Implementation Issues of an ATM Switch

Xiaoman Song

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements
For the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

September, 2001

0-612-68516-0

ABSTRACT

Implementation Issues of an ATM Switch

Xiaoman Song, M.S.
Concordia University, 2001

The ATM (Asynchronous Transfer Mode) is the switch and multiplexing chosen by CCITT for the broadband ISDN network and has been widely applied in the industrial. The objective of this project is to address the most important issues concerned with the implementation of a switch. In this report, we focus our discussion on Software Architecture and ATM Scheduling. For a network software implementation, an appropriate software architecture must be chosen. We will discuss the three software architectutes used in the network software implementation and make the justification for them. The scheduling disciplines of the switching nodes, which control the order in which cells are transmitted, determine how cells from different connections interact with each other. The scheduling discipline has the great affect on the switch's performance In this report, we will present guildience to choose a cell scheduling algorithm for an ATM switch

Table of Contents

List of Figures

ACRONYMS

AAL	ATM adaptation layer
ACSE	Association control service element
ASIC	Application Specific Integrate Circuit
ATM	Asynchronous transfer mode
CDMA/CD	Carrier sense multiple access with collision detection
CLP	Cell loss priority
CS	Convergence sublayer
EDD	Earliest-due-date
EDF	Earliest-deadline-first
FCFS	First-come-first-serve
FIFO	First-in-first-out
HEC	Header error control
HRR	Hierarchical round robin
IP	Internet protocol
ISR	Interrupt service routine
GFC	Generic flow control
LAN	Local area network
MAC	Medium access control
NNI	Network-network interface
OSI	Open systems interconnection
PDU	Protocol data unit
PTI	Payload indicator
OS	Operating system
QOS	Quality of service
RCSP	Rate –controlled static priority
SAP	Service access point
SAR	Segmentation and reassembly
SONET	Synchronous optical networks

TCP	Transmission control protocol
UNI	User-network interface
VCI	Virtual channel identifier
VPI	Virtual path identifier
WAN	Wide area network

Chapter 1 Introduction

1.1 ATM introduction

Asynchronous Transfer Mode (ATM) has been accepted universally as the transfer mode of choice for Broadband Integrated Services Digital Networks (BISDN) [BO92, MS98]. ATM can handle any kind of information, i.e. voice, data, image, text and video, in an integrated manner. ATM provides good bandwidth flexibility and can be used efficiently from desktop computers to local area and wide area networks. ATM is a connection-oriented packet switching technique in which all packets are of fixed length, namely 53 bytes (5 bytes for header and 48 bytes for information). No processing, such as error control, is done on the information field of ATM cells inside the network and is carried transparently in the network.

As indicated above, ATM has been designed to satisfy the need to carry both real and non-real-time data over a single network. To achieve this goal, ATM must incorporate the following important features:

- Packet Switching: Packet switching can transfer information between two end points asynchronously without dedicated synchronous circuits. This reduces the waste of network resources, since packets from multiple sources are queued for transmission over the same link and any unused bandwidth by one source can be used by others. In addition to increasing network utilization, packet switching can support connections with a wide range of bandwidth requirements, as it is not tied to a specific data rate. This protects packet switching from becoming outmoded.

- High speed: Packet switches should be able to operate at a very high speed to support the high bandwidth connections of the future (e.g., HDTV).

1.1.1 ATM protocol architecture

Figure 1.1 shows the protocol architecture for ATM defined by the ITU. ATM is the common layer used by all services running over ATM networks. All information at the ATM layer is transported in 53-byte fixed-size cells. The adaptation layer is service-specific; there are different adaptation protocols for different services. The adaptation layer maps application information into ATM cells and vice versa. The physical layer supports encoding of data onto physical transmission media.

In addition to its protocol layers, the ATM protocol architecture includes three separate planes: the user plane, control plane, and the management plane. The user plane supports the transmission of user information; the control plane provides connection controls; and the management plane coordinates activities among layers and manage network resources.

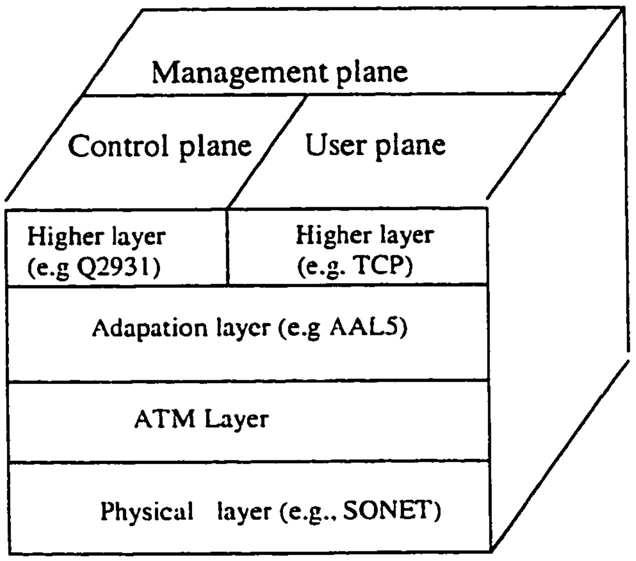


Figure 1.1 ATM protocol architecture

The ATM cell is 53 bytes in length and is capable of carrying any type of data. The first 5 bytes are used for the header. The payload portion of the cell is 48 bytes. There are two formats for the ATM header. One header is used for the UNI (User-network interface), while the other is used for the NNI (Network-network interface). The difference lies in the Generic Flow Control (GFC) parameter found in the UNI header. The GFC is not supported in the public network, nor was it intended to be.

The header is placed in front of the payload (it arrives first). There is no trailer used in ATM. Figure 1.2 shows the two header formats. The GFC values are not currently defined in the UNI header, but the intent is that the GFC could be used to provide flow control from the user to the network (and vice versa). This parameter will never be used in the public network and is overwritten by network nodes.

