Deadlock Avoidance in Parallel Design

Colin Willcock & Peter Welch
Computing Laboratory, University of Kent at Canterbury

Abstract. In this paper we will attempt to classify and describe the different types of deadlock encountered in parallel programming. We will then go on to derive some general design guide-lines for avoidance of these deadlock problems. These design guide-lines will then be demonstrated by considering the design of the occam X server.

1 Introduction

Since occam 2 makes use of synchronized point-to-point communication, if a given process attempts to send a value along a given channel and the process on the other end of that channel refuses to accept that message then the sending process will wait for ever. In a similar manner a process waiting to receive a value on a given channel will wait for ever if the process on the other end of the channel fails to send a message. We use the term "deadlock" to refer to this situation, where one or more processes within a system are brought to a permanent standstill. In occam, all deadlocks caused by erroneously programmed allocation of shared resources (e.g. servers) have communications deadlock at their base [2] [3].

2 Deadlock classification

In this section we will discuss the four main types of deadlock-like behaviour. We will use the terms primitive deadlock, structured deadlock, primitive livelock and structured livelock to classify them.

Primitive deadlock occurs when a process within a system has, at some point in its code, the behaviour of a `STOP` process (a `STOP` process starts but never terminates). Such a process is shown in figure 1. This causes the process to refuse any further communication with its environment.

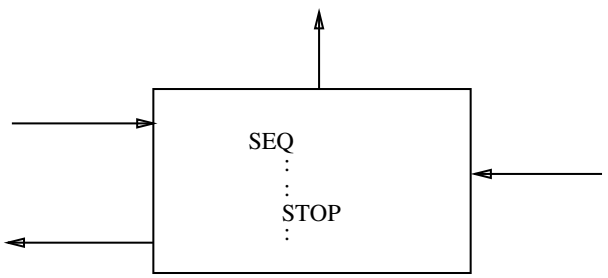

Figure 1: Example of primitive deadlock

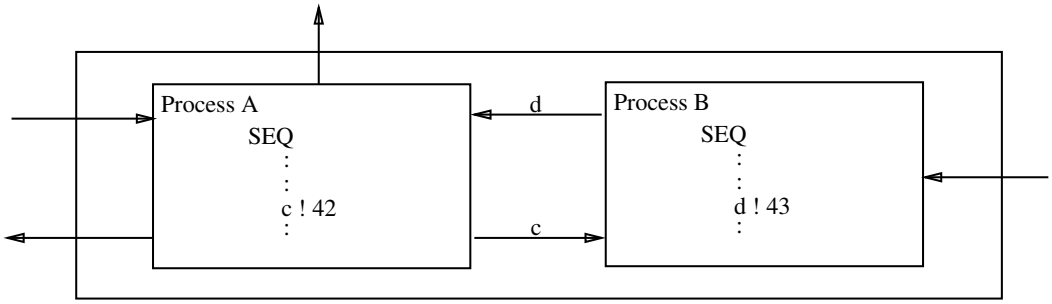

Figure 2: Example of structured deadlock

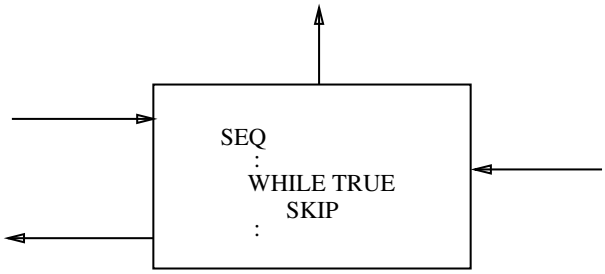

Figure 3: Example of primitive livelock

An example of structured deadlock is shown in figure 2. `Process A` has become committed to output on channel `c` for its next communication (perhaps at the start of its indicated `SEQ` loop). During this commitment period, `Process B` similarly becomes committed to output on channel `d` as its next communication. Since neither communication can complete, both processes stop and this system refuses any further communications from its environment — it is deadlocked!

It is worth noting that to an outside observer, the primitive deadlock shown in figure 1 appears the same as the structured deadlock shown in figure 2. The two situations are therefore semantically identical.

The third type of deadlock to consider is primitive livelock, this is shown in figure 3. The process shown contains an infinite non-communicating loop at some point in its execution. When the process commits itself to this infinite loop it refuses any further communications with its environment and hence the process exhibits deadlock-like behaviour.

