A PROTOCOL AND MEMORY MANAGER FOR ON-CHIP COMMUNICATION

K.G.W. Goossens

Philips Research Laboratories, Eindhoven, The Netherlands
Kees.Goossens@philips.com

ABSTRACT

We define a protocol for on-chip communication that supports dynamic interconnect networks with global memory management based on the notion of distributed shared memory with a uniform address space. The protocol is implemented by a memory manager. It is beneficial to separate the steady-state processing (data communication with static interconnect) from changing from one steady state (or user function) to another, which may necessitate reallocation of resources or changes in the network topology.

1. INTRODUCTION

Embedded systems can often be specified as a set of communicating tasks (or processes). Tasks can be as large as MPEG encoders, transcoders, hard disk (interfaces), or be at a finer grain, such as DCT/IDCT, or motion estimation functions. The functionality offered to the user grows day by day and hence not only the number of tasks in a system increases, but task graph transitions evolve from simple mode switching to more refined dynamic behaviour (which includes, for example, switch-over effects). In this paper we define a protocol and its implementation to model and implement these dynamic networks of communicating tasks.

Tasks and their communications are often modelled with Kahn networks [4, 3]. Producers and consumers (to be more precise, ports, of which a task may have several) communicate via one-to-one FIFO channels of infinite capacity in a static network. It is an abstract model without any notion of time or resources (Figure 1A).

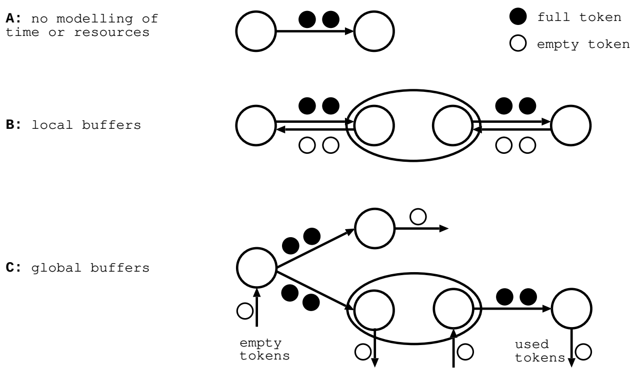

Figure 1: No, Local, And Global Buffers

1.1. Time

The main advantage of the Kahn process model is the deterministic behaviour resulting from the read primitive that blocks on absence of data, and the non-blocking write. While this is a natural model for signal processing, many functions cannot be modelled easily. For example, an MPEG decoder expects its incoming data to arrive with a certain (average) bandwidth. If no data arrives in time alternative action must be undertaken (such as redisplaying the last correct picture). Time is significant not only for the presence or absence of data, but also for changes in the network topology (enabling and disabling communication) and resource allocation. This cannot be modelled either.

1.2. Resources

In the Kahn model a write operation is non-blocking because FIFOs have infinite capacity. But in an implementation resources are limited, so that buffers and communication bandwidth must be used efficiently. A one-to-one communication channel can be efficiently realised with local buffering, as shown in, for example, [8] (Figure 1B).[1] Note that producer, consumer, and channel (with its buffers) form a triad which introduces several disadvantages. First, data cannot be passed along from one channel to another without copying of data because full and empty tokens circulate, usually FIFO-fashion, on one channel. Filters, (de)multiplexers, and shufflers (such as MPEG's BBP→PBB shuffle) are then more expensive to implement [5]. For the same reason, run-time choice of token size is most likely precluded. Second, sending data to several consumers (multicast) implies duplicating the data because buffers cannot be shared. Third, buffer usage cannot be optimised globally (over channels). For example, the number of tokens in the system as a whole could be considerably smaller than the sum of the worst-case token use of all channels. Global buffer management does not suffer from these drawbacks (Figure 1C). The global memory manager receiving and providing empty tokens is not shown because it is not a user task.