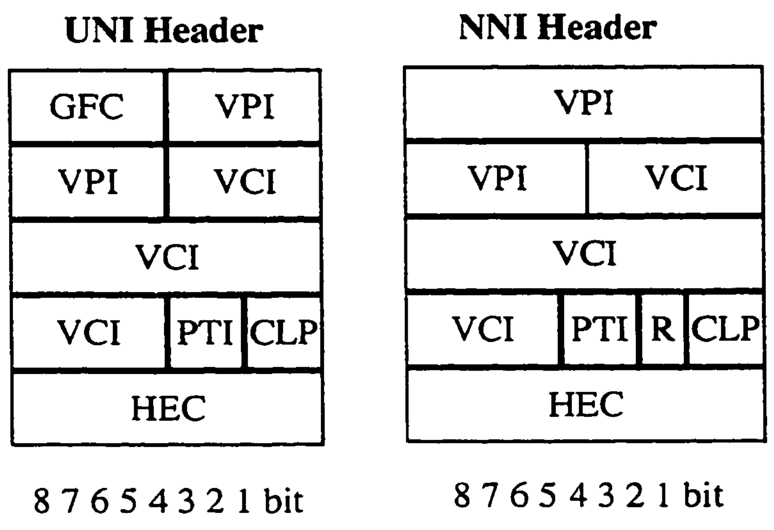


Figure 1.2 ATM Header (UNI and NNI)

The VPI (Virtual path identifier) is used to identify a group of virtual channels within the same endpoint. This is the form of addressing supported in ATM. Rather than identifying millions of unique nodes, ATM addressing identifies a connection. A virtual channel is used for a communication link. Each virtual channel is identified by the VCI (Virtual channel identifier). Following the VPI and the VCI is the Payload Indicator (PTI). This parameter indicates the type of data found in the payload portion of the cell. The payload could be signalling information, network management messages, and other forms of data. These are identified by the PTI. The PTI is followed by the Cell Loss Priority (CLP) parameter. This is used to prioritize cells. In the event of congestion or some other trouble, a node can discard cells that have a CLP value of 1 (considered low priority). If the CLP value is 0, the cell has a high priority and should only be discarded if it cannot be delivered. The last byte in the header is the Header Error Control (HEC) parameter, which is used for error checking and cell delineation. Only the header is checked for errors. The HEC works like other error checking methods, where an algorithm is run on the header and the value placed in the HEC. ATM is capable of fixing single bit errors but not multiple bit errors.

1.1.2 ATM layer Vs OSI layers

The OSI (Open systems interconnection) Model was developed in 1977. It is a seven-layer model as shown in Figure 1.3. Although OSI has standardized many of these protocols for each layer, only a few are in widespread use. The layering concept, however, has been widely adopted by every major computer and communications standards body and most proprietary implementations as well.

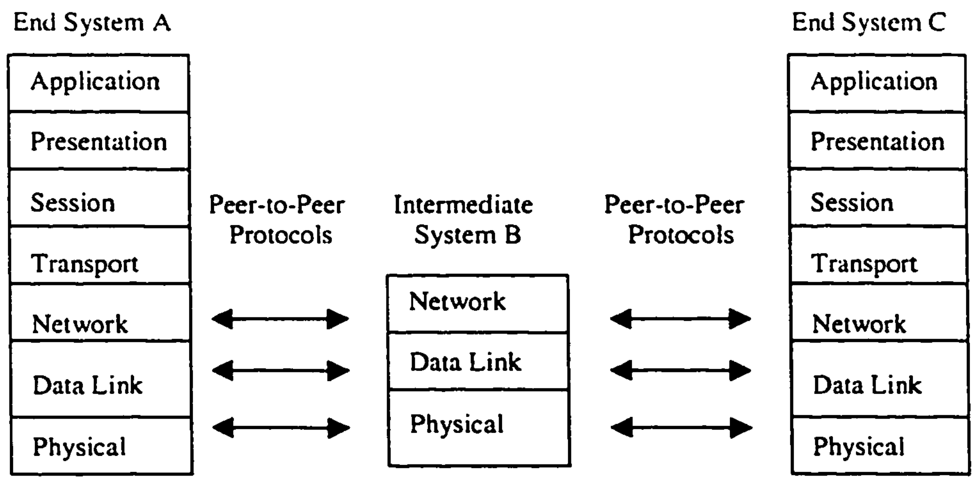


Figure 1.3 OSI Model

ATM is also divided into layers (See Figure 1.4). The ATM physical medium layer is responsible for transmission of data over the physical medium, regardless of the type of medium used. ATM was originally designed to operate over fibre optics but because of the slow deployment of fibre, it was later modified to operate over copper and coaxial facilities as well.

The ATM layer is responsible for multiplexing cells over the interface. ATM (Asynchronous Transfer Mode) must read the VPI/VCI of incoming cells, determine which link cells are to be transmitted over, and place new VPI/VCI values into the header. At endpoints, the ATM layer generates and interprets cell headers (endpoints do not route cells).

The AAL (ATM Adaption Layer) is used mostly by endpoints. It is divided into two sublayers: SAR (Segmentation and reassembly) and the Convergence Sublayer (CS). The SAR reconstructs data that has been segmented into different cells (reassembly). It is also responsible for segmenting data that cannot fit within a 48-byte payload of an ATM cell (segmentation).

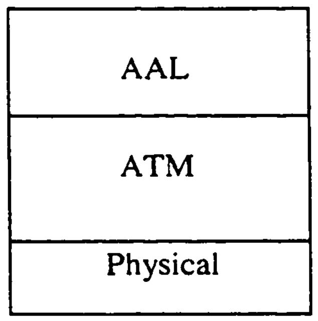


Figure 1.4 ATM layer

1.1.3 ATM application and benefits

The ATM network was designed to be used as both a LAN (Local area network) and WAN (Wide area network). Consequently, it was once envisioned that the ATM network would become a ubiquitous network that would replace most of the current networks. However, because of the large installed base of legacy LANs and associated TCP/IP infrastructure and the higher prices of ATM equipment, the current trend is to continue to use legacy LANs as much as possible and to use ATM as a high-speed backbone network to interconnect legacy LANs using TCP/IP. Therefore, IP over ATM is one of the most important applications for ATM. The following figure shows that TCP/IP networking using the ATM in the backbone approach.

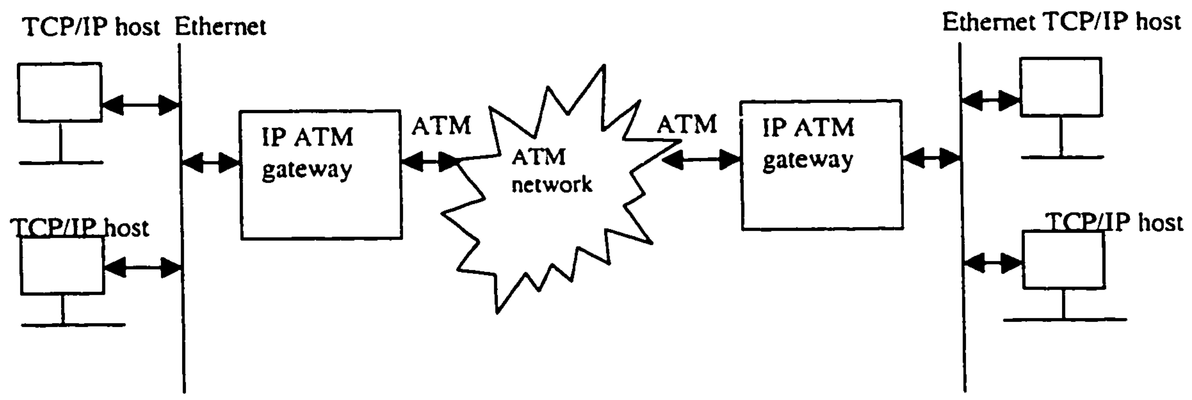


Figure 1.5 TCP/IP networking using the ATM in the backbone approach

The IP over ATM method was standardized by the IETF as a mechanism to run IP over ATM networks where IP treats ATM as a new technology and has access to all ATM functions and features including the guaranteed QoS (Quality of service).