The last example situation we will consider is shown in figure 4. This example demonstrates an instance of structured livelock. Both processes enter an infinite loop within which they only accept communications from the other process. Hence, when the two processes commit themselves into the loops, they refuse any further communication with other processes in the system and a deadlock situation is initiated. Notice that, to an outside observer, this structured livelock is indistinguishable from the other deadlock situations, discussed previously, and is hence semantically identical to them.

To their environments, deadlocked or livelocked systems appear the same — namely they fail to respond. For the processor, a deadlocked system imposes zero load, whereas a livelocked system imposes continuous demands.

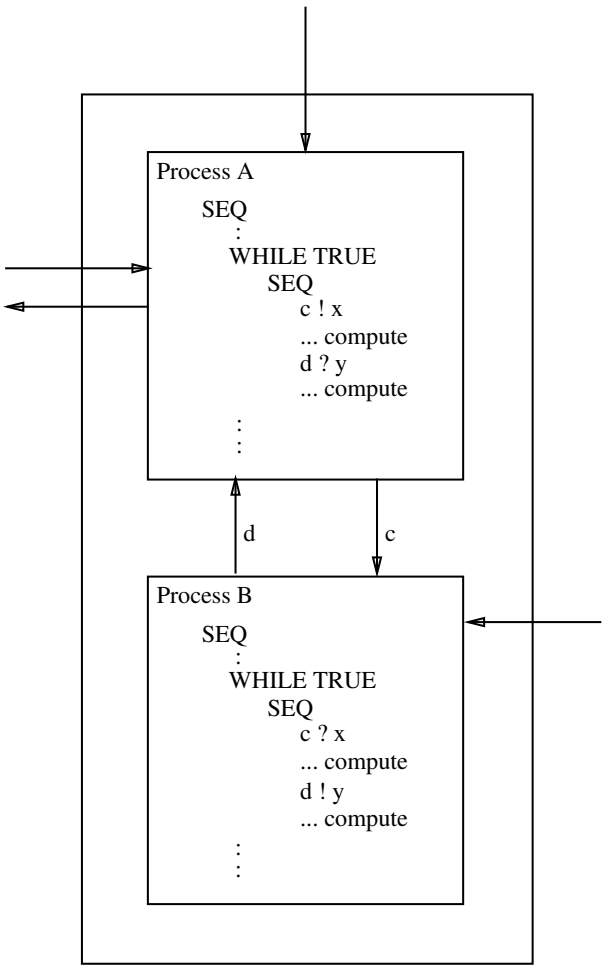

Figure 4: Example of structured livelock

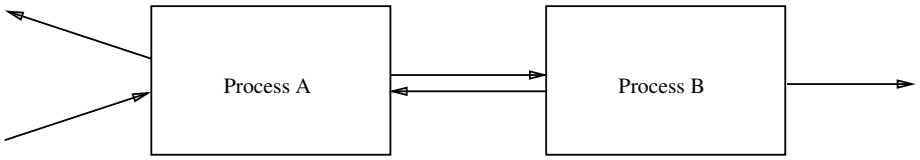

Figure 5: Example of bi-directional communication

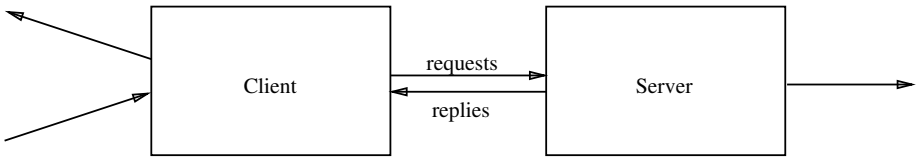

Figure 6: Example of client/server relationship

3 Deadlock avoidance methods

3.1 Introduction

Deadlock is a common design fault and its absence requires verification. One standard method of verification is testing. However, in all but the most simple design, this is impractical due to the prohibitive size of the required test data for exhaustive testing. The alternative verification approach involves an exhaustive "case-by-case" check through all the possible "categories" of state. By intelligent choosing of these "categories", exhaustive checking for a well-designed system need not be lengthy.

Livelock represents a less common design fault. Such design faults are usually caused by mis-programmed loops which fail to terminate. As in the case of deadlock, absence of livelock requires verification using similar techniques.