2. THE ARACHNE PROTOCOL

We provide a communication protocol, named Arachne, for the specification and implementation of networks of communicating tasks where time and resources are taken into account (e.g. is there sufficient space or data, adding and removing channels, allocating and freeing buffers, deadlines, share and/or distribute buffers and bandwidth over channels or producers). Our protocol facilitates the modular construction of tasks by shielding them as much as possible from communication intricacies such as buffer management and reconfiguration. At the application level (that is, where tasks are composed to perform the function requested by the user), sufficient facilities must be present to analyse and manage (reconfigure) the collection of autonomously communicating tasks without unduly disturbing the system. We advocate a separation between the work to be done in the *steady state* (data communication) and that done for administration or reconfiguration. The

[1] We have drawn two arrows, for full and empty tokens, to indicate a single channel.

former should be transparently performed by the protocol and not burden the application, which should concentrate on the latter. For example, the use of DMA to implement communication channels requires periodic intervention of a central CPU, which must know and keep up with the communication speeds of the channels, even in the steady state. We want to avoid this.

2.1. Target Architectures

Due to the global nature of buffer management we concentrate on a centralised implementation of the Arachne protocol. The application and all producers and consumers communicate via a control network with a token manager. The data transport to and from buffers takes place over a data network without intervention of either token manager or application. Control and data communication networks could be combined, of course. All data communication takes place via distributed shared memories that can be addressed within a uniform address space. It is not essential over what kind of structure communication takes place; it could be a bus, token ring, packet-switched network, and so on. Figure 2 shows two possible architectures.

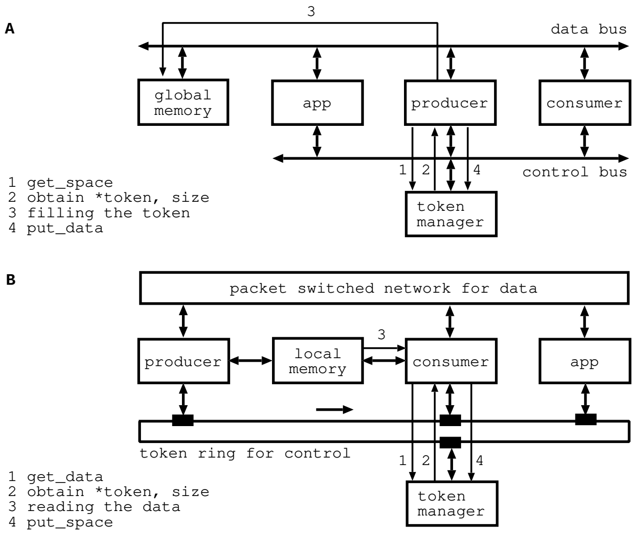

Figure 2: Sample Architectures

2.2. Concepts

In this paper, our world consists in independently executing tasks that can communicate via ports. A *port* comes in two flavours, producer and consumer; a task may have any number of either. Producers generate a *stream* of data, i.e. a finite or infinite sequence of *tokens*. Tokens are used to transport data from a producer to a consumer; they consist of a reference (pointer) to a buffer (a contiguous range of memory locations) together with its size. *Pools* are collections of tokens. It is a task of the application (which has an overview of the whole system) to determine how many tokens, and of what sizes are allocated to which pool. One strategy could be to allocate all tokens of a particular size to one pool, so that a producer can easily request a token of the appropriate size. An alternative is to group tokens based on their physical memory location. For example, a pool could contain tokens mapped to the local memory of a producer (and thus fast to use for that producer; Figure 2B), another pool could correspond to a global on-chip memory (and perhaps requires a slower bus to write to; Figure 2A), and yet another pool could be in off-chip memory. It is essential to have a uniform address space that allows this kind of mapping. A producer can decide at run time from which pool to request a token. A *credit* mechanism [6] allows for limiting the amount of buffer space a producer can use at a given point in time. It also serves to regulate the data rate of a producer if it is faster than its consumers.

A port autonomously indicates its willingness to start or to stop creating or using data. However, data production and consumption only starts when a communication *channel* has been added between a producer and a consumer; this is done by a third party (the application). A port can be connected to several channels, this then corresponds to multicast for producers and merging input streams for consumers. Tokens arriving over several channels at a consuming port are interleaved in the order they arrive in. One of the principles of Arachne is that all activity that is of no immediate interest to a port will not impinge on it. Adding (or removing) all but the first (or last) channel is therefore invisible to a port because it can continue producing (or consuming) its data stream. Sending tokens to more than one consumer, or merging data streams from several producers is a protocol service, and is performed transparently. Handing over a token is asynchronous in the sense that there is no point in time where the control flows of producer and consumer coincide to exchange the token.[2] This allows for pipelining of producer and consumer to decouple their respective data rates, and also enables efficient multicasting. A channel can be removed without loss of data before a consumer has read all pending tokens. This allows for reconfiguration of connected networks in which at no time all tasks are idle [7]. As a protocol Arachne shares many characteristics with Manifold [1].