Comparing to the traditional network, ATM has several key benefits

- One Network -- ATM will provide a single network for all traffic types-voice, data, video. ATM allows for the integration of networks improving efficiency and manageability.
- Enables new applications -- Due to its high speed and the integration of traffic types, ATM will enable the creation and expansion of new applications such as multimedia to the desktop.

- Compatibility -- Bccause ATM is not based on a specific type of physical transport, it is compatible with currently deployed physical networks. ATM can be transported over twisted pair, coax and fiber optics.
- Incremental Migration - Efforts within the standards organizations and the ATM Forum continue to assure that embedded networks will be able to gain the benefits of ATM incrementally-upgrading portions of the network based on new application requirements and business needs.
- Simplified Network Management - ATM is evolving into a standard technology for local, campus/backbone and public and private wide area services. This uniformity is intended to simplify network management by using the same technology for all levels of the network.
- Long Architectural Lifetime - The information systems and telecommunications industries are focusing and standardizing on ATM. ATM has been designed from the outset to be scalable and flexible.

1.2 Motivation of this project

While most of today's public packet-switched computer networks support only the best-effort service, a growing number of applications demand real-time communication over high speed integrated-service networks, such as ATM networks, that allow the user to transport information with the pcrformance guarantees on delay, delay jitter, throughput and loss rate, etc. For the most demanding applications, the networks must offer a service which can provide deterministic guarantees for the maximum delay and delay jitter. In order to achieve the deterministic guarantees over the packet-switched network, extensive studies [MZ92, SZ93, KHBG91] have been done in the academic domain. Several important issues which have a great influence on implementing a high performance ATM switch have been addressed . These are discussed in the next two sections.

1.2.1 Network Software Architecture

From the ATM switch implementor's point of view, ATM can be seen as a three-layer architecture (Application layer, ATM layer and physical layer). The application layer includes the SSCOP (Service Specific Connection-Oriented Protocol), SAR and ATM adaptation software. The ATM layer mainly includes the traffic management, connection and disconnection and ATM cell construction. The physical layer includes the device drivers such as T1 or T3 drivers (In this report, we don't discuss the hardware-related issue) as shown in the following Figure.

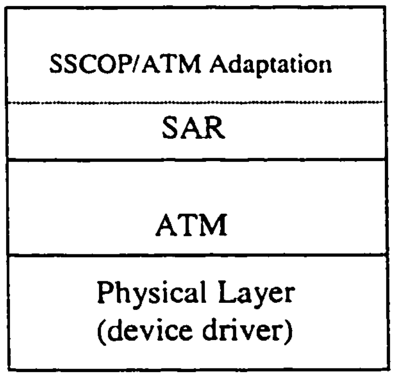


Figure 1.6 ATM layer from the implementor's point of view

In the ATM market, most of the functionalities of the ATM layer are also implemented by the ASICs (Application Specific Integrate Circuit). The implementation choice for the layer-protocol software structure and the inter-layer communication will have a great influence on the ATM performance.

There are essentially three approaches to translate the layer-protocol network architecture into software:

- Process-based approach: the layers are implemented as processes provided by the supporting operating system or kernel such as VxWorks kernel.
- Procedure-based approach: layers are implemented as a set of procedures which may be called by other layers (above, below) [CL85] or by a master.
- Hybrid approach: this approach combine the process-based implementation and procedure-based implementation. In this report, we will describe each approach and present its features. We are particularly interested in the inter-layer communication mechanism in these three approaches because it has an important influence on the performance of the system.

1.2.2 Cell Scheduling and Queuing

In an ATM network, cells from different connections interact with each other at each switching node. These interactions may adversely affect the network performance without appropriate control. The following figure shows that the high-priority cells (black box) may be delayed by the low-priority cells (gray box) if no appropriate control is done in the switch.

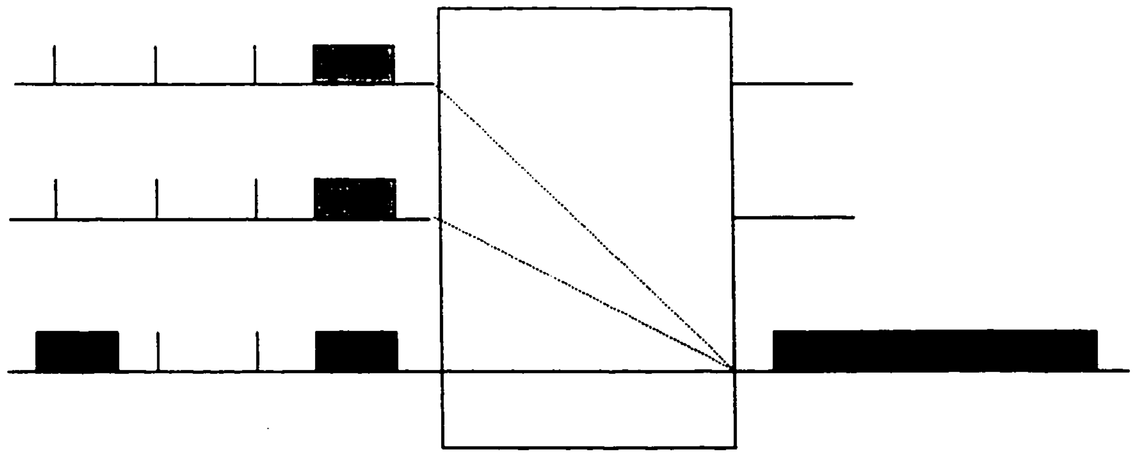


Figure 1.7 High-priority cells are delayed by low-priority cells

The scheduling disciplines of the switching nodes, which control the order in which cells are transmitted, determine how cells from different connections interact with each other. The scheduling discipline at a switching node affects three nearly independent factors:

- bandwidth (which cell gets transmitted);
- promptness (when does a cell get transmitted);
- buffer space (which cells are discard).

In this report, we will present previous studies of cell scheduling algorithm and make the comparison between them. We also give the guidelines to choose a cell scheduling algorithm for an ATM switch.

Chapter 2 Procedure-based Architecture

2.1. Introduction

The objective of this chapter and the following chapters (chapters 3, 4 and 5) is to discuss the most important issues concerned with the implementation of a high performance ATM communication stack as shown in Figure 1.6. For a network implementation, an appropriate software architecture must be chosen. Three software architectures – procedure-based [CL85], process-based [BU84, SI84] and hybrid architectures – are presented and justified. The first two approaches have been studied intensively in the academic community. The last approach has been used in some network implementations, but, to my knowledge, there has been no published study of this approach. In some cases, hybrid architecture may be the best choice for the network implementation. In this report we make a detailed study of these three approaches and show how they can be used to implement the ATM communication stack.

