Using Channels for Multimedia Communication

David May Henk L Muller
Department of Computer Science, University of Bristol, UK
http://www.cs.bris.ac.uk/~{dave,henkm}/

Abstract

In this paper we present a paradigm to express streams and its implementation. Streams are a convenient mechanism to communicate multimedia data, for example video or audio, between systems. The paradigm is based on channels, as found in CSP and Occam, but with two important modifications. First, our Flexible Channels *can be connected dynamically, and passed around as first class objects. Second, although synchronous in nature, the compiler and run time support will implement communication over channels in a less synchronised mode, when program semantics allow for this. With this technique, a programmer can use a high level language to communicate tiny messages (audio samples) at a high data rate. We have initially used streaming mode to optimise inter node communications, but we hope to use optimisations based on the same techniques to reduce overheads of communications within a node.*

1 Introduction

A convenient representation for multimedia data is the stream. For example, an audio object can be represented as a stream of frames, where each frame consists of two 16-bit samples. Using the notion of a stream of frames, a complex operation on an audio object can be partitioned neatly. A network of processes can be built, where the processes pass audio objects (streams of frames) to each other. Indeed, this network can be implemented on a distributed or parallel platform.

Almost every system supports streams. Solaris (Sun Unix) has a `streamio` library and system-calls for performing asynchronous I/O on these streams. The Netscape API uses streams as its basic communication mechanism. Java has 19 IO classes which support stream-like objects in various ways. The problem with all of these systems is that they do not have a semantically clean model for a stream. Most provide a low level programming interface in which, for example, buffer management and flushing must be carried out by the programmer.

The mechanism that we propose uses a well founded mechanism from CSP, the channel, and extends it to carry multimedia data types such as audio or video signals. Our extensions have relaxed some of the properties of CSP channels, such as compile time fixed communication graphs. We have retained the synchronous communication model, but we developed an implementation that optimises communication so that data is streamed almost asynchronously between two processes.

In Section 2 we will give some background and the motivation for our work. The communication model is discussed in Section 3. The implementation and performance figures on a distributed system are presented in Sections 4 and 5. We conclude the paper with a discussion of future work.

2 Requirements and Background

There are three requirements when designing a communication mechanism that will handle multimedia data.

First, the implementation must be efficient. Architectures have only just become capable of handling multimedia streams and we cannot afford a significant loss of efficiency.

Second, the programmer should not be bothered with low level implementation details. The programmer should only be required to implement the operations on the multimedia objects. For example, we want them to concentrate their efforts on algorithms that generate sound rather than on strategies for collating data samples into blocks so that they can be transmitted efficiently.

Third, streams should be treated as a first class object. For example, it should be possible for a stream which carries audio samples (representing sound) to be passed from process X to process Y so that process Y subsequently receives the sound. Below we discuss two example systems that have been designed to cope with streams, and their shortcomings with respect to the requirements above.

2.1 Existing systems

Solaris 2.5 has a number of libraries and system calls that allow the user to handle streams. A stream is represented in Solaris with a file descriptor. System calls can be used to push data onto a stream (`write`), or to get data from a stream (`read`). In contrast with earlier systems, asynchronous read and write operations are supported. As an example, the function `aiowrite` writes data to a stream asynchronously. This means that the write operation is performed concurrently with other activities in the program and the user is notified when the write is completed. With this function a buffer of data can be streamed to another process while the next buffer is prepared.

This approach has some complications. First, it is the programmer's responsibility to pack data into buffers of appropriate size. The natural data size of audio is two to four bytes per sample. This is too small for the `aiowrite` call, so the user will have to pack the data into bigger chunks. Second, and a more fundamental problem, is that the buffer must not be used when `aiowrite` is using it. There is no run-time or compile-time check on mutually exclusive access, and if the buffer is used during the transfer, then the outcome is unpredictable. Some of these problems can be solved by implementing a library on top of `aiowrite` (for example `stdio`), but we will show a more comprehensive solution later.

Java has the opportunity to provide a better communication library. Java has a type system and a more or less built-in concurrency model. Java has around 20 I/O classes that handle various types of I/O. One of these classes, `java.io.BufferedOutputStream`, comes close to what we require. The class provides buffered output. Bytes are copied to an internal buffer one at a time, and subsequently sent in large chunks to the receiver, where the data can be read piecemeal again.