2.3. Application Interface

There are three kinds of commands that a port can use: those for basic data communication (obtaining empty or full tokens, sending full tokens, or returning emptied tokens); commands dealing with dynamism (starting, stopping); and finally, what could be termed root, administrative, or application commands (adding and removing channels, adding and removing tokens from the system, giving credits to producers, and status commands for observing the system). Although every port can execute every command, it is wise to separate basic functionality (encoding, filtering, and so on) from system-wide application-level functionality (determining the overall functionality that must be delivered to the user, and quality of service issues). The third category of commands is most appropriate for the latter.

All commands in the following overview identify the port through its first argument `me` (omitted from function definitions for brevity). The deadline `d` will be discussed later. Function arguments preceded by an asterisk are return variables.

`get_space(d,pool,threshold,*token,*size)` Wait until at least `threshold` bytes of free space are available in pool `pool`, and return the first empty token. The port is now the *owner* of this token until it is reused (its life-time ends), and until that time pays for it out of its credits.

`put_data(d,token,bytes)` Send (or resend) `token` to connected consumers, i.e. indicate that the token with `bytes` bytes has been (partially) filled or modified and is ready to be consumed.

[2] Synchronous hand-over is also supported, but is a derived feature. It can be used to flush pipelines, and to synchronise control flows.

`get_data(d,threshold,*token,*bytes)` Wait until at least `threshold` bytes of data are pending. Optionally receive the first token on return; its ownership can also be requested. Note that modification of the data is only legal for the unique owner of a token.

`put_space(d,token)` Processing of token `token` is complete; it has been (partially) read and/or modified, and is no longer needed by this task.

`start(d)` Indicate a willingness to start production or consumption of data; wait until started.

`stop(d)` Stop producing or consuming data; sever all outgoing (resp. incoming) channels of a producer (resp. consumer). It is equivalent to a `remove_channel` for each channel.

`add_channel(d,prod,cons)` Connect the producer and consumer, if they are willing to communicate. They will start (become active) if this is the first channel to be added.

`remove_channel(d,prod,cons)` Disconnect the producer and consumer; if this is the last channel of a producer port it will be switched off. If it is the ultimate channel of a consumer, it will end the input stream.

`set_credit(d,prod,credit)` Set the maximum number of bytes that producer `prod` can use at any point in time to `credit`. With no limiting credit a producer could use all buffers and perhaps impede progress of the rest of the system. A similar credit system could be implemented for bandwidth regulation.

`alloc_token(d,pool,token,size)` Declare the memory region of `size` bytes starting at location `token` for use as a token in pool `pool` managed by the token manager.

`free_token(d,pool,*token,*size)` Remove a token from pool `pool`. Tokens can be added and removed during normal operation.

object`_status(d,...)` Obtain the status of the *objects* in the system, where *objects* are ports (off, waiting, or active), channels (the interconnect matrix of producers and consumers), space (free bytes per pool), data (bytes used by producers, pending bytes in queues of consumers), and commands (have ports made progress, and so on; can be used detect dead-lock, for example). The commands wait until there has been a change in the status of the *objects*. An application would typically analyse the system at regular intervals and make adjustments like adding or removing tokens, (dis)connecting producers and consumers, etc.

Almost all commands can take some time to complete. Examples are waiting until there is enough space or data, and waiting for ownership of a token. By giving a deadline the time spent after which a command returns can be limited. An infinite deadline corresponds to blocking, a deadline of zero means fast polling. Intermediate values are useful because they avoid the need for polling, reducing memory contention and network use. What's more important, the concept of a deadline provides for a more natural API, at a higher level.