When implementing a network protocol, we need to have a methodology for the program structure, a methodology suitable for operating system, especially for programs dealing with communications and networks. The methodology described in this report is relevant to programs which have been modularized according to the principle of layering. Traditionally, a layer is thought of as providing services to the layer above, or the client layer (layer N+1 in Figure 2.2) The client uses some mechanism for invoking the layer (layer N in Figure 2.2), for example a subroutine call. The layer N performs the service for the client (layer N+1) and then returns. In other words, service invocation occurs from the top down. However, the natural flow of control is not always downward. In a network driven environment, for example, most of the actions are involved, not by the client from above, but by the network from below. The natural flow of control is thus upward, not downward. In summary, the natural flow of control in a network environment should be upward for the packet reception, and downward for the transmission as shown in the following figure.

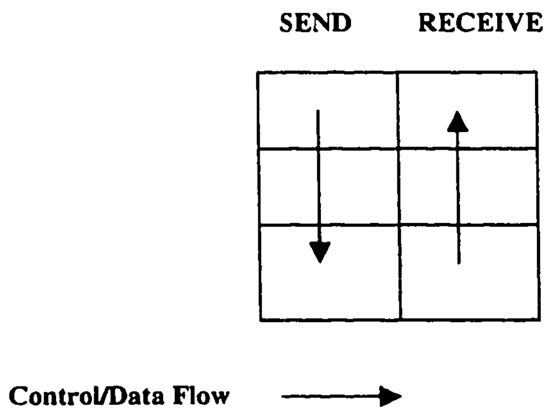


Figure 2.1 Control/Data Flow in a Network

For a layered network protocol, the data flow for the transmission is always from the top to the bottom, while the data flow for the reception is always from the bottom to the top. In what follows, we first make a brief discussion of layering and its motivation. And then we present the three approaches which can be used to implement a layered network protocol, such as ATM communication protocol.

2.2. Layering Protocol

In the introductory chapter, we presented the OSI seven layer model (see Figure1.3) and the ATM layer model (see Figure 1.1). The layers are represented starting from the bottom as the first layer, which has a physical interface to the adjacent node, to the topmost seventh layer, which usually resides on the user end device (workstation) or host that interacts with or contains the user applications. Each layer represents one or more protocols that define the functional operation of communications between user and network elements. Although OSI has standardized many of these protocols, only a few are in widespread use. The layering concept, however, has been widely adopted by every major computer and communications standards body and most proprietary implementations as well. Figure 2.2 illustrates the basis elements common to every layer of the OSI model [CY78, II]. This is the portion that has become widely used to categorize computer and communications protocols according to characteristics contained in this generic model. Often, the correspondence is not exact or one-to-one; for example, ATM is often described as embodying characteristics of both the data link and network layers.

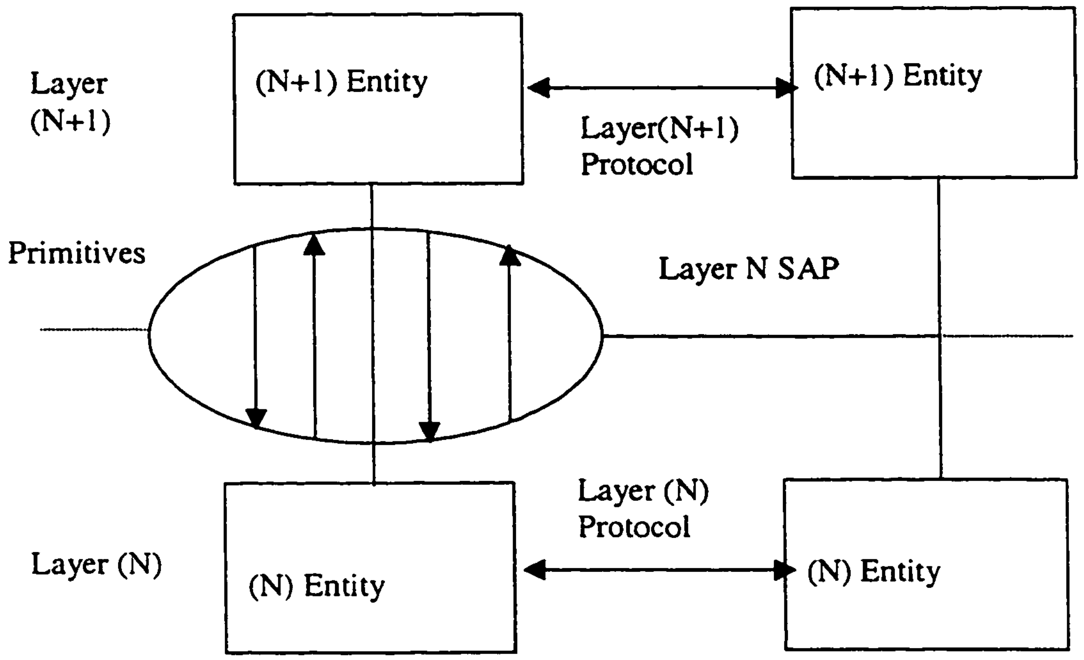


Figure 2.2 Illustration of Layered Protocol Model

Referring to Figure 2.2, a layer N+1 entity communicates with a peer layer N+1 entity by way of a service supported at layer N through a Service Access Point (SAP). The layer (N) SAP provides the primitives between layer (N) and (N+1) of request, indicate, confirm, and response. Parameters are associated with each primitive. Protocol Data Units (PDUs) are passed down from layer (N+1) to layer (N) using the request primitive, whiles PDUs from layer (N) are passed up from layer (N) to the layer (N+1) using the indicate primitive. Control and error information utilize the confirm and response primitives. In the following we give an example for primitive between layer N+1 and layer N. The primitives are defined in the layer N and are called by the layer N+1

function **sendRequestToLayerN** (Param, ...)
function **sendResponseToLayerN** (Param, ...)
function **receiveIndFromLayerN** (Param, ...)
function **receiveConfFromLayerN** (Param, ...)

The layer N+1 calls the primitive **sendRequestToLayerN** or **sendResponseToLayerN** to send a request or a response to the layer N, and calls the primitive **receiveIndFromLayerN** or **receiveConfFromLayerN** to receive an indication or a confirmation.

Although the OSI has defined each layer and the primitives between layer (N+1) and layer N in detail, however, the following two issues are left to the implementor

(1) how is each layer implemented? As a task or as a set of functions ?

