An Optimised Implementation of the DCAN Divider Server on the Nemesis Operating System

Simon Crosby and Paul Menage

Tuesday 3$^{\underline{\text{rd}}}$ March 1998

Abstract

The *Divider Server* is one of the central parts of the *Open Service Support Architecture* around which the DCAN project is based. The initial implementation of the Divider Server had no support for Quality of Service guarantees between multiple clients. This document reports on an optimised reimplementation of the Divider Server on a platform capable of providing such guarantees.

1 The Divider Server

The Divider Server presents a network switch switch to a set of clients in the form of separate *switchlets*[1]. Each client has control over a subset of the resources of the switch. The main function of the Divider Server is to police clients' requests and prevent clients from accessing resources which have not been allocated to them. A full description of the generic form of the Divider Server is given in [2]. This also contains a discussion about the desirability of Quality of Service guarantees to clients, but comments that the initial implementation did not support such guarantees due to the OS and distributed environment being used.

2 Nemesis

The work described in this deliverable has been carried out on the Nemesis operating system [3, 4]. Nemesis is a vertically structured single-address space microkernel system developed at Cambridge University which aims to reduce application cross-talk by providing Quality of Service guarantees for resources such as processor time, network bandwidth, memory, etc.

2.1 Nemesis kernel

The Nemesis kernel is light-weight, dealing only with memory protection, process scheduling and interprocess event notification. The process scheduler makes the distinction between *guaranteed* time, which is processor time allocated to fulfill the process' QoS contract, and *extra* time which is spare processor resource allocated to those processes which can utilise it. This allows applications to adapt to variations in available processor time, such as by producing complete but low-quality results rather than high-quality but incomplete results.

2.2 Nemesis philosophy

Nemesis tries to reduce QoS crosstalk between applications by removing shared servers from the data path wherever possible. For example, whereas traditional operating systems multiplex at each layer in a protocol stack, and only pass results to the user domain at the application or presentation layers [5], under Nemesis network traffic is multiplexed only at a network device driver, which exists only to check permissions on transmitted and received packets. All protocol operations are dealt with by the applications, using shared library code [6].

This has the effect of reducing *crosstalk* between applications — packets destined for an application with a high QoS guarantee will not be held up behind those of an application with a lesser guarantee. Also, if an application is falling behind its incoming streams of data, the data can be thrown away when (or, given sufficiently intelligent hardware, before) it comes off the wire, rather than being thrown away just below the presentation layer as happens in many existing OSs.

3 The Divider Server on Nemesis

Implementing the Divider Server on Nemesis gives significant advantages in terms of the service guarantees that can be provided to the clients of the network. Figure 1 shows the structure of the Divider Server running on Nemesis. Various aspects of the architecture are discussed in this section.

3.1 Client Creation

As each client is created by a network controller such as the NetBuilder, an Ariel[7] interface is constructed for the client. This can be over any supported Nemesis transport — currently there is support for IIOP over TCP (CORBA[8]), GSMP over ATM and a Nemesis specific lightweight RPC transport over UDP.

The network controller can specify a set of Quality of Service parameters for the client at this point. Connections will be made to the relevant device driver to obtain *Rbuf*[9] data channels to and from the network. The server will request from the device driver sufficient network QoS to enable it to meet the guarantee given to the client.

The incoming data channel will have an event endpoint associated with it — this is used to bind the client into the scheduler.

3.2 Scheduler

The choice of scheduler is not dictated by the architecture of the Divider Server. Possible implementations might be:

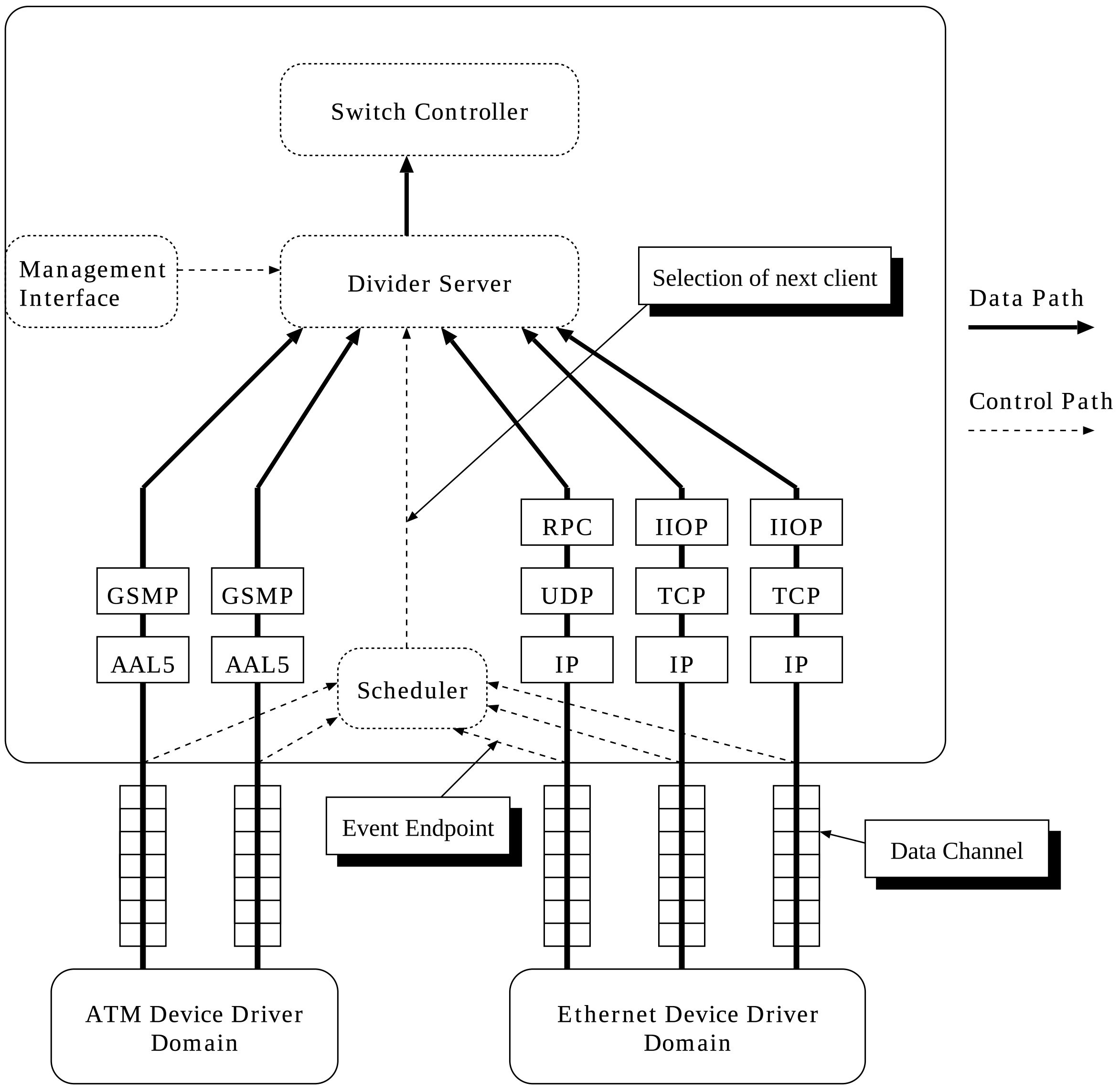

Figure 1: The Divider Server running in a Nemesis domain

Round-Robin The scheduler would serve one request from each client in turn while there were outstanding requests in its queue. This is a simple and fast policy, and permits a fair distribution of the Divider Server's time between all clients. However, it does not permit guarantees to be given to clients.

Priority-based The scheduler would divide clients into classes of different priorities, and serve requests in a round-robin style from the highest priority class with outstanding requests. This permits high priority clients to be given a faster service — but it still does not give proper guaranteed bounds on service latency, particularly for the lower priority clients.

Earliest Deadline First Each client is allocated a slice of time, and a period over which this time can be used up. When a client's time has run out in a particular period, it is blocked until its next period starts. Any left over time at the end of a period is discarded. See [10] for a fuller description. This scheduling scheme permits all clients to be given a guarantee — even a low bandwidth client can be sure that they will receive some regular processing.

3.3 Operation

When incoming requests (in the form of Ethernet frames, AAL5 packets, etc) are received at the appropriate device driver, they are delivered straight to the Divider Server. No protocol processing is performed by the drivers — rather, the raw packets are delivered. This prevents crosstalk between clients in the network device driver.

Delivering the packets to the Divider Server causes the event endpoint associated with the client to be activated. The scheduler will then mark that client as having an outstanding request. At some point in the future, depending on the scheduling policy in use and the client's scheduling parameters, the client will be allocated time by the Divider Server. At this point, the protocol processing for the received packets is performed, and unmarshalling of a request with its arguments occurs. The request is turned into an Ariel[7] invocation on the transport-independent part of the Divider Server, containing the policing functions.