This is a typical producer task, always ready to produce data:

```
while true do
 start(me,blocking);
 repeat
  r = get_space(me,blocking,p,0,&token,&size);
  if SPACE(r) then
   fill token[0..bytes-1] for bytes <= size;
   r = put_data(me,blocking,token,bytes);
  end
 until EOS(r) or we want to stop;
 if we want to stop then
  stop(me,blocking);
  /* else someone else already stopped us */
 end
end
```

Figure 2 shows the execution of some commands in an architecture for a producer and a consumer. An internal buffer (scratch pad), or space for dynamic data structures is obtained like this:

```
r = get_space(me,blocking,pool,minbytes,
             &token,&size);
... use token, but never send it ...
r = put_space(me,blocking,token);
```

2.4. Implementation

The Arachne protocol has been implemented in C. Systems for which C descriptions of (hardware and software) tasks exists can be simulated and analysed (see the graphs in Section 3). A hardware design of the central token manager is under way. For the data and control communication existing wrappers are available for several types of software and hardware [8] that can be used with little or no modification. This means that a task written in C that uses the Arachne protocol can be used in the same form for specification, simulation, and embedded software implementation. If a task is to be implemented as hardware it must be synthesised.

3. CASE STUDY: A SET-TOP BOX

Consider a system like a set-top box containing MPEG encoder(s) and decoder(s), at least one CPU, IEEE1394 interconnect, and storage devices such as RAM, hard disk, and digital tape recorder. All

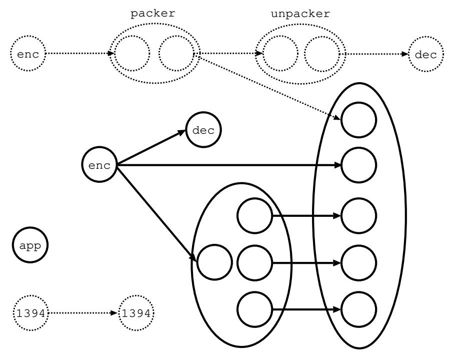

Figure 3: Set-top Box Model

these functions have widely varying IO characteristics, and need to communicate flexibly to offer the user functionality such as play, replay, pause, fast-forward, live or automatically recorded television programmes. Systems such as these can be simulated using Arachne to verify their functionality, and to obtain timing and resource figures feedback like Figures 4 and 5 to dimension memories in an implementation.

The model of a simplified system is shown in Figure 3. The dashed subsystems are active only part of the time. We see two MPEG encoders that produce a variable-rate time-stamped partial packetised transport stream (pTS+). The functionality of the packer and unpacker is not relevant here. The programme identifier filter demultiplexes the pTS+ stream in its constituent audio, video, and teletext substreams. This can be done without copying the data;

also, only part of the data needs to be read to determine the output stream. The hard disk is data-driven: it waits until at least 2Mb of data are available on an input port and then uses it as quickly as possible.

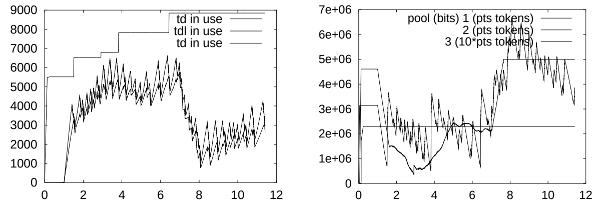

Figure 4: Resource Usage (Left), Free Space in Pools (Right)

