Scheduling Policies for Protocol Stacks for User Applications Requirements

K . Vijayananda, G. Berthet

Swiss Federal Institute of Technology
EPFL-LIT, IN-Ecublens
CH-1015, Lausanne, Switzerland

D. Sidou

Institut Eurécom
Sophia Antipolis
06560 Valbonne, France

Abstract

This paper presents some of the scheduling issues in a communication protocol stack and discuss it with respect to the requirements of the user applications. We present the architecture of LITMAP [1] *and discusses its salient features from the performance point of view. In particular, the flexible scheduling facility provided by LITMAP is presented in detail. Two scheduling policies are considered : First-Come First-Served (FCFS) discipline and Fair Queuing (FQ) discipline. Experimental results show the expected benefits of the FQ discipline. Fair queuing policy provides relatively small delay for short communications and stable overall performance at high loads.*

1 Introduction

The design and implementation of a communication protocol stack play an significant role in the performance of the user applications. It is essential that the services provided by the protocol stack meet the performance requirements of the user applications. Moreover, if these requirements cannot be satisfied, the protocol stack should be able to meet the user requirements in a fair manner. The following can be considered as a fair service provided by a server:

- Low load: The server satisfies the requests of all the clients.
- High loads: The server enforces a fair policy so that all the clients get an equal share of the server.

The main requirement of user applications on top of a protocol stack is the response time. In general, response time can be defined as the time elapsed between a request and the receipt of the response. In a Client-Server model, response time is defined as the time between sending a request to the server and the receipt of a response by the client. Considering the protocol stack as a server that provides services to user applications, in this paper we address the issues related to the service time of the protocol stack. The data transfer delays across the network and remote processing time are not taken into account in this study. The protocol stack provides services to many user applications simultaneously. The scheduling policy plays a major role in determining the time taken by the protocol stack to complete the service requests from the different user applications (*response time* of the protocol stack).

The main goal of this paper is to study the effect of scheduling policies on the response time of the protocol stack. In this paper, we present two different scheduling policies: First Come First Served (FCFS) and Fair Queuing (FQ) scheduling policies. These scheduling policies are analyzed under different load conditions. Experiments are performed using the scheduler of the *LITMAP* [3] to study the behaviour of the proposed scheduling policies.

This paper is organized as follows. Section 2 presents different implementation techniques for protocol stacks. The architecture of LITMAP is also discussed in section 3 with the emphasis on its scheduling abilities. In section 3, we discuss some of the salient features of LITMAP, its advantages and disadvantages. Section 4 discusses the two scheduling policies that are used in LITMAP. Section 5 describes the experimental setup and methodology. The results of the experiment are analyzed in section 6.

2 The LITMAP Protocol Stack

In this section, we present some of the methods of implementing protocol stacks. We then present the implementation details of LITMAP stack and some of its salient features, advantages, and disadvantages.

2.1 Protocol Stack Integration

There are many ways to integrate a protocol stack with a multi-user/multi-tasking environment. A survey is presented in [1]. On one hand, there are monolithic organizations, where the entire protocol stack supported by the system is implemented within a single address space(kernel address space) [5]. This is to provide robustness and for security. On the other hand, non-monolithic implementations (like in LITMAP) have a dedicated server containing the code for different parts of the protocol stacks. At low level, the servers interact with network device drivers through the kernel system calls (TRAPS). At high level, user applications communicate with the protocol stack through Inter-Process Communication mechanisms

[1] LITMAP is the implemetation of the OSI Stack at LIT-EPFL.

(IPCs) like shared memories, message queues and semaphores. There are several factors in favor of non-monolithic protocol implementations and especially for solutions outside of the kernel address space. The first reason is the ease of prototyping, debugging and maintenance. Second, new communication modes such as graphics and video with bulk transfer and real-time requirements are emerging today. These communication modes require special purpose protocols and all these protocols must coexist in a single protocol stack. Also, exploiting application specific knowledge for fine tuning and optimization is easy in a non-monolithic implementation.