A fundamental problem with this approach is the lack of coupling between sender and receiver. There is no synchronisation at any stage between the sender and receiver. The lack of synchronisation makes it more difficult for the programmer to reason about the program, and more difficult for the implementor to implement (given that we want channels to be first class). UNIX and many other systems have similar problems with the synchronisation.

Neither the UNIX nor the Java solution offers a first class stream: a UNIX file descriptor only has a meaning in the current process, and cannot be passed to another process. Similarly, the Java object identifier for the buffered output stream only has meaning within the application, and cannot be transferred to another application. Other systems such as Chorus and the MACH kernel address efficient streaming of data, but none of those provide a clean high level model.

3 Flexible channels

We have developed a programming paradigm, Icarus, for mobile and multimedia software. Icarus is a concurrent language with facilities for process migration and flexible communication structures. Icarus allows processes to communicate multimedia data streams and ordinary data using *flexible channels*. In this section we give a brief description of these channels. For an in-depth discussion of other aspects of Icarus we refer to the Icarus Definition [3]. The efficient implementation of this mechanism is discussed in Section 4.

3.1 The channel

Icarus processes communicate by means of channels. Like CSP and Occam channels, Icarus channels are point-to-point and uni-directional. That means that 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. The channel ends are called *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.

As is the case in Occam, the user can perform output operations on a channel using a pling `!`, and an input using a question-mark `?`. The rest of the syntax of Icarus is close to C; the semantics are close to Occam. As an example, a process that merges two audio streams is:

```
buffer( ?int in1, ?int in2, !int out ) {
  int o, x, y ;
  while( true ) {
    in1 ? x ;
    in2 ? y ;
    o = ( x + y ) / 2 ;
    out ! o ;
  }
}
```

The parameters `in1` and `in2` are input ports of integers, `out` is an output port of integers. In the loop we input an integer from each of the ports, calculate the average, and output the results on `out`.

3.2 Flexible channels

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. This is similar to the pi-calculus [5], except that our ports are point-to-point.

There are two ways to look at mobile channels and ports. The easiest way to visualise mobile channels is to view a

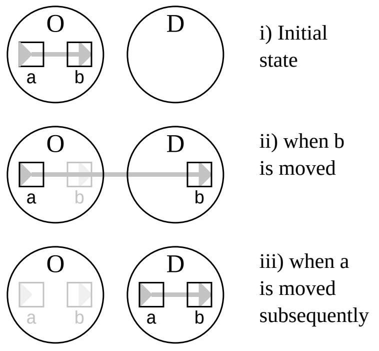


Figure 1. i) Port a **is connected to** b**, both reside on node 'O'. ii) port** b **is passed to node 'D', stretching the channel; iii) port** a **is passed to 'D', the channel snaps back.**

channel as a rubber pipe connecting two ports. Wherever the ports are moved, the channel connects them and transports data from the output port to the input port when required. The grey line in 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 number 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 of 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.

3.3 Use of the flexible channel

A trivial example of the use of a flexible channel is in a phone switchboard. When a connection is to be established, a channel is created in the switchboard to transfer the data (the audio signal). The two ports are sent to the two terminals that are to be connected. The switchboard is now no longer involved, since the channel is now, at least conceptually, directly connecting the two terminals.

A slightly more demanding example is a video conferencing system. In such a system, we want to establish a communication channel to transfer live video and audio between two terminals. The video signal needs to be compressed at the sending end and decompressed at the receiving end. With our system, we can set-up a compressor and decompressor connected by a channel on one terminal, and then ship the decompressor to the other terminal. The connection between the compressor and decompressor will span the low bandwidth network between the two.

A final example is a large parallel audio composition system. In order to produce a realistic synthesis of the sound of an orchestra one needs a massively parallel machine. Channels are a natural mechanism to transfer audio between nodes of the parallel machine, and to transfer data between various sound generating processes in a similar style as that used by Kirk et al [2].