In order to derive some simple deadlock free design elements, let us consider the bi-directional channel pair in figure 5 connecting `Process A` with `Process B`. This configuration of bi-directional communication between parallel processes represents a common design element. As shown earlier, this bi-directional communication has the potential for deadlock if `Process A` and `Process B` both become committed to output to each other as their next communication. For a deadlock free design, we therefore need to ensure that `Process A` and `Process B` do not "spontaneously" attempt to communicate with each other. If the two processes obey a "client-server" relationship, this constraint will automatically be satisfied.

3.2 Client/server relationship

In a client-server relationship, any communications between the two processes have to follow a set pattern. This pattern dictates that communications consist *only* of the client making a request to the server followed by the server responding with an answer (or simply an acknowledgement). The server *must* reply and the client *must promise to accept* this reply. This simple scheme, shown graphically in figure 6, ensures no deadlock is possible between this client-server pair. For correctly implemented clients and servers, it should be possible to confirm these arrangements by simple inspection of their low-level code.

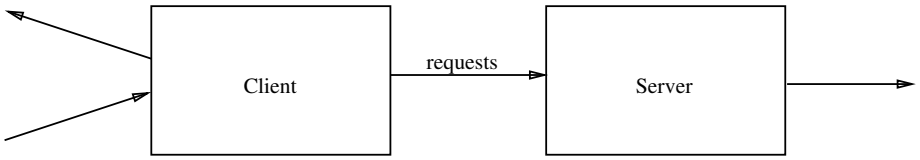

Figure 7: Single channel client/server relationship

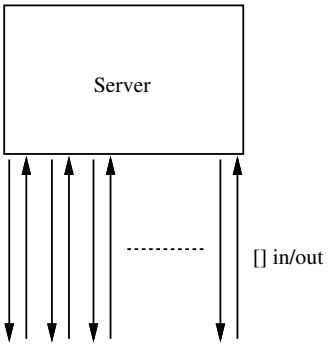

Figure 8: Example of simple server

The case of a single channel between client and server, as shown in figure 7, can be considered a degenerate version of the client-server model just outlined where no replies from the server are required. The communication between these two processes introduces no deadlock. A system such as this, can by the addition of a reply channel between server and client be transformed into a (functionally equivalent) version that fits the "standard" client-server model described above.

3.3 Simple server implementation

In the previous section we defined the basic client-server relationship. We now consider implementing a simple server for multiple clients using this model. The server process is shown in figure 8. It provides services to multiple clients, with each client connected to the server by bi-directional channels (whose usage must conform to the client-server model defined above).

Using the occam 2 language, an implementation of such a simple server would be :-

```
PROC server ([]CHAN OF REQUESTS in,
             []CHAN OF ANSWERS out)
  SEQ
    ...  initialise
    WHILE TRUE
      ALT  i = 0 FOR SIZE in
        in[i] ? request
          SEQ
            ...  compute answer (*assume always completes*)
            out[i] ! answer
:
```

It can be seen from this code that each client-server pair complies with the previously

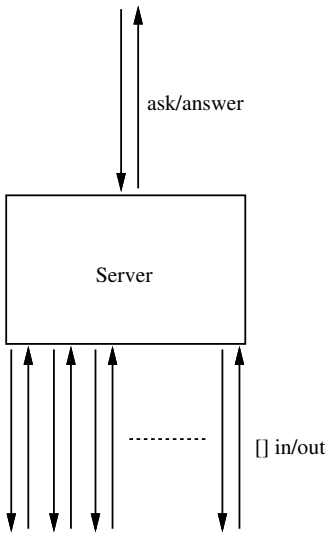

Figure 9: Example of server calling another server

defined client-server relationship: the ALT construct* selects one client to communicate with, the server receives a single request from this client and then replies with an answer.

We have just shown that, for the case of a single server with multiple clients, the client-server relationship can be upheld and hence a deadlock free design produced. In many situations however more than a single server is required. Let us now consider the situation where a given server has to call upon another server (i.e. act as a client) to help with certain requests. The process model of such a server is shown in figure 9. Such a server would block its other clients whilst waiting for the second server to respond. From the point of view of deadlock freedom, this is acceptable, as long as we know that the other server will (eventually) respond. The occam 2 code for such a server would be :-

```
PROC server ([]CHAN OF REQUESTS in,
             []CHAN OF ANSWERS out,
             CHAN OF REQUEST ask,
             CHAN OF ANSWERS ans)
  SEQ
    ...  initialise
    WHILE TRUE
      ALT  i = 0 FOR SIZE in
        in[i] ? request
          SEQ
            IF
              local (request)  -- handle request locally
                ...  compute answer (*assume always completes*)
              TRUE     -- make request to another server
                SEQ
                  ask ! request --(*assume always taken*)
                  ans ? answer  --(*assume always comes*)
```

*This construct is in fact not acceptable because it makes no guarantee to be "starvation free" [1].