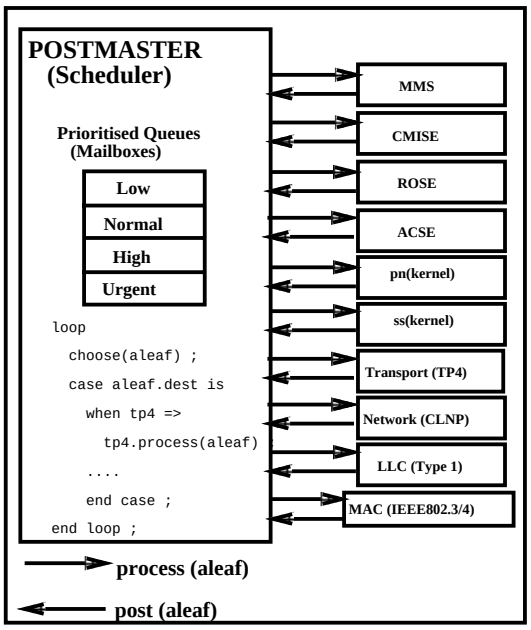

Figure 1: Communication Architecture of LITMAP.

2.2 The LITMAP Architecture

As shown in Figure 1, the communication architecture of LITMAP [3] is based on a scheduler (*postmaster*) managing prioritized queues (*mailboxes*). The postmaster supports four levels of priority in the following order with *Low* having the lowest priority: *Low, Normal, High and Urgent*. The mailboxes are used to store service requests from users. Entities are service providers and users make use of the services provided by the Entities. Examples of entities and users are Transport, Session, Network layer,etc. and they are represented as a pair (entity-id, process). The entity-id enables the postmaster to uniquely identify the entity. The process represents the protocol machine corresponding to the services provided by the entity.

Users request the services of an entity by posting a service request (aleaf in Figure 1) to one of the mailboxes in the postmaster. The words service request and task mean the same thing in this paper. The service request contains separate fields for entity-id of the requested entity, priority of the request, service request identifier, parameters for the service request and user data. When the postmaster receives a service request in one of the mailboxes, it calls the process corresponding to the entity, that is specified in the destination field of *aleaf*. In this architecture, inter-layer communication is asynchronous and is achieved through the postmaster. There is no direct interaction between entities. All communication between entities is achieved through the postmaster.

3 Features of LITMAP architecture

3.1 Scheduler

The postmaster is a non-preemptive priority-based scheduler. Currently, four levels of priority are available on the LITMAP. Separate queues are maintained for each level of priority. Service requests are placed in the queue corresponding to the priority level indicated by the *priority* field of the *aleaf*. The *choose* function selects a service request from the non-empty queue with the highest priority. A service request which is being processed by the scheduler, cannot be preempted by another request with higher priority. Interrupts due to Timer and Network activity (e.g. receipt of a packet) can interrupt the current service request.

3.2 Association

Associations are referenced through *association_ids*. This structure contains the address information which consists of local and remote Service Access Points (SAPs) and the path of *connection_ids* for each connection oriented layer involved in the association. The user applications views the *association_id* as a communication end point reference that is used whenever an exchange occurs between the user application and the protocol stack (sending a request and receiving a response). For the layer entities themselves, address fields of the association enable them to identify which layer entities are above and below them.

3.3 Constraints

The postmaster imposes a few constraints on the scheduling policy. Constraints imposed by the postmaster scheduling algorithm are : Priority levels are respected, i.e. at each cycle, an aleaf with higher priority is selected first. Another constraint is the order in which the service requests from the same association are scheduled. The postmaster cannot invert the order in which two consecutive service requests from a given association are processed. Thus, strict FCFS discipline has to be observed within an association. If this constraint is violated, a user application making two consecutive service requests will have no guarantee about the order of receipt of the corresponding responses. This may result in an unacceptable behavior.

3.4 Advantages

The main advantages of this architecture are its flexibility to schedule service requests and priority handling.

Priority mechanism is supported by providing separate queue (mailbox) for each priority in the postmaster and by providing priorities to service requests from each entity. These priorities may be mapped directly to the priority handling mechanism at the MAC level. The priority mechanism can also

be used to compensate the lack of priority support in some networks (Ethernet IEEE 802.3 etc). Priority is a powerful feature of this architecture, and it can be used to convey urgent network management traffic or any out-of-band data needed for real-time applications (manufacturing applications).

