening).

Because the port being sent is not active, it is sufficient to send some identification of the companion port to the destination node. The identification that we send consists of a local index number, and the node address of the source (for example an IP-address and IP port-number). This tuple (local index, node address) is a globally unique identification of the port.

Upon receiving the identification, a new port is allocated, and the companion port index and node address are stored in this port. The port-state is initialised to REMOTE_IDLE, for the port was inactive. The second step is to inform the



1
2
3
4
12 13 - IDLE
13 12 - IDLE
1. Send companion port-ID (137.222.102.50,2314,13)
34
35
?
12 13 - GONE
13 12 - GONE
2. Send ID of allocated port (137.222.103.28,1127,34)
34 13 50 REMOTE_IDLE
13 34 28 REMOTE_IDLE
4. Acknowledge, system can go (34)

Fig. 4. Steps to send a port to a remote node. Grey arrows are channels.

companion node of the newly created port index, and the companion port's state is set to REMOTE_IDLE, REMOTE_INPUTTING, or REMOTE_OUTPUTTING. After the companion port is updated, step three is to delete the original port. Even though the port send was inactive, it was possible that at most one message was waiting on it, in that case the data is forwarded in order to preserve the invariant that the data has been sent to the remote node. Finally, in the fourth step an acknowledgement is sent to the destination node, signalling that the companion port is ready.

This four step process is summarised in Figure 4. Between steps 1 and 2 is a transitional period when the companion port is not connected to anything. For this reason, the companion port-state is set to GONE in step 1. Any I/O operation between steps 1 and 2 will wait until this transient state disappears.

2. Transfer of a port from a source node to a destination node, where the companion port is on a *different*, third, node. This case is slightly more difficult, and we need a generalised version of the previous protocol. This generalised protocol has four steps, outlined in Figure 5. In step 1 the global identification of the companion port is sent from 'S' to 'D'. In step 2 the newly allocated port-index is sent from the destination node 'D' to the companion node 'C'. This will allow the companion port to be updated. Step 3 will acknowledge to 'S' that the port has been transferred, allowing the source node to delete the original port. Finally step 4 is an acknowledgement to 'D' signalling that the port is ready for use.

Performing these four steps in this order will ensure that any message that might have been sent to the source node is collected and forwarded to the destination node. In the worst case, a message might be output on the companion port, just after the source node has initiated the transfer. This message will be delivered to the source node while the port is in state GONE. The data will be kept here until step 3 of the protocol, whereupon it is transferred to the destination node.

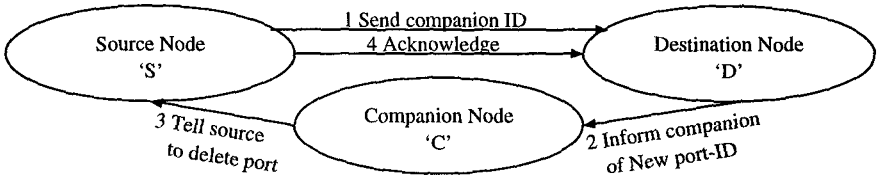


Fig. 5. Steps to send a port in a system involving three nodes.

Note that the case with two nodes discussed in the previous section is a special case of this four step protocol. If the companion port resides on the source node, then step 3 is a local operation.

3. Transfer of a port from a source node to a destination node, where the companion port is on the destination node. Because the port is being sent to the node where the companion resides, we must ensure that both companion ports are in a 'local' state, allowing data transfer to be optimised for local transport.

We use the same protocol as before. Because the destination node and the companion node are now one and the same, step 2, informing the companion node of its new status, is now a local activity. We still need to execute steps 3 and 4, telling the source node that the port can be deleted, and acknowledging the completion. These steps take care of the situation in which the port sent was originally in the state REMOTE_READY. The companion had sent data to the input-port, and the data was waiting to be picked up. Before the port is deleted, we forward the data to the destination node. Both companions are now on the destination node, so we change the input port-state to OUTPUTTING, and keep the data in the outputting process until a process is ready to input it. The data can be safely kept in the outputting process because it had not been allowed to proceed.

4. Difficult cases. There are two difficult cases worth discussing: two companion ports that are sent at the same time (to the same or different destination nodes); and the case where a port is transferring a port, which is in turn transferring a port (which in turn might be transferring a port).

When two ports are transferred simultaneously, the protocols moving the two ports might interfere with each other. The source of the interference is step 2, where the companion node is updated to reflect the new location of the port transferred. We overcome this problem by checking during step 2 whether the companion port is actually being moved, which is indicated by the port-state GONE. If this is the case, then the message is forwarded to the companion's destination node.

If a companion port is found in the state GONE, then the other port must find its companion GONE (because of the message ordering), and both nodes will forward the message of step 2. Eventually, both newly created ports will have a reference to each others' location, and both old ports are deleted. Note that if

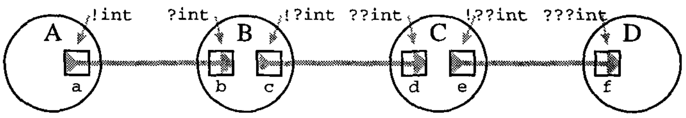


Fig. 6. Ports over ports over ports.

two companion ports are sent simultaneously, then both ports must have been IDLE, and no data will be sent until both have arrived at their destinations.

The most difficult situation is the case where a port is sent, and this port is actually a port of ports, carrying a port. In Figure 6 we have sketched a situation with 4 nodes (A, B, C and D) and 6 ports (a, b, c, d, e and f). The ports are linked up as denoted by the grey lines. Outputting port d over e is a normal output operation as discussed before. Similarly, outputting port b over c to d is a normal operation.