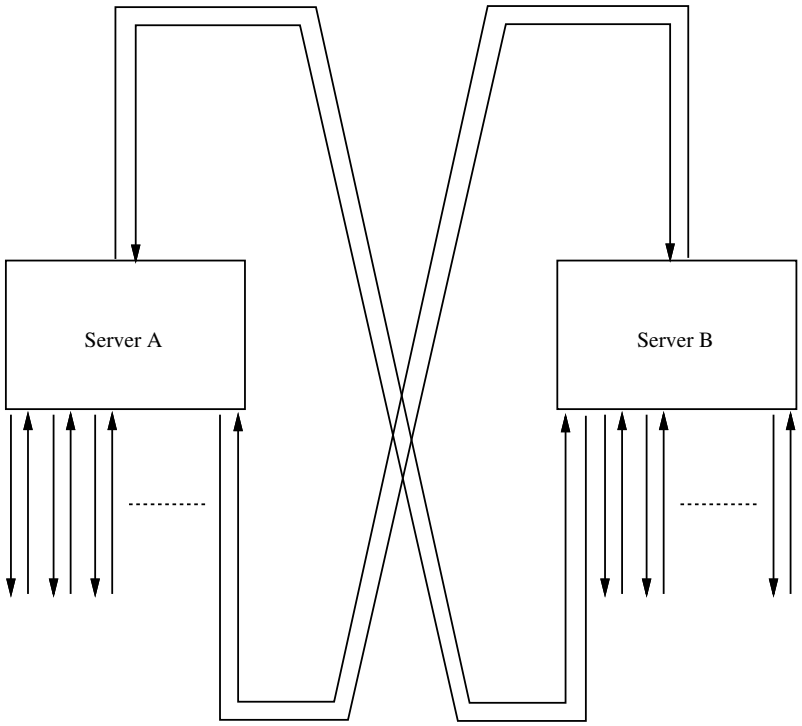

Figure 10: Example of cross mounted servers

```
            out[i] ! answer
:
```

If the second server fails to respond to a request issued by the first server then the first server's clients will be blocked permanently and hence the system enters deadlock.

3.4 Cross-mounted servers

In the previous section we considered the case of a server which calls another server in the course of satisfying certain requests. We concluded that such a situation would not deadlock as long as the second server responded. We will now consider the situation of cross-mounted servers, where *both* servers make requests of each other. Such situations often occur in practice — e.g. file-servers (and without care, they do deadlock).

In figure 10, `Server A` provides the services of `Server B` to its clients by being a client of `Server B`. In a similar manner, `Server B` provides the services of `Server A` to its clients by being a client of `Server A`. In this case `Server A` and `Server B` are said to be cross-mounted. If, `Server A` becomes committed to make a request to `Server B` and `Server B` becomes committed to make a request to `Server A`, we will enter the situation of structured deadlock described previously in section 2. Hence, using such a design for cross-mounted servers could result in the servers becoming deadlocked.

A design which provides the same functionality to the clients, but which will not deadlock is shown in figure 11. The two sets of clients now communicate with the "main" servers through intermediary `Multiplex` servers. These `Multiplex` servers forward the requests from the clients to the relevant server. The choice of relevant server can be decided by request to the `Name Server` process. This configuration is