The architecture of the postmaster permits to schedule the service requests based on their priority. In Figure 1 the function **choose** selects the next service request from the mailboxes. This makes it easy to have different scheduling algorithms based on the requirements of applications. This facility can be used to schedule real-time applications to meet their deadlines.

3.5 Disadvantages

The implementation of the protocol machinery inside entities can be very complex, particularly for the error handling mechanisms. In the synchronous case, when an error occurs (for e.g. non-availability of a resource), the calling entity is immediately informed and the corresponding procedure is canceled. In the asynchronous case, the error handling is much more complex. Returning an error primitive to the upper entity may not be correct because in the mean time many other events may have occurred preventing any possible recovery procedure.

4 Scheduling Policies

In this section we discuss two different scheduling algorithms: FCFS and Fair Queuing. On a given priority level, strict FCFS has to be observed for each association. However, it is not necessary to observe global FCFS discipline among all associations and for a given priority level. This enables one to think about a more elaborate scheduling discipline like the Fair Queuing.

4.1 FCFS Discipline

The FCFS discipline is used as the primary scheduling policy in the **LITMAP** . The corresponding queuing model for each priority level is a simple server as shown in Figure 2. There is a single queue for each class of customer. Inspite of being a popular scheduling policy, the simple FCFS scheduling policy often results in some class of users hogging the resources and starvation for other class of users.

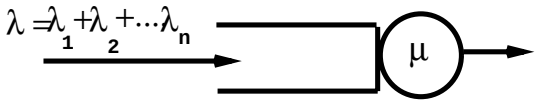

Figure 2: FCFS Discipline.

4.2 Fair Queuing Discipline

Fair Queuing (FQ) is an attempt to provide a service discipline emulating Processor Sharing (PS) discipline [4, 2]. FQ provides an interesting paradigm for the fair sharing of a server. As shown in Figure 3, a queue corresponding to a given priority level is now split in n queues corresponding to the maximum number of associations/customers admissible into the system. The protocol stack is the single server and the n job arrival streams are the n potential associations, each joining a different first-come, first-served queue.

In theory, PS disciplines serve the associations at a rate proportional to $1/k$ when there are $k > 0$ non empty queues. Maintaining a separate queue for each association results in *firewalls*, which protects well-behaved associations from associations that might saturate the server and result in unacceptable delays. If appropriately emulated, a mechanism like FQ/PS provides relatively small queuing delays for short communications like request/response communications (for e.g. telnet sessions). It provides protection against associations that would otherwise swamp the server with lots of service requests and saturate the server. It also provides stable overall performance, especially when the load is high. This is an important feature, because when the load is high, a robust and powerful policy is necessary to ensure stable overall performance.

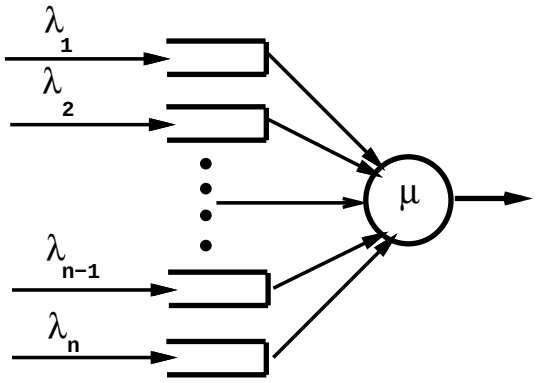

Figure 3: Fair Queuing Discipline.

4.3 Implementation of Fair Queuing in LITMAP

A simple and classical method to implement fair queuing is to use **Round Robin** (RR) scheduling. Service-requests from the non-empty queues are processed in a cyclic manner. Processing a service request consists of executing the *process* procedure corresponding to the *entity* specified in the destination field of the **A Leaf** at the head of the selected queue. Round robin scheduling is not a perfect scheme with respect to fairness especially when the associations are not homogeneous. First, variable PDU (Protocol Data Unit) lengths will rule out emulating FQ/PS by serving associations in a round robin manner. Second, at lower layers (up to transport layer), the association_id for incoming frames is not known. A common default, value is assumed, resulting in a single queue. Thus, an ill-behaved remote source can swamp the local protocol stack, and prevent the enforcement of the fair policy. Third, a malicious application process may also open several associations to convey its traffic, increasing proportionally its scheduling rate. In spite of this problems, FQ/PS emulation has been implemented using round robin scheduling, leaving the postmaster as simple as possible to limit the over-