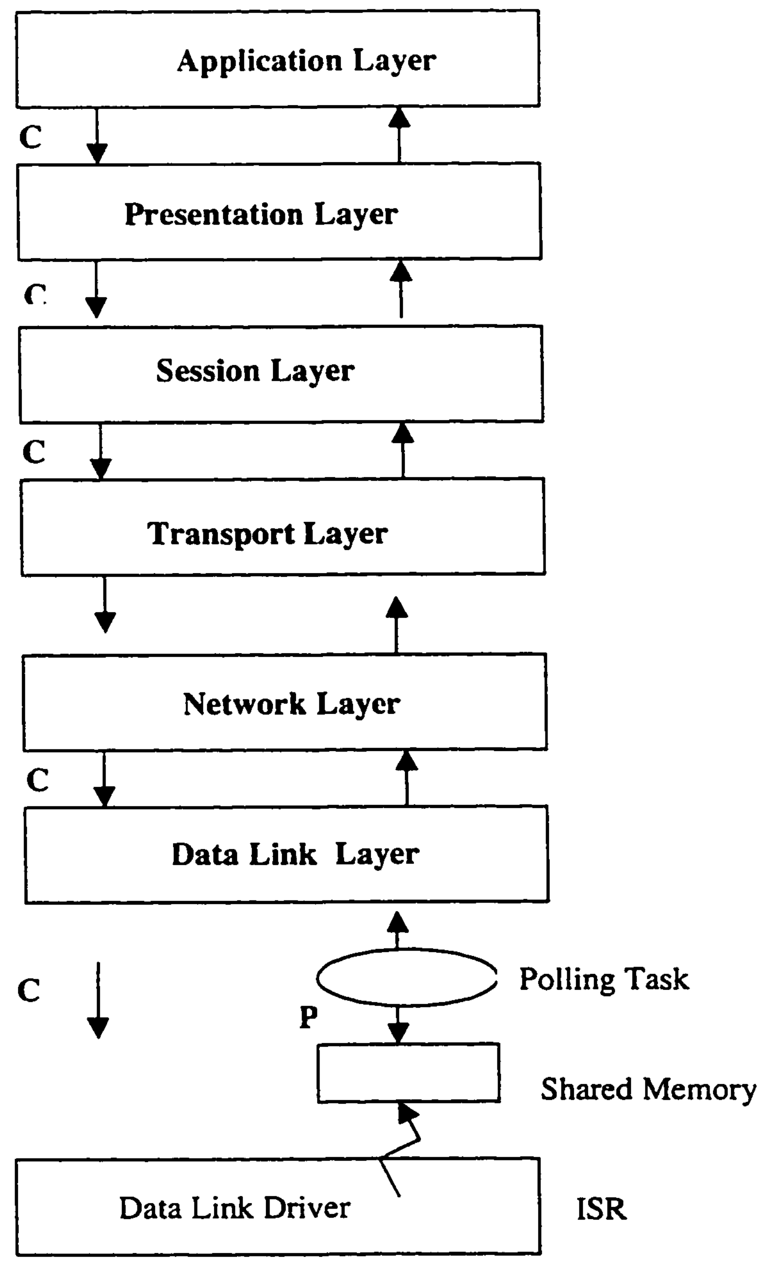
(2) how does the layer N+1 communicate with the layer N? By message queue or function call?

In the following sections we will present the three approaches.

2.3 Procedure-based Approach

As indicated above, a specification in layered form does not by itself commit the implementor to any particular approach to modularity and interface design. The *process* is the fundamental structuring component provided by the most systems. It is natural to try to map the basic module of the specification to the basic component of the system; this maps layer to process as described in the next section. However, this mapping may be substantially inefficient. The implementor smart enough to avoid this trap then discovers that neither the layered specification nor the operating system facilities really give any implementation guidance at all, forcing the implementor to design the program structure from scratch.

In the procedure-based approach, a layer is implemented as a set of subroutines (procedures or functions). The invocation across layer boundaries occurs by subroutine call, and a layer is organized as subroutines which are called by the layer above or the layer below as shown in Figure 2.3.



C : Procedure Call
I : Interruption
P : Polling

Figure 2.3 Procedure-based Architecture

In a procedure-based implementation, the inter-layer communication is done by procedure call. An exception is the communication between the lowest layer and the layer above it. In general, the data receive part of the lowest layer (Data Link Layer in Figure 2.3) in a communication system is implemented as an ISR (Interrupt Service Routine), while the transmission part is implemented as a set of procedures or functions. So, the communications between ISR and procedures must be performed.

When the user issues a request, it calls (down call) a procedure in the layer N. Then the layer N calls (down call) a procedure in the layer N-1. This procedure continues until the corresponding PDU (Protocol data unit) is inserted into the send-queue in the Data Link Layer.

When a frame is received, an interrupt is generated and processed by the entity of Data Link Layer. This entity checks various information given in the frame header and then puts the frame in a shared memory or mailbox. The Polling task can receive the frames in the following two ways:

- Polling

The polling task enquires if the mailbox is empty or not. If the mailbox is not empty, it indicates that the ISR has put a received frame in it. The polling task will take this frame from the mailbox.

- Semaphore

The polling task waits on a semaphore. When a frame arrives, the ISR first puts the received frame into the shared memory and then wakes-up the polling task by doing a "given" operation (i.e., release the polling task which is waiting on the semaphore) on the semaphore as shown in Figure 2.4

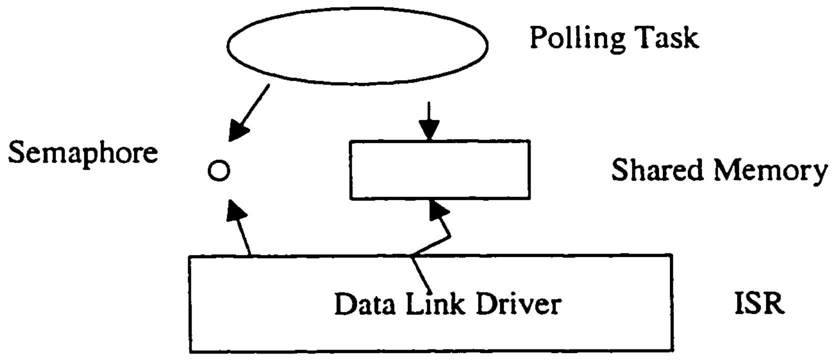


Figure 2.4 Asynchronous Communication Mechanism

2.3.1 ATM implementation based on procedure-based approach

In the following we show how the procedure-based approach is used to implement a three layed-ATM switch. Figure x illustrates the ATM implementation.