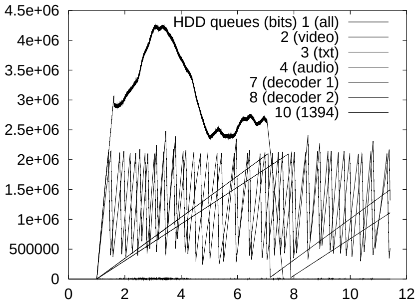

Figure 5: Pending Data Queues

The encoders use tokens of 188+4 bytes (pTS packet with 4 bytes for the time-stamp). The IEEE1394 ports use packets ten times larger. In the resource use graph (Figure 4) the highest line indicates the number of tokens in the system; note that tokens are added to the system while it is active. The lower of the two agitated lines shows the number of tokens in use; the higher line displays the number of tokens in all queues (with multicast, this may be higher than the amount of tokens in use). The hard disk has a very peaky IO; this can be observed in Figure 5 where the queues of the full pTS+ stream, and the audio, video, and teletext substreams are shown. The audio substream takes nearly 7 seconds to fill 2Mb; this means that the life-time of a token produced by the encoder varies from 0.5 to 7 seconds, depending on whether it contains video or teletext. It also explains the marked (4Mb) increase of free space in the token pool reserved for this subsystem between 7 and 8 seconds; the token usage in the resource use graph also decreases at that point. The queue of the decoder connected to the packer (the highest graph in Figure 5) shows that substantial buffering takes place (3Mb) before the decoder starts; thereafter the variable bit rate is clearly visible. The amount of free space in the token pool reserved for the encoder-packer-unpacker-decoder chain shows the same curve, except for the increase around 3 seconds due to the addition of extra tokens (Figure 4). The pending data of the other decoder and the IEEE1394 port are hardly visible in Figure 5 because they are closely coupled to their producers.

4. RELATED WORK

We envisage that many of the tasks that are traditionally performed by a software operating system, such as inter-task communication, memory management, and memory protection, will migrate to an on-chip communication infrastructure because of the mixing of hardware and software components on a single chip. Quality of service is another aspect that must be supported by this infrastructure because it depends on the ability to observe, steer, and trade-off resources like computation, buffer space, time (latency, throughput). There is a plenitude of work published in this area.

There is a limited amount of work on memory management in hardware. In [2] a memory manager for dynamic data structures is presented, but communication is ignored. This is only a part of our work, see the scratch-pad example in Section 2.3. In [11] hardware is generated from C to deal with distributed memories for dynamic data structures (again no communication). Every data memory has its own allocator, and pointer disambiguation is partially supported. [9] contains a software approach for buffer sharing over channels. The elegant space-time memory of [10] addresses many of the issues discussed here, but needs garbage collection, which is expensive to implement.

5. CLOSING REMARKS

We have argued that modelling of systems built of dynamically communicating components requires a concept of time (for adding and removing channels and resources). Moreover, in a realisation resources are finite and must be managed (shared, distributed). The Arachne protocol offers this in a smooth trajectory from specification (without time and resources) to implementation (with time and resource constraints). Currently systems can be modelled and simulated, and implementation of efficient dedicated hardware for the token manager is in progress. The set-top box example shows how Arachne is used to analyse dynamic systems and make implementation decisions. It also shows that this is complex due to time-dependent behaviour and resource sharing. We advocate the decoupling of the steady-state processing (done by the protocol infrastructure), resource and connection management (done by the application task), and functional processing (done in the tasks) to ease system design.

6. REFERENCES

[1] F. Arbab. Manifold version 2.0: Language reference manual. Technical report, CWI, The Netherlands, June 1998.

[2] G. de Jong, et al. Background memory management for dynamic data structure intensive processing systems. In *ICCAD*, pp 515–520, 1995.

[3] E.A. de Kock, et al. YAPI: Application modeling for signal processing systems. In *DAC*, 2000.

[4] G. Kahn. *Information Processing*, chapter The semantics of a simple language for parallel processing. 1974.

[5] J. Kang, et al. Mapping array communication onto FIFO communication – towards an implementation. In *ISSS*, pp 207–213, 2000.

[6] H. Kung, et al. Credit-based flow control for ATM networks: credit update protocol, adaptive credit allocation and statistical multiplexing. In *Conf. on communications architectures, protocols and applications*, pp 101–114, 1994.

[7] J. Leijten. *Real-time constrained reconfigurable communication between embedded processors*. PhD thesis, Technical University Eindhoven, 1998.

[8] A.K. Nieuwland and P.E.R. Lippens. A heterogeneous HW-SW architecture for hand-held multi-media terminals. In *Workshop on signal processing systems*, pp 113–122, 1998.

[9] H. Oh and S. Ha. Data memory minimization by sharing large size buffers. In *ASP-DAC*, Tokyo, January 2000.

[10] U. Ramachandran, et al. Space-time memory: a parallel programming abstraction for interactive multimedia applications. In *Symposium on Principles and Practice of Parallel Programming*, May 1999.

[11] L. Séméria, et al. Resolution of dynamic memory allocation and pointers for the behavioral synthesis from C. In *DATE*, pp 312–319, 2000.