head introduced. The cost of achieving strong fairness among associations is the increased overhead of the postmaster. However, comparing the benefits of achieving strong fairness with the increase in the overheads of the postmaster will have to be investigated in detail in a a separate study. Handling, variable PDU lengths is possible with the use of mechanisms introduced and well stated in [2]. For more details the readers can refer to [2]. Frames without *association_id*, a simple and pragmatic approach would be to distribute them, again "fairly" among all other associations. This would prevent frames belonging to other associations being blocked in the association-less queue, thus solving the main problem of an ill-behaved remote source swamping the local protocol stack.

4.4 Expected Benefits for User Applications

The expected benefits resulting from the introduction of a fair queuing discipline as the protocol stack scheduling policy are in favor of short communication associations. Suppose we have two user applications, one file transfer or download application (long communication) and another with sporadic traffic or request/response application (short communication). In the case of the FCFS discipline, all of the remaining service requests already in the postmaster queues have to be processed first before a request from a short communication application can be processed. This results in a large response time for the short communication application (which requires short response time).

In the case of the FQ discipline, the response time of the short communication application will not be affected by the large number of service requests from other long communication applications. The two application are serviced alternatively, layer by layer. Moreover, the long communication application does not suffer at all and even does not notice the small delay introduced by the insertion of the sporadic traffic. Obviously, the same goal could have been achieved by using different priority levels for each application. The problem with priority mechanisms is that they are subject to the correct setup of the administrative policy of assigning priorities to different application. Fair queuing achieves this goal implicitly in a self-organized fashion. That is the reason why it is a powerful mechanism which can be of great usefulness to user applications.

5 Experimental Setup

The experimental setup is shown in Figure 4. It is based on the Network Management protocols and applications of the LITMAP environment :

1. A constant traffic flow, based on CMIS `M_Get` service primitive is generated between pairs ($< MASH_i, ASH_i >$) of connected MAnagement and Agent SHells. This constant traffic flow is implemented as an infinite loop whose body gets an integer attribute value.

2. A sporadic traffic flow is implicitly generated thanks to a counter and its associated threshold. The counter is the `DMI::octetsSentCounter` and the threshold is the `DMI::octetsSentCounterThreshold` of the Transport layer (TP4) entity. This threshold is configured so that each time the `DMI::octetsSentCounter` counter is increased by more that a fixed amount of octets (e.g. 10 kilo bytes), a notification is signaled to the stack agent. Then the stack agent sends an actual `M_EventReport` to any management party which is registered to receive such a notification, e.g. NOSH the NOtification SHell in our case.

Note well that since we are interested only by the processing time of the stack itself, all the involved processes are run on a single host[2]. The stack itself, including the agent entity are running inside a single process. Each MAnagement SHell ($MASH_i$) and Agent SHell (ASH_i) pair are running as two different processes communicating with the stack through IPCs (i.e. shared memories). Finally, the NOtification SHell ($NOSH$) also runs as a separate process which establishes upon initialization a connection with the stack agent entity.

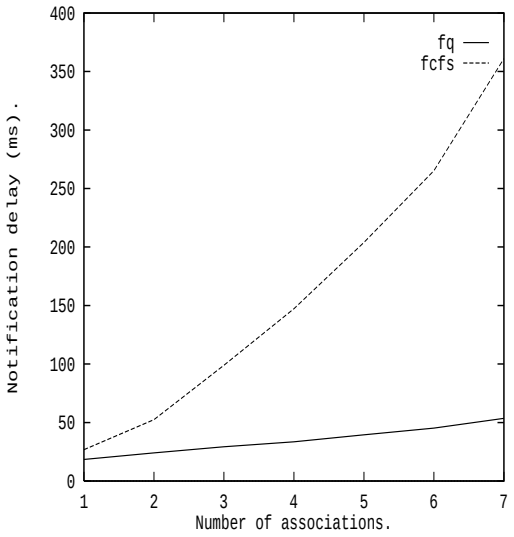

Figure 5: Average stack traversal time for sporadic traffic.