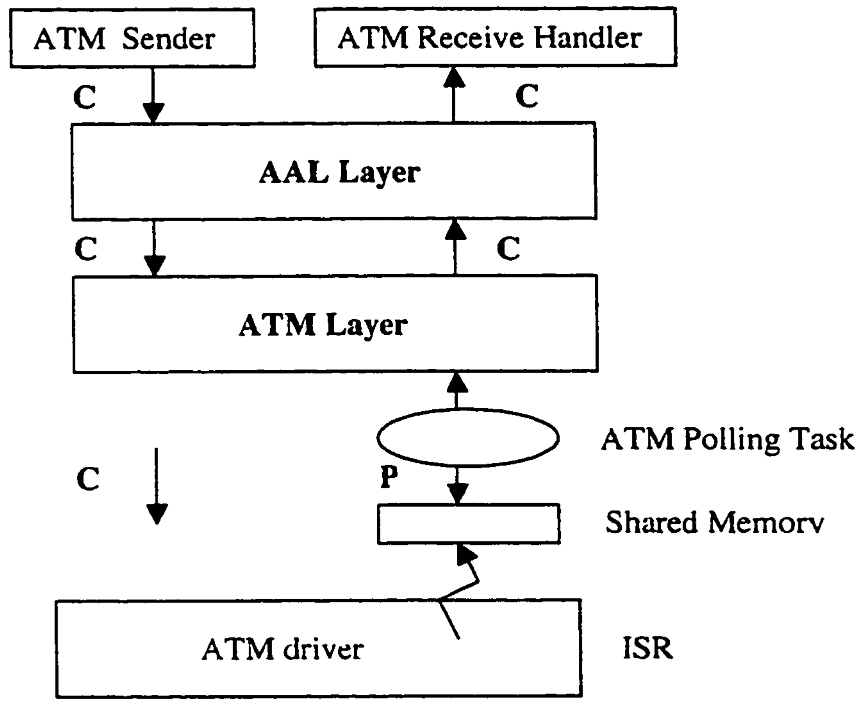


Figure 2.5 Procedure-based ATM Architecture

First, we focus our discussion on the receiving direction and show how the upcalls are executed when an ATM cell is received. Then we study on the sending direction.

2.3.1.1 ATM Receive

For the ATM receive, the main functionalities in each layer can be described as follow:

- Bottom layer (ATM driver)

This layer is responsible for receiving a ATM cell and putting it into the shared memory.

- ATM layer

The polling task periodically checks if an ATM cell has arrived or not. If it has, the polling task gets the cell from the shared memory and then upcalls the subroutine ATM_receive (see Figure 2.6c) to handle the received ATM cell, which would in turn invoke the AAL subroutine AAL_receive (see Figure 2.6b) to reassemble the ATM cells into AAL message.

- AAL layer

The AAL subroutine AAL_receive assembles the received ATM cells into AAL messages and then upcalls the subroutine ATM_app_receive_handle (see Figure 2.6a) to display the received AAL messages.

- Application

The application includes the two subroutines for the receive: `ATM_Initialize` and `ATM_app_receive_handle` (see Figure 2.6a). The former performs the connection initiation code, which involves the downcall sequence. The latter ATM_app_receive_handle is used to display the received AAL messages.

Figure 2.6 illustrates a skeleton implementation of the ATM protocol described above for the initialization and incoming cell processing. This figure includes the connection initiation code, which involves the downcall sequence, and incoming cell processing which, fairly naturally, involves an upcall. There are actually the three upcalls illustrated as part of cell receipt. When the polling task find a cell in the shared memory, it executes the subroutine ATM_receive, which in turn upcalls the AAL layer (the subroutine AAL_receive), which in turn upcalls the subroutine the application layer (ATM_app_receive_handler).

```
void ATM_Initialize()
{
   /* local_port is a received port */
   /* pass the application handler --  to AAL layer */
   local_port = AAL_open(ATM_app_receive_handler);

   /* create the polling task */
   fork(ATM_polling_task);
}


void ATM_app_receive_handler(aalMessage)
{
   /* display the received the AAL message */
   for (i=0; i < lnegth_of_cell; i++)
   {
     print each aalMessage[i]
   }
}
```

Figure 2.6a. The application layer

```
Port_Type AAL_open(receive_handler)
{
   local_port = ATM_open(AAL_receive)
   /* store the up-call subroutine for AAL layer */
   /* i.e., ATM_app_receive_handler */
   AAL_handler_table[local_port] = receive_handler;
   return local_port
}




/* AAL_receive is called by the ATM_receive. */
/* The port is from ATM_receive and is used  */
/* to get the corresponding handler */
AAL_receive(atmCell, port)
{
   /* get up-call handler, i.e. ATM_app_receive_handler */
   handler = AAL_handler_table[port];

   assemble the atm cells into aalMsg

   if it is a last cell of a message

      handler(aalMsg);
}
```

Figure 2.6b AAL layer

```
/* assign a local port and store the corresponding */
/* up-call handler */
Port_Type ATM_open(receive_handler)
{
   local_port = get_new_port();
   /* store the up-call subroutine for ATM layer */
   /* i.e., AAL_receive */
   ATM_handler_table[local_port] = receive_handler;
   return local_port
}


/* ATM_receive is called by polling task. */
/* The polling task gets the ATM cell and */
/* gets the port based on the ATM header - VPI/VCI */
/* The port is used to get the corresponding handler */
ATM_receive(port, atmCell)
{
  /* get the up-call subroutine, i.e. AAL_receive */
  handler = ATM_handler_table[port];
  handler(atmCell, port)
}


/* It is a ATM polling task */
void ATM_polling_task()
{
   /* enter an infinitive loop */
   while (1)
   {
     /* check if there is an ATM cell in shared memory */
     /* it is a block call */
     get_cell(atmCell);

     validate the received ATM cell

     /* get the port based on ATM header */
     port = get_port_from_cell(atmCell);

     /* upcall the ATM layer to handle the */
     /* received ATM cell */
     ATM_receive(port, atmCell);

   }
}
```

Figure 2.6c ATM layer

```
void ATM_driver()
{
    get the atm cell from device

    /* store the received cell into the shared memory */
    put_cell(atmCell);
}
```

Figure 2.6d ATM driver

Figure 2.7 illustrates the control relationship which exist between the various modules cell defined in Figure 2.6 for receiving an ATM. The figure indicates the upcalls and downcalls between layers with arrows, and the interrupt service routine.

This example illustrates that the upward calls are generally made using procedure variables (parameters). Use of a procedure variable is not a defining characteristic of an upcall, but it is very common. In general, layers are constructed to serve a community of higher layers which may be unknown at program definition time. So, the layer N cannot upcall the layer N+1 subroutines until the layer N+1 has first downcalled, perhaps as part of initialization, with the entry point to be upcalled later. Thus, the upcall methodology requires a language and system with suitable mechanism for procedure variables.