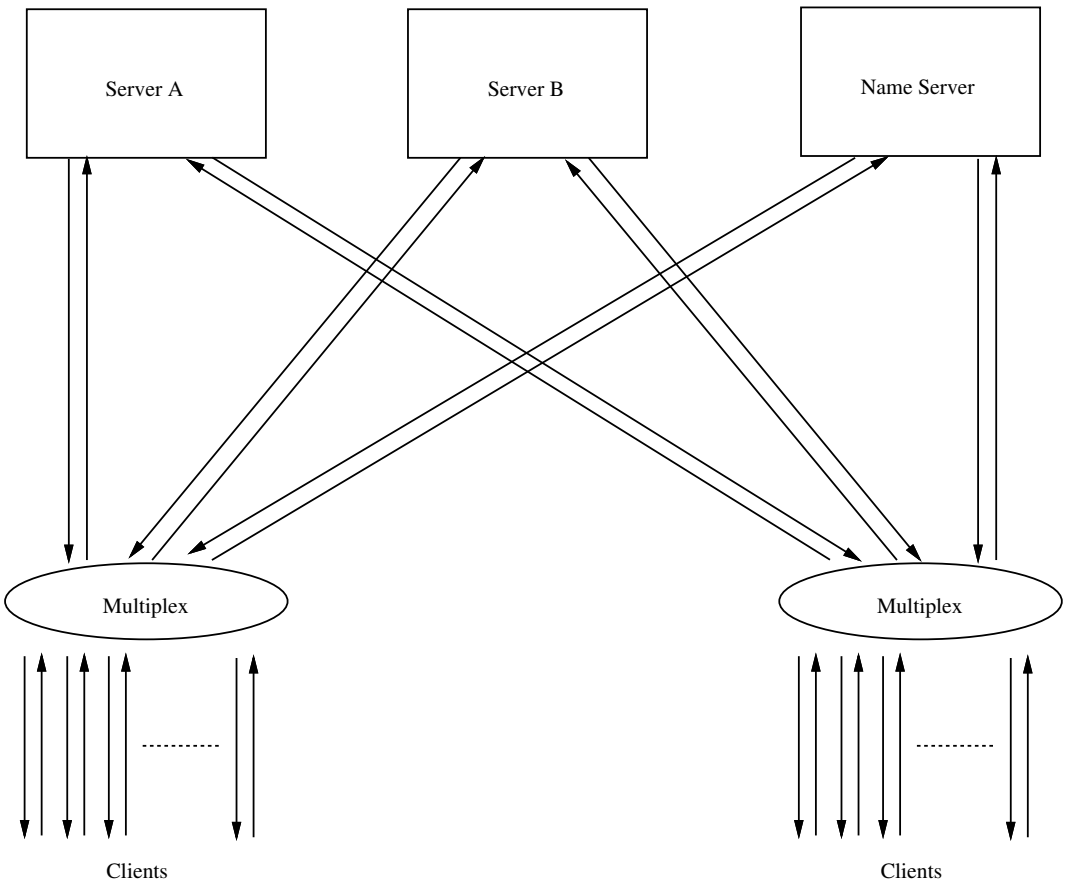

Figure 11: Deadlock free design for multiple servers

inherently deadlock free because, if the processes adhere to the client-server model discussed previously, all the servers become instances of the simple server which we have shown to be deadlock free in section 3.3.

3.5 Generalized deadlock free design

Having considered the design issues associated with specific examples we now consider a generalized heuristic for deadlock-free client-server design.

Consider the arbitrary set of communicating client-server processes shown in figure 12. Any pair of processes, using the client-server notion of bi-directional communication, has a client-end and a server-end (denoted by the letters `c` and `s` respectively in figure 12). If we represent such communication by a single arrow from the server-end to the client-end we can represent any set of such processes as a graph. Within this graph, the processes are displayed as nodes and the communications arrows constitute the edges. Such a graph for the example processes is shown in figure 13. This graph can then be partially ordered (as long as it contains no cycles) whereby the later items in the ordering are only clients of servers already included in the list. Given this set of ordered processes we will now prove the following theorem :-

Theorem

Any collection of processes, Such as those shown in figure 13, which are connected