This example is quite interesting, because it is also realistic. In effect, being able to handle short communications, like event reports or alarms, in an efficient way, and above all without having to wait for non-urgent traffic to be processed is very important and is perfectly illustrated by this small experimental setup.

The results are presented in Figure 5, showing the average time for the processing inside the stack for the event report communications between the Stack Agent and the notification shell. As claimed before, the experiments confirm the fair policy enforcement by the fair queuing scheduling. With the FQ discipline, the delay increases smoothly since one more association corresponds only to one more

[2]The experiment were made on a Sun SparcStation 2 running SunOS 4.1.3 (32Mb RAM, 40 MHz).

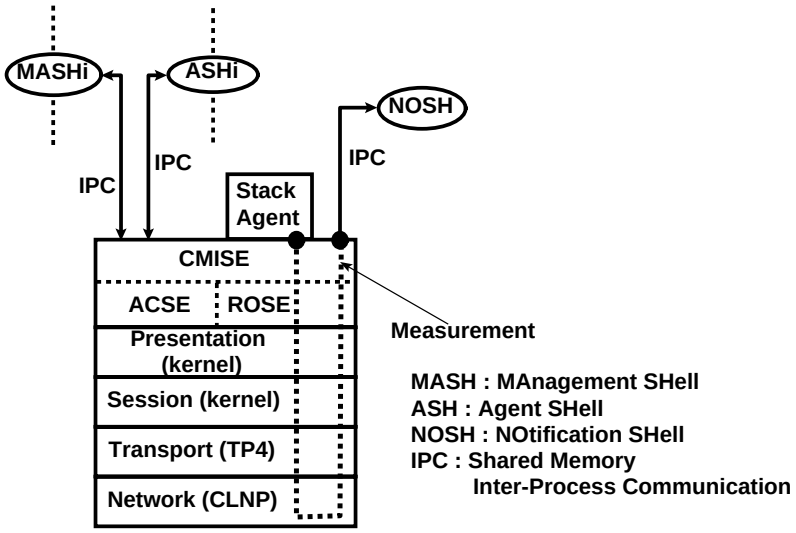

Figure 4: Experimental Setup.

service request to be processed before the alarm notification request from the short communication is processed. With the FCFS discipline, the delay increases more rapidly with each new association. In effect, each new association increases significantly the mean number of service requests to be serviced by the postmaster before the alarm notification can be serviced.

6 Conclusion

In this paper, we have discussed the issues related to scheduling in a communication protocol stack and how they are related to the requirements of the user applications. The LITMAP protocol stack which is a non-monolithic implementation, is used to study two scheduling policies. The first scheduling policy is the simple first-come, first-served (FCFS) discipline. The fair queuing (FQ) discipline is then considered and compared with FCFS. A Network Management experimental setup is used as an user application to compare the two scheduling policies. The expected benefits of using FQ are relative small delays guaranteed for short communications and stable overall performance when traffic is heavy. Moreover, FQ is a very flexible and powerful mechanism in the sense it is a self-organizing policy processing maximum user service requests at low loads, and providing only the fair share at high loads.

Our current implementation of FQ is based on a simple *round-robin* scheduling policy. This work can be extended to implement a fair queuing discipline using other mechanisms and in particular, taking into account heterogeneous associations and evaluating the corresponding overhead of the scheduler.

Acknowledgments

We are grateful to the MAP team at LIT for their support in implementing the LITMAP stack and our colleagues at LIT for their valuable comments

References

[1] Evelyn Moy Chandramohan A. Thekkath, Thu D. Nguyen and Edward D. Lazowska. Implementing network protocols at user level. In *Conference Proceedings, Communication Architectures, Protocols and Applications*, pages 13–17. ACM SIGCOMM'93, September 1993.

[2] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queuing algorithm. *Internetworking: Research and Experience*, 1:3–26, 1990.

[3] D.Sidou, K.Vijayananda, and G.Berthet. An architecture for the implementation of osi protocols: Support packages, tools and performance issues. In *Proceedings of the IEEE SICON/ICIE'93*, pages 6–11. IEEE Singapore, September 1993.

[4] Albert G. Greenberg and Neal Madras. How fair is fair queuing? *Journal of the Association for Computing Machinery (ACM)*, 39(3):568–598, July 1992.

[5] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quatterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, Inc, 1989.