If the request is valid (i.e. refers to resources owned by the client) then it is passed on to the switch controller to actually set up the VCIs/VPIs requested.

After the request has been completed, the scheduler charges the client for the time taken to carry out the request, and then makes a new scheduling decision to select a new client to serve. (This could result in the same client being served again, if it has appropriate QoS guarantees and more requests outstanding).

4 Advantages of the Nemesis approach

This section describes some of the advantages experienced by implementing the Divider Server architecture on Nemesis.

4.1 Greater efficiency

Since the Nemesis network stack is optimised for a Single Address Space architecture, it can avoid copying data more than is necessary. In a typical Unix implementation, copying could take place at the following points:

1. The network interface transfers the data from the network in to main memory.

2. The device driver and kernel copy the data (possibly multiple times) while reassembling/processing packets.

3. The kernel copies the data to a user buffer.

4. The user unmarshals the data to form an Ariel invocation.

Under Nemesis, the single address space model means that there is no need for a copy between kernel and user buffers. The Nemesis network protocol architecture (see [6]) processes data in place wherever possible. This means that only the copies at stages one and four are needed. This produces a substantial performance increase, reducing latency and increasing throughput.

4.2 Clients pay for transport overheads

When deciding what transports to allow clients to use, on a typical system where protocol processing/unmarshalling may be carried out 'below the covers' it is difficult to account for overheads incurred by using inefficient transports. However, restricting client to using efficient transports will reduce interoperability — typically, open standardised transports such as IIOP (used by CORBA) tend to be inefficient due to their need to cover all possibilities.

In the Nemesis implementation, a client is charged for *all* the work carried out on its behalf; therefore a client who chooses to use an inefficient protocol will see a correspondingly smaller number of requests being processed in a given charged period of time.

4.3 Prevention of Denial of Service attacks

In a typical system, if a client were to start producing requests at a very high rate, the kernel would spend large amounts of time doing protocol processing on the incoming packets before having to throw the data away since the Divider Server would not be making sufficient progress due to being starved of time. Since the kernel has no knowledge of the scheduling policies being used by the Divider Server, it cannot prevent crosstalk between clients.

Under Nemesis, the misbehaving client's data channel would quickly fill up, causing the network device driver (or the network interface, if it has sufficient intelligence) to discard packets at the lowest possible level. Packets for well-behaving clients would not be affected.

4.4 Multiple Divider Servers

In the same way that multiple clients can be protected from one another using the described architecture, multiple Divider Servers can be run on the same Nemesis machine, and each given a certain CPU and network bandwidth allocation. The two Divider Servers will then run without interference between one another.

5 Conclusion

This paper has described an architecture for a Divider Server supporting multiple clients with differing Quality of Service guarantees and running over different transports. The details of the architecture have been presented, and the advantages given by this approach have been discussed. The authors believe that it is clearly the case that just as new networking technologies must be able to offer QoS guarantees in network bandwidth, they must be able to offer QoS guarantees to clients wishing to set up connections using that bandwidth.

References

[1] Kobus van der Merwe. Switchlets and Dynamic Virtual ATM Networks. *Submitted to ISINM'97*, July 1996.

[2] Kobus van der Merwe. Open service support for atm. Technical report, University of Cambridge Computer Laboratory, September 1997.

[3] Timothy Roscoe. The Structure of a Multi-Service Operating System. Technical Report 376, University of Cambridge Computer Laboratory, August 1995.

[4] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996. Article describes state in May 1995.

[5] David L. Tennenhouse. Layered Multiplexing Considered Harmful. In *Protocols for High Speed Networks*, IBM Zurich Research Lab., May 1989. IFIP WG6.1/6.4 Workshop.

[6] Richard Black, Paul Barham, Austin Donnelly, and Neil Stratford. Protocol Implementation in a Vertically Structured Operating System". *Local Computer Networks*, 1997.

[7] Kobus van der Merwe. *Ariel* Open Switch Control Interface. Draft specification, October 1996.

[8] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Draft 10th December 1991. OMG Document Number 91.12.1, revision 1.1.

[9] Richard Black. Explicit Network Scheduling. Technical Report 361, University of Cambridge Computer Laboratory, December 1994. Ph.D. Dissertation.

[10] C. Liu and J. Layland. Scheduling algorithms for multprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, February 1973.