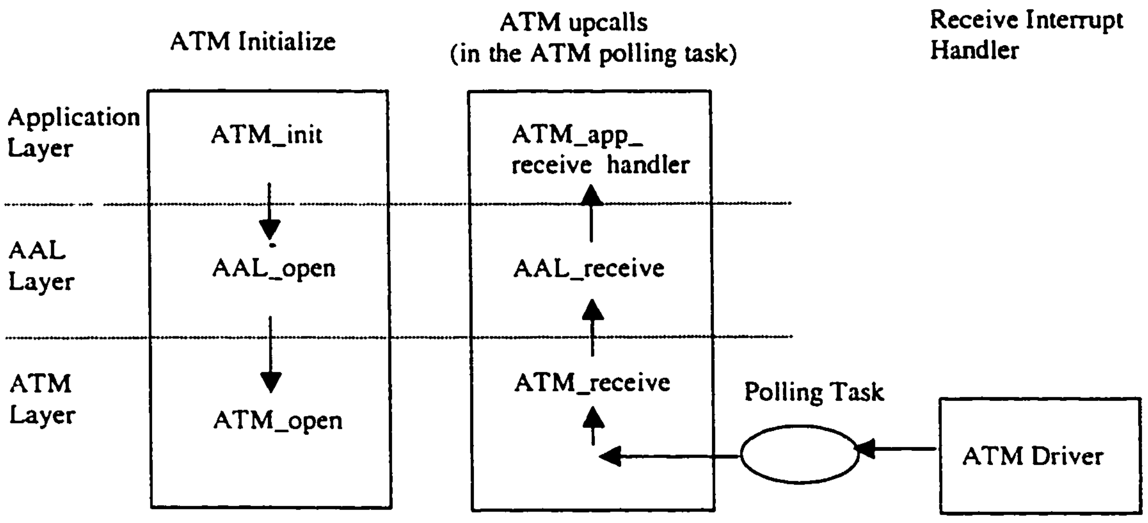


Figure 2.7 Control Flow for Receiver

2.3.1.2 ATM Send

The processing of received cells seems to fit rather naturally into the upcall philosophy. A cell is received, and the processing of that cell must necessarily proceed from the lower layers to the higher, however that flow is achieved. On the other hand, it might seem that the sending of a packet would more naturally proceed from above. The application, having some data to send, would invoke an AAL layer to format a packet, which would in turn invoke an ATM layer to segment the packet into the ATM cells, which in would in turn invoke the an ATM driver to sending the cells. Figurc 2.8 illustrates a skeleton implementation for the ATM protocol for sending the ATM cells.

```
/* This function is called by a sending task */
void ATM_app_send(aalMsg)
{
  validate aal message

  /* send message to AAL layer */
  AAL_send(aalMsg)
}
```

Figure 2.8a: The application layer

```
void AAL_send(aalMsg)
{

  segment an aalMsg into ATM cells

  /* sending ATM cells */
  ATM_send(atmCell)
}
```

Figure 2.8b: AAL layer

```
void ATM_send(atmCell)
{

  validate ATM header
  /* send out ATM cell */
  ATM_driver(atmCell);
}
```

Figure 2.8c: ATM layer

```
void ATM_driver(atmCell)
{
   /* send out cell into the device */
   for each bit in ATM cell do send bit
}
```

Figure 2.8d: ATM driver

In the following we give the description of each layer shown in the above.

- Bottom layer (ATM driver)

This layer is responsible for sending a ATM cell into the network.

- ATM layer

This layer downcalls the ATM_driver to send out an ATM cell. Before sending an ATM cell, it will validate the ATM header.

- AAL layer

The AAL_receive segments the AAL message into ATM cells and appends the ATM headers into each ATM cells, then it downcalls ATM_send to send out the ATM cells.

- Application

ATM_app_send subroutine is used to send out an AAL message

Figure 2.9 illustrate the control relationship which exist between the various modules defined in Figure 2.8 for sending an ATM cell

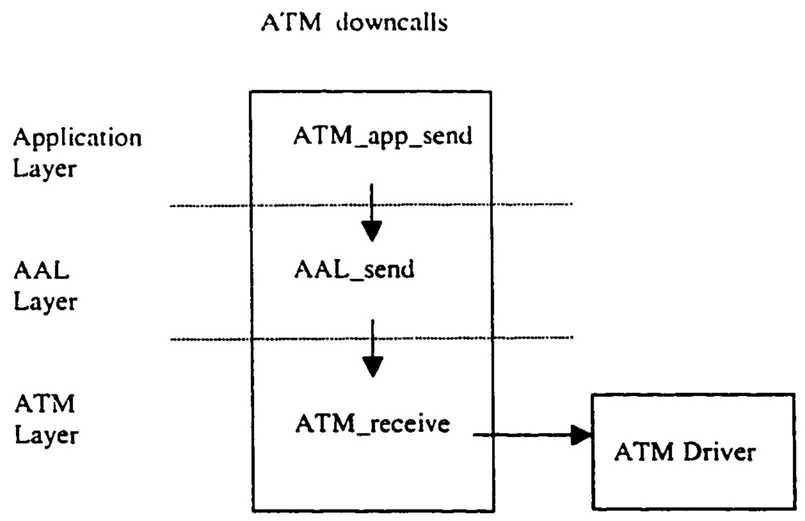


Figure 2.9 Control Flow for Send

Here, we describe the ATM implementation using the procedure-based approach, especially upcall methodology. In the following we study the two variants for the procedure-based ATM architecture.

2.3.2 The ATM architectures without the polling task

In the previous section, we presented a procedure-based ATM implementation. In the ATM layer, a polling task is used to check if an ATM cell has arrived or not. So, the underlying operating system must provide a multi-tasking environment. However, in some operating systems such as PC DOS, no multi-tasking mechanism is provided. In this section we study the ATM implementation without the polling task.

2.3.2.1 Upcall Approach

The first approach is: when an ATM cell arrives, an interrupt will be generated. The ISR (Interrupt Service Routine) gets the ATM cell and then upcalls the ATM layer function to handle the incoming cell, which in turn upcalls the AAL layer function, which in turn upcalls the application function. So, all of the incoming ATM cell processing is in the

ISR context. No other tasks are needed to handle the incoming cells. The ATM implementations is shown in Figure 2.10.

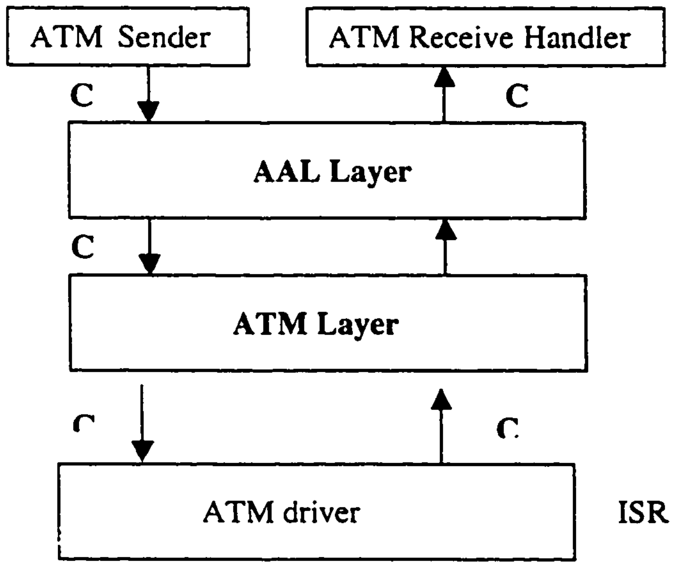


Figure 2.10 ATM Architecture without Polling Task

In the following we illustrates a skeleton implementation for the ATM protocol for sending/receiving the ATM cells.