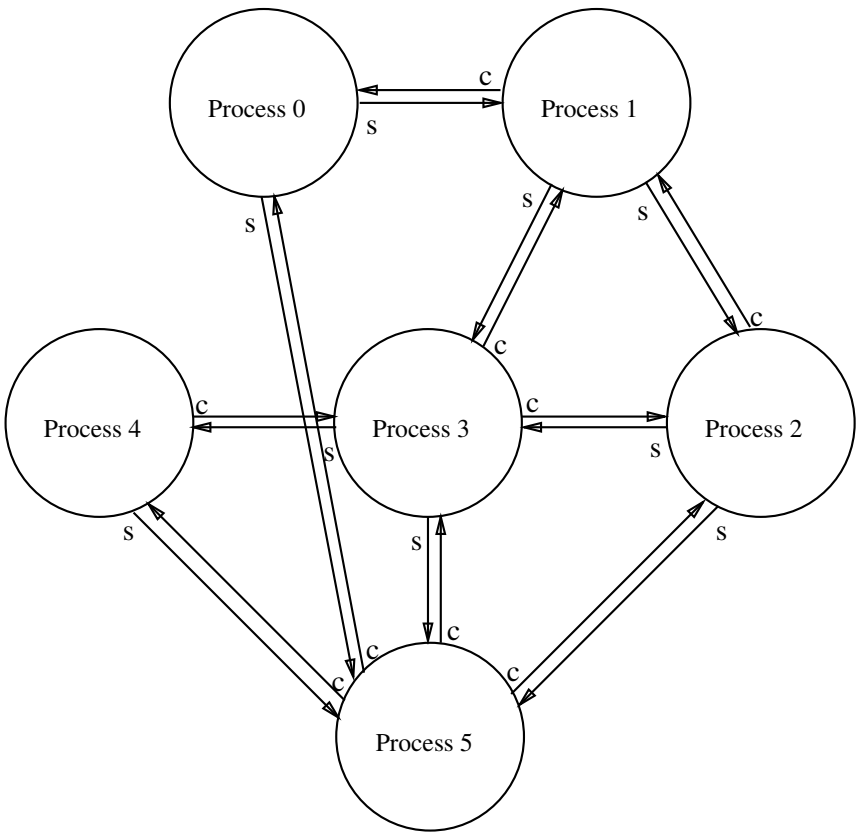

Figure 12: Example set of communicating client-servers

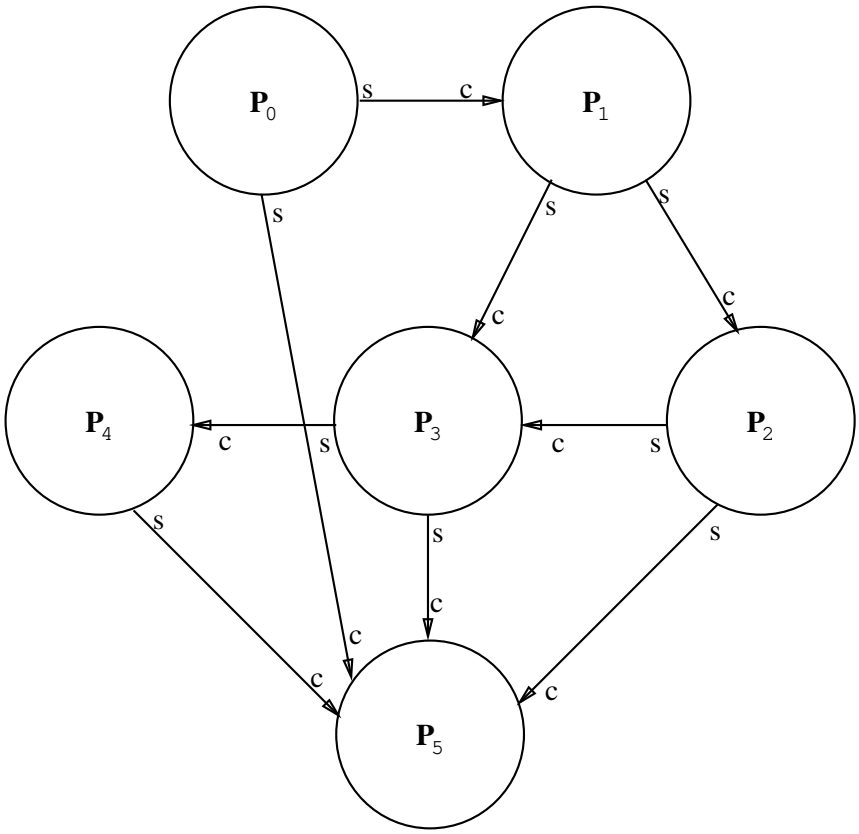

Figure 13: Example set of communicating client-servers transformed into an ordered graph

by bi-directional channels whose usage conforms to the client-server model (stated above) and has acyclic topology (in terms of the derived graph mentioned above) has no deadlock.

Induction Hypothesis H(i) $\equiv$ ($\forall$j such that $0 \leq j \leq i$),
the process $\mathtt{P}_j$ will always respond to a client signal (i.e. it will accept it and generate a reply) within a finite time.