The power of the flexible channel is that it is a generic mechanism that can be used within in a node, between nodes in a tightly coupled systems, and between nodes of a distributed system

4 Implementing flexible channels efficiently

So far we have discussed the principles of how a flexible channel should operate. In order to be useful in a multimedia context, we must implement the channels efficiently. This section describes the implementation, the following sections present performance figures and discusses future enhancements.

4.1 Synchronous mode

The naive implementation of a channel is similar to the operation of transputer channels [4]. A channel can be in one of three states: idle, inputting (where the input side is ready but no one is willing to output over the channel), or outputting (where the output side is ready but no process is willing to input). When both processes are willing to communicate the data is transferred from the output process to the input process.

When implementing simple channels on a distributed system, a process performing output will speculatively send the data over the channel where it will reside on the input-port until a process is ready for input. After the data has been input, an acknowledgement is sent to the outputting process that the process may continue. Note that at most one process is waiting on a channel and therefore at most one data item is waiting to be input or output. This protocol results in a low bandwidth channel, because it is essentially stop-and-wait.

4.2 Flexible channels

The protocol for sending ports over channels is discussed in detail in a previous publication [6]; here we give a brief overview. When a port is output over a channel, the actual information sent is the location (IP address, internal port number) of the companion-port. On input of a port over a channel, the receiving end acknowledges reception to both the outputting process and the companion port. On reception of this port, the output process may continue.

Flexible channels can be implemented using this simple protocol because of the synchronous nature of channels [6]. If non-synchronous communication were allowed, then the communication of a port would have to be preceded by a “pipe-cleaning” step, to ensure that any outstanding data on the channel is collected before transferring the port.

4.3 Switching to streaming mode

The synchronous communication mode above has a very limited bandwidth. The bandwidth is limited by the latency between the two nodes that are communicating, because every data item that is sent needs to be acknowledged before the next data item can be sent.

In order to increase performance, we need to switch to a less synchronised, buffered, protocol. However, if we do this for all communication, the user might observe unexpected program behaviour. As an example, the program shown in Figure 2 has two processes X and Y connected by two channels K and L. Process X outputs data over K and L alternately. Process Y selects input from K and L, but will always input data from K and L alternately because of the structure of X. If we stream data over K and L asynchronously, then Y could first input a number of items from one channel before inputting from the other, which is not in accordance with the programming model.

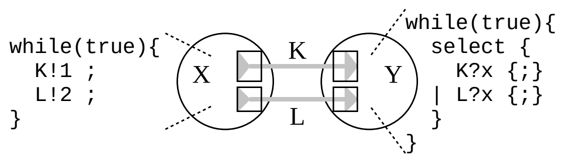
However, There is a class of processes where the user cannot distinguish between synchronous communication and streaming communication. This class offers a natural model to capture many multimedia applications. If a process is found to be of this class, then the communication library can switch to streaming mode, allowing the execution of the user program to overlap with communications. Below we first specify this class of processes and then we discuss two optimisations that can be applied.

4.3.1 Process model

An example that can be implemented with streams is a process which merges two audio streams:

```
buffer( ?int in1, ?int in2, !int out ) {
  int o = 0, x, y ;
  while( true ) {
    par {
       in1 ?  x ;
    || in2 ?  y ;
    || out !  o ;
    }
    o = ( x + y ) / 2 ;
  }
}
```

This process cannot distinguish whether the channels are synchronous or not. In general, data can be streamed if



```
while(true){
  K!1 ;
  L!2 ;
}
```

```
while(true){
  select {
    K?x {;}
  | L?x {;}
  }
}
```

Figure 2. Neither X nor Y is I/O Par

all I/O operations are performed in parallel. This class of processes is closely related to those identified by Welch as I/O-PAR [10]. The compiler could easily detect maximum sets of processes with this property. In order to maintain an intuitive performance model, we have added an I/O-PAR facility to the language Icarus. The rules for a process to qualify for switching its ports to streaming are:

- the process does not use any select statement (select statements allow one out of a set of I/O operations to be executed), and
- the process has no input or output statements that are executed in sequence, so only parallel I/O.

There are two additional requirements, which are due to the mobile nature of Icarus. A process does not qualify for streaming if:

- the process communicates ports.
- the process migrates.