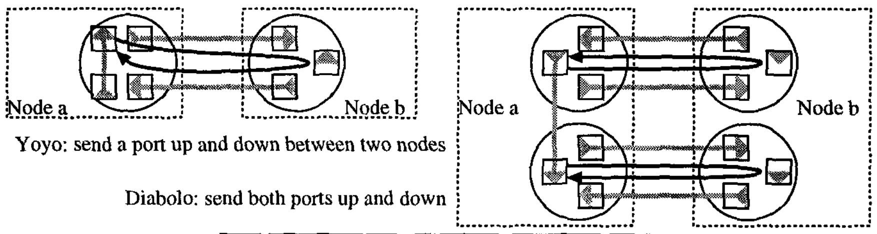
If the transfers of b and d happen simultaneously, then b is actually making a double hop, from node B via node C to node D. Indeed, there may be an arbitrarily long finite list of ports to be moved, only restricted by the number of '?' in the port type definition (the typing system guarantees it to be finite). This forwarding causes a serious complication in the protocol (around 20% of the code), and we are considering whether it is worthwhile to prohibit this.

3.4 Security

The protocol described above is not secure. Most notably, an attacker can forge a message for step 2 of the protocol, and deliver it to a node, in order to obtain a port in the process. The solution is to extend our protocol with an extra step. When executing step 1 of the process, we have to send a clearance message to the companion port, informing it that a move is imminent. The companion node will only execute step 2 of the protocol when the source node has sent this clearance. This provides security against unknown processes taking channel ends, but on the assumption that the underlying network enforces a unique naming scheme, and secure delivery of messages.

3.5 Discussion

The protocol has been optimised so that the frequent operations, such as sending ordinary data or ports, have high performance. There are communication patterns which will perform badly. The worst case is a program where a process outputs a large array over a port, while there is no process willing to input this data. If the input-port is now sent from one process to the next, then the data is shipped with it. If the input port were to be sent many times, we would waste bandwidth and increase latency. We could optimise this case by not sending data until required, but this would seriously impair performance in the more common case, when data is communicated between two remote nodes (incurring double latency on each transferral).


Node a
Node b
Yoyo: send a port up and down between two nodes
Diabolo: send both ports up and down
Node a
Node b

Bouncing between		Yo-yo	Diabolo
Bristol-a	Bristol-a	7.70 μs	16 μs
Bristol-a	Bristol-b	4.7 ms	9.9 ms
Bristol-a	Amsterdam	322 ms	660 ms

Fig. 7. The two test programs that we measured, with round trip times.

4 Results

The protocol has been implemented in C, to implement Icarus on a network of workstations. The implementation of the protocol takes less than 1000 lines of C, underlining its simplicity. We use UDP (User Data Protocol) for the underlying network, with an extra layer to guarantee reliable in-order delivery.

We have tested the protocol with many different applications. A class of students has written multi user games in Icarus. To convince the reader that our implementation works over the Internet, we show timings of the two test programs displayed in Figure 7. The first test program, Yo-yo, keeps one port at a fixed node, and sends the companion port forwards and backwards to a process on a second node. In this program the channel must stretch and snap back every time. The second test program, Diabolo, stretches the protocol with some more demanding communication. Two pairs of processes are running on two nodes, each pair of processes sends one of two companion forwards and backwards between two nodes.

We have measured three configurations: one where all processes are running within the same node (which measures communications inside the run-time support only), a configuration where we run the processes on two machines on the same LAN, and a configuration where we run the processes on two distant machines. The performance is linearly related to the average latency between the nodes involved, and to the number of messages that is needed for a round trip. This is clearly shown in the timings of Yo-yo versus Diabolo for the three configurations.

5 Conclusions

In this paper we presented a simple protocol that allows us to treat communication channels as first class objects. The programming paradigm allows processes to communicate over point-to-point channels. Values communicated can either be simple values (like integers or arrays of integers), or *ports* (ends of channels). Because the paradigm allows a synchronous implementation, we have been able to produce a simple implementation. Whenever a port is transferred, the port is known to be idle (because ports are point to point and synchronous), so we do not have to transfer any state around.

We have developed a four step protocol to move a communication port from one node to another. Step one carries the port identifier, creating a new port on the destination node. Step two informs the companion port of its new companion. Step three informs the source that the original port is to be deleted. Step four allows the newly created port to start communication. These steps may be local or remote, depending on the relative positions of the source, destination, and companion nodes. If the two connected ports end up in the same node, the channel snaps back to within the node, and the implementation switches to a highly efficient local communication protocol.

The applications of this protocol lie in many areas: multi user games (where players receive channels that allow them to communicate with other players), continuous media switchboards (where a channel carrying, for example, video signals is handed out to two terminals), and wearable computing (establishing communication channels between parts of a changing environment). We are currently integrating this protocol in a wearable computer system, and its applications in continuous media.

References

1. Inmos Ltd. *Occam-2 Reference Manual.* Prentice Hall, 1988.
2. C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.
3. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1–40, Sept. 1992.
4. R. Strom and S. Yemini. The NIL Distributed Systems Programming Language: A Status Report. *SIGPLAN notices*, 20(5):36–44, May 1985.
5. R. Kirk and A. Hunt. MIDAS-MILAN An Open Distributed Processing System for Audio Signal Processing. *Journal Audio Engineering Society*, 44(3):119–129, Mar. 1996.
6. D. May and H. L. Muller. Icarus language definition. Technical report, Department of Computer Science, University of Bristol, January 1997.
7. M. D. May, P. W. Thompson, and P. H. Welch, editors. *Networks, Routers & Transputers.* IOS Press, 1993.
8. G. N. Buckley and A. Silberschatz. An Effective Implementation for the Generalised Input-Output Construct of CSP. *ACM Transactions on Programming Languages and Systems*, pp 224, Apr. 1983.