(1) skeleton implementation for receiving the ATM cell

```
void ATM_Initialize()
{
   /* local_port is a received port */
   /* pass the application handler --  to AAL layer */
   local_port = AAL_open(ATM_app_receive_handler);

}



void ATM_app_receive_handler(aalMessage)
{
   /* display the received the AAL message */
   for (i=0; i < lnegth_of_cell; i++)
   {
      print each aalMessage[i]
   }
}
```

Figure 2.11a. The application layer

```
Port_Type AAL_open(receive_handler)
{
   local_port = ATM_open(AAL_receive)
   /* store the up-call subroutine for AAL layer */
   /* i.e., ATM_app_receive_handler */
   AAL_handler_table[local_port] = receive_handler;
   return local_port
}




/* AAL_receive is called by the ATM_receive. */
/* The port is from ATM_receive and is used  */
/* to get the corresponding handler */
AAL_receive(atmCell, port)
{
   /* get up-call handler, i.e. ATM_app_receive_handler */
   handler = AAL_handler_table[port];

   assemble the atm cells into aalMsg

   if it is a last cell of a message

      handler(aalMsg);
}
```

Figure 2.11b AAL layer

```
/* assign a local port and store the corresponding */
/* up-call handler */
Port_Type ATM_open(receive_handler)
{
   local_port = get_new_port();
   /* store the up-call subroutine for ATM layer */
   /* i.e., AAL_receive */
   ATM_handler_table[local_port] = receive_handler;
   return local_port
}


/* ATM_receive is called by ISR routine. */
/* The ISR gets the ATM cell and */
/* gets the port based on the ATM header - VPI/VCI */
/* The port is used to get the corresponding handler */
ATM_receive(port, atmCell)
{
  /* get the up-call subroutine, i.e. AAL_receive */
  handler = ATM_handler_table[port];
  handler(atmCell, port)
}
```

Figure 2.11c ATM layer

```
/* It is an ISR */
void ATM_driver()
{

     get the atm cell from device

     validate the received ATM cell

     /* get the port based on ATM header */
     port = get_port_from_cell(atmCell);

     /* upcall the ATM layer to handle the */
     /* received ATM cell */
     ATM_receive(port, atmCell);
}
```

Figure 2.11d ATM driver

(2) skeleton implementation for sending the ATM cell

```
/* This function is called by a sending task */
void ATM_sender(aalMsg)
{
  validate aal message

  /* send message to AAL layer */
  AAL_send(aalMsg)
}
```

Figure 2.12a: The application layer

```
void AAL_send(aalMsg)
{

  segment an aalMsg into ATM cells

  /* sending ATM cells */
  ATM_send(atmCell)
}
```

Figure 2.12b: AAL layer

```
void ATM_send(atmCell)
{

  validate ATM header
  /* send out ATM cell */
  ATM_driver(atmCell);
}
```

Figure 2.12c: ATM layer

```
void ATM_driver(atmCell)
{
   /* send out cell into the device */
   for each bit in ATM cell do send bit
}
```

Figure 2.12d: ATM driver

Figure 2.13 illustrates the control relationship which exist between the various modules cell defined in Figure 2.12 for sending/receiving an ATM cell. The figure indicates the upcalls and downcalls between layers with arrows, and the interrupt service routine.

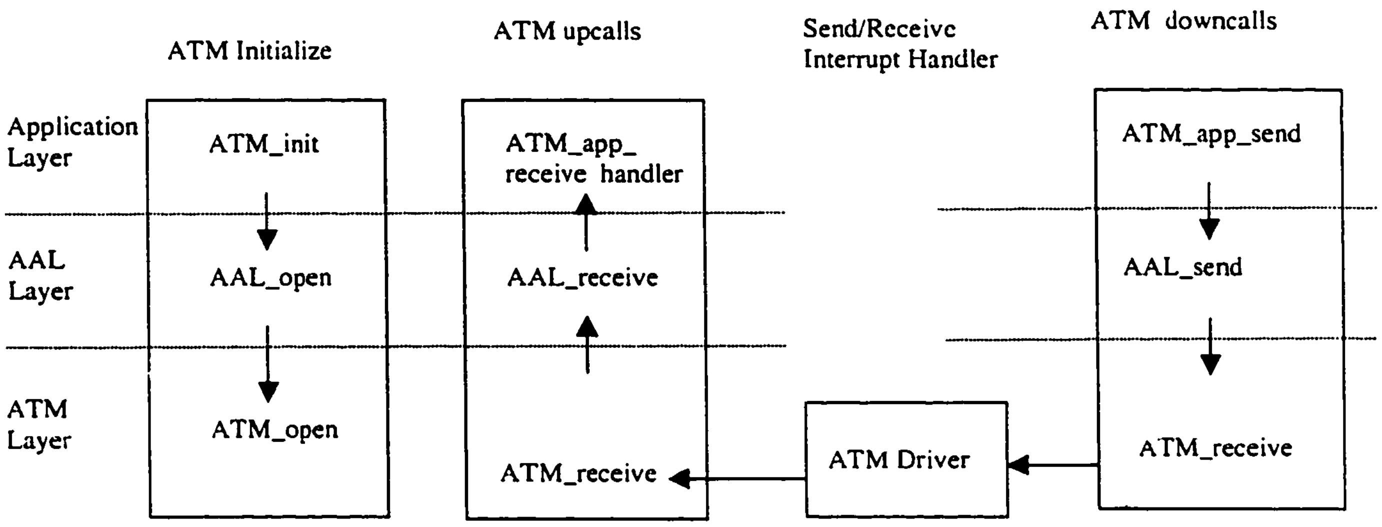


Figure 2.13 Control Flow for Upcall Approach

2.3.2.2 Downcall Approach

The second approach is described as follows: ATM application task checks periodically if there is an incoming ATM cell or a user sending request or not. If it is an incoming ATM cell then the task calls first the ATM layer function ATM_Receive and then calls the AAL layer function AAL_receive to assemble the ATM cells into the AAL message. If it is a user sending request then the task first calls the AAL layer function AAL_send to segment an AAL message into ATM cells, and then calls the ATM layer function ATM_send. So, all of the incoming ATM cell processing and the sending ATM cell processing are in the ATM application task context. The ATM implementations is shown in Figure 2.14