Note that even though a process itself may not qualify for streaming, its constituent sub-processes can qualify. For example, an outermost process may not qualify because it migrates. However, it may subsequently execute an I/O-PAR construct within the migrated process, where streaming can be enabled.

4.3.2 Streaming implementation

Once the compiler identifies a process which can use streaming, the compiler inserts two calls in the code. Before the process the compiler inserts a call to a function in the run time support system that permits streaming. After the process a call to disable streaming is inserted.

Switching from synchronous communication to streaming works as follows. The run time support system sets a flag to record that the ports used in this section of code can stream. It then sends *stream-requests* to all companion ports. Streaming is enabled on a channel when processes on both ends have sent a stream-request.

Implementing the streaming mode is not trivial. A naive unsynchronised implementation will saturate the low level network, which can lead to a considerable loss of performance. Some mechanisms are needed to limit the number of outstanding messages, and to throttle traffic. Many solutions have been developed for these problems [9, 7], for example sliding window protocols or an exponential slowdown. We have implemented just a simple sliding window protocol.

Switching back to synchronous mode can be difficult. If the sender wants to switch back before the receiver, then the sender simply stops streaming after informing the receiver. A more difficult case is when the receiver wants to switch back while the sender is still streaming. In this case a message is first sent to the sender to stop, whereupon the sending process is forced to resend all rejected traffic in synchronous mode on its next I/O request.

Conveniently, the streaming mode is confined to a well defined section of the run time support. Only a small part of the protocol implementation had to be changed when the streaming mode was implemented. No changes had to be made in the sections of the protocol that deal with first class ports and migration of processes and in the corresponding sections of the run time support system.

4.3.3 Buffered implementation

When in streaming mode output can be buffered. We do not have to prevent deadlocks, since I/O-PAR processes cannot deadlock [10]. We therefore buffer data in a packet buffer and only send the packet when the buffer is full. When the sender switches back to synchronous mode the buffer must be flushed to restore synchronous mode.

Other systems, such as the Java model or a high level Unix model, may deadlock if the user forgets to flush the buffer. In our case the run time support flushes the buffers implicitly because the compiler inserts a call to switch back to synchronous mode, *where necessary*.

5 Results

We have done a number of performance tests, to show that our system is working. We have not yet run any complicated programs, but instead we measure the bandwidth of the system *at language level*. Our test program creates two processes connected by a channel. One process migrates to another node and inputs integers from the channel; the other process stays on the current node and outputs integers over the channel. We time the transfer of a million input operations and calculate the mean time for inputting an integer. Note that at user level we send tiny messages of only 4 bytes, equivalent to a stereo 16-bit audio sample.

We have measured two benchmarks with three different communication strategies on two different machine configurations. The three implementations measured are the naive implementation (with synchronous communication), the streaming implementation, and the buffering streaming implementation. The configurations that we used are two processes on the same node and two processes on two nodes connected by a LAN.

Transfer of tiny messages (32 bits)		
Implementation	Single	LAN
Synchronous	18 μs	1350 μs
Streaming	18 μs	149 μs
Buffered Stream	18 μs	20 μs

Table 1. Performance in μs per message.

Table 1 shows the performance figures. The times given in this table are in microseconds per message. For a single node architecture the figures are all similar, around 18 μs per message. Most of this time is spent in context switching between the two processes and physically copying the data.

On the LAN, the detection of streaming processes can lead to significant performance gains. Synchronous transfers take around 1350 μs per transfer, Buffered Streaming reduces this to 20 microseconds per message, an improvement of a factor of 65. This can be expected because the synchronous implementation is entirely latency bound (the latency between the two machines is around 700 μs), while the streaming implementation is bandwidth bound. Given that the LAN bandwidth is 250,000 messages per second, or 4 μs per message, we see that there is room for further optimisations. This is probably due to the fact that we have not yet optimised buffer sizes.

It is hard to compare these timings to C or Java. Their performance is low ($> 1ms$ per message) if we force synchronous transfer, or very high (5-20 μs per message) but then the programmer must flush buffers explicitly, increasing risk of deadlocks.

6 Future work

Our future work will concentrate on two areas: other optimisations that can be performed once the I/O-PAR regions are detected, and other requirements on data communication primitives for multimedia.