Base Case: establish **H(0)**
Since, (from figure 12), $\mathtt{P}_0$ handles all requests itself (there are no earlier servers to which it can make a request), by the client-server logic we can guarantee it will always respond to a client signal.

Induction step: assume $\mathtt{H}_i$, establish $\mathtt{H}_{i+1}$
Assuming $\mathtt{H}_i$ we only have to check the behaviour of process $\mathtt{P}_{i+1}$. Since this process too has the client-server logic we can follow its code on response to a client request. With reference to the example server in section 3.3 the response will be :-

either Generate reply for request locally. In which case the response will be in a finite time.

or Generate a reply by making a request to a server $\mathtt{P}_j$, where $j \leq i$. But hypothsis **H(i)** says that we will receive a reply from this server ($\mathtt{P}_j$) within a finite time. Therefore by passing this response back we have now fulfilled our contract with our client in a finite time.

Conclusion: ($\forall i \geq 0$)(**H(i)**)

In section 3.1 it was stated that the only viable method for verification of the absence of deadlock from a set of processes involves an exhaustive "case-by-case" check through all "categories" of state for that system. By application of the above theorem we now only have to check that each process conforms to the client-server paradigm defined in section 3.2 and that the client-server topology is acyclic. The first checking involves both insuring that communication follows the defined behaviour (of the client-server model stated above) and that the local serial code that drives the channels contains no infinite loops. The order in which the constituent processes are checked is of no importance and the checking only involves local inspection of serial code. The second check involves local inspection of the network topology (i.e. no consideration of the logic within the nodes). These two checks, performed separately, are trivial to make

4 Deadlock avoidance in X server design

The X window system (abbreviated to X) is a network oriented windowing system developed for bitmapped graphics displays. The graphics displays supported by X include colour, monochrome and gray scale. The network communications of the X window system is based on the client-server paradigm[†]In X, the client is the application

[†]It should be noted that the meaning of the "client-server" model in this context is not the same as that used in the previous section considering deadlock free design. Although the functional relationship