6.1 Other optimisations

Observing the results of the previous section in Table 1, we can see that there is hardly any difference in performance for communication of data between two nodes on a network, and communication of data inside a node. The reason for this is that we have optimised inter-node communication, but have not optimised intra-node communication in any way. The knowledge gained by identifying processes where streaming is allowed can also be used to optimise intra-node communication.

The optimisation that we have in mind is to *flatten* the process graph, in a manner similar to work done previously by Plevyak et al [8] on concurrent C, and Goldsmith et al [1] on CSP/Occam. The idea of flattening the process graph is that instead of having two processes run concurrently with a channel that allows them to communicate data, the compiler can actually transform them into one sequential process, where data is passed internally using a variable. This optimisation would eliminate execution of a large section of the run time support system, increasing the speed of the benchmark to around $0.5\mu s$ per "message".

This optimisation cannot always be performed at compile time. The reason is that our channels are first class and that processes may migrate. The consequences are that the compiler cannot know which channels are local, and so we have to implement this optimisation at run time. When two processes successfully negotiate that streaming is allowed, the run time support system can patch up the two sections of code responsible for consuming and producing data.

Processes, channels, and a flattening compiler might seem to be an elaborate way to efficiently implement, for example, a program modelling an orchestra. However, channels are a significantly higher level abstraction than, for example, passing arrays of samples between functions. In addition, channels allow the implementation on parallel, distributed, or mobile architectures. Concepts such as unit generators fit very well with this [2].

6.2 Other communication issues

There are other communication issues that are relevant to multimedia, which are not mentioned in this paper. Most notable are the issues of service quality and jitter freedom. Our current implementation was developed on top of UDP/IP (User Data Protocol), which does not even pretend to offer consistent quality of service. We expect to use low level network abstractions that do offer consistent quality of service. Our channel paradigm carries significant semantic knowledge down to the run time support system, and this could be made available to the service quality management layer.

7 Conclusions

Streams are a natural mechanism to represent data types such as audio and video. In this paper we have explained how to implement streams with clean (synchronous) semantics and a fast (less synchronous) implementation. Channel ends, also known as ports, are first class so that processes and ports can migrate maintaining the communication structure.

The implementation of our channels is relatively straightforward. This is due to the synchronous semantics. The only time where we switch to a non-synchronous communication mode is when we know that the semantics of a process are not affected. The class of processes for which this can be proven contains many if not most multimedia algorithms.

When a compiler spots a process which can be implemented using streaming communication, the compiler adds calls to the run time support system to loosen the synchronisation model in that section of code. The run time support system will switch dynamically to a streaming implementation if both ports of a channel are part of a process that can stream.

When adding the streaming communication we only had to extend parts of the run time support system. The vast majority of the run time support system was not affected. In particular, ports that are about to migrate are in a completely synchronous mode, hence the protocol for migrating a port is not affected.

The performance figures show a dramatic improvement in performance in inter-node communication. We are planning to extend our optimisations to intra-node communication, in order to speed up communications between processes running on the same node. We hope that we can effectively eliminate most calls to the run time support system when the system detects that two processes can stream data within a node.

References

[1] M. Goldsmith. The Oxford Occam transformation system, 1988.

[2] 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.

[3] D. May and H. L. Muller. Icarus language definition. Technical Report CSTR-97-007, Department of Computer Science, University of Bristol, January 1997.

[4] M. D. May, P. W. Thompson, and P. H. Welch, editors. *Networks, Routers & Transputers*. IOS Press, 1993.

[5] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, Sept. 1992.

[6] H. L. Muller and D. 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.

[7] L. L. Peterson and B. S. Davie. *Computer Networks*. Morgan Kauffman, 1996.

[8] J. Plevyak, V. Karamcheti, X. Zhang, and A. A. Chien. A hybrid exectuion model fo fine grained languages on distributed memory multicomputers. In *Supercomputing '95*, pages 1–18, San Diego, CA, Nov. 1995.

[9] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[10] P. H. Welch. Emulating digital logic using transputer networks. In *PARLE '87, LNCS 258/259*, pages 357–373, Eindhoven, NL, June 1987. Springer Verlag.