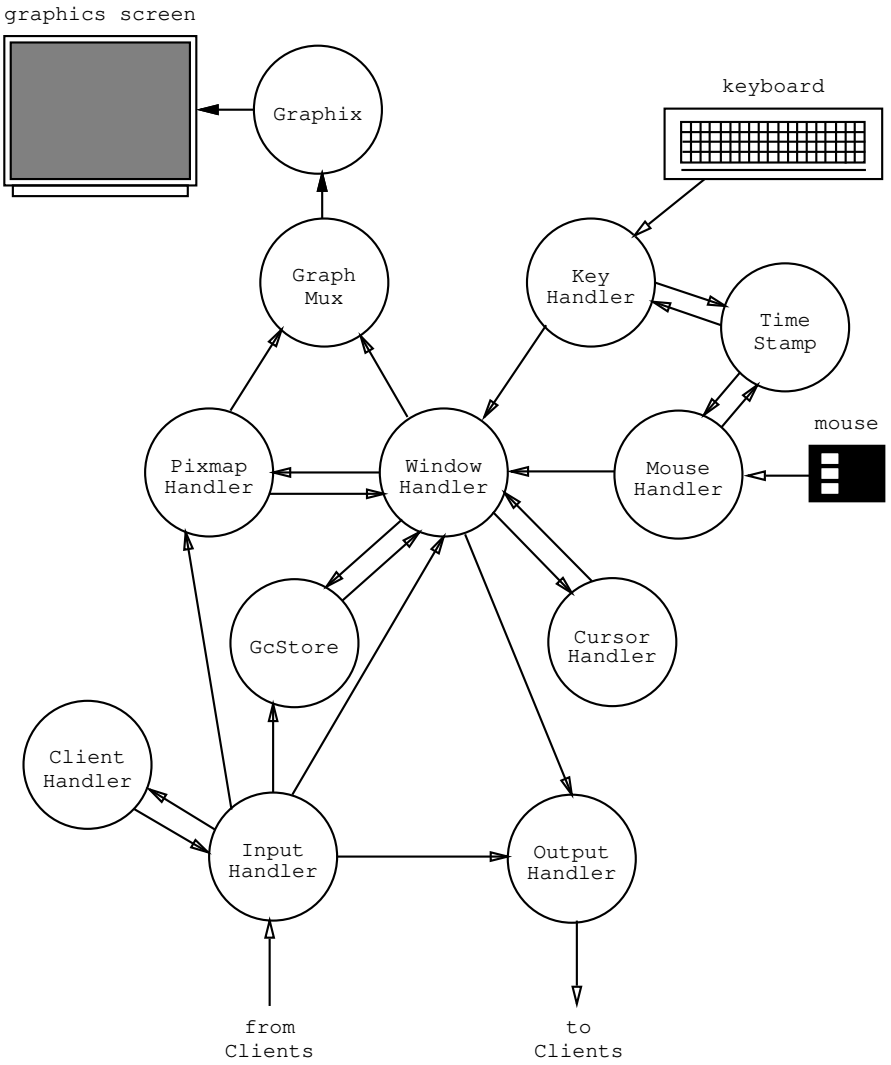

Figure 14: Overall process model of occam X server

program making requests of the server to draw windows, text and other objects. The server process (referred to as the X server) controls and draws onto the display screen. As well as the display screen, the X server also controls the keyboard and mouse. The X window system is defined by its network protocol of asynchronous, stream-based communication between client and server.

The main design aim was to implement a parallel X server that would be scalable, both in terms of performance and functionality, by the addition of multiple processors. Another design requirement was that the X window system be secure and maintainable. One aspect of this is avoidance of deadlock within the system and we will see that the software designs specified comply with the explained strategies for deadlock avoidance.

It can be seen that the derived process structure for the occam X server (shown in figure 14) consists of an acyclic topology — i.e. it contains no cycles of client-server

between client and server processes is the same in both instances, namely the client makes requests of the server, in the X window system there is no implicit communications pattern — i.e. in the X window system the request and replies are asynchronous.

relationships. This avoids the possibility of cyclic deadlock and allows the system of processes to be linearly ordered with respect to these relationships. Remembering that this linear ordering involves listing all processes in such a way that all "server" processes for a given "client" should be placed before that "client", one such order would be :-

1. `ClientHandler`: this process is just a server to `InputHandler`.
2. `TimeStamp`: this process acts as a server to `keyHandler` and `MouseHandler`.
3. `Graphix`: this process behaves as a black hole, always accepting graphics requests. It behaves as a server towards `GraphMux`.
4. `GraphMux`: this process can be considered a client of `Graphix` and a server to `PixmapHandler` and `WindowHandler`.
5. `OutputHandler`: this process behaves as server to `WindowHandler` and `InputHandler`.
6. `PixmapHandler`: this process acts as a server to `WindowHandler` and `InputHandler` and as a client process to `GraphMux`.
7. `CursorHandler`: acts only as a server to `WindowHandler`.
8. `GcStore`: this process acts as a server to `WindowHandler` and `InputHandler`.
9. `WindowHandler`: this process acts as a server for `InputHandler`, `MouseHandler` and `KeyHandler`. It also acts as a client process to `GraphMux`, `PixmapHandler`, `GcStore`, `OutputHandler` and `CursorHandler`.
10. `KeyHandler`: this process acts as a client to `TimeStamp` and `WindowHandler`.
11. `MouseHandler`: this process acts as a client to `TimeStamp` and `WindowHandler`.
12. `InputHandler`: this process behaves as a client to `ClientHandler`, `PixmapHandler`, `GcStore`, `WindowHandler` and `OutputHandler`.

From the theorem, shown previously, we can deduce this system is free from deadlock. The first stage is to check the high-level design to ensure that there are no cycles of client-server relationships. The second stage is to check the low-level serial code of each process adheres to the client-server paradigm.

5 Conclusions

In this paper we have initially defined the term deadlock and then continued by describing simple generic process pairs which, by conforming to a client-server model, are inherently deadlock free. Acyclic topologies of these simple client-server process pairs are then demonstrated in the design of a highly concurrent X window system.

We have also demonstrated at least one example of converting a cyclic topology (cross-mounted servers) into a functionally equivalent acyclic topology.

References

[1] G. Jones. Carefully Scheduled Selection with ALT. *OUG Newsletter*, 10, 1989.

[2] W. Joosen and P. Verbaeten. A deadlock detection tool for occam. In *Developing Transputer Applications, Proceedings of the 11th Occam Users Group Technical Meeting, Edinburgh*. IOS Press, Netherlands, 1989.

[3] A. W. Roscoe and N. Dathi. The pursuit of deadlock freedom. Technical Report PRG-57, Oxford University Computing Laboratory, Programming Research Group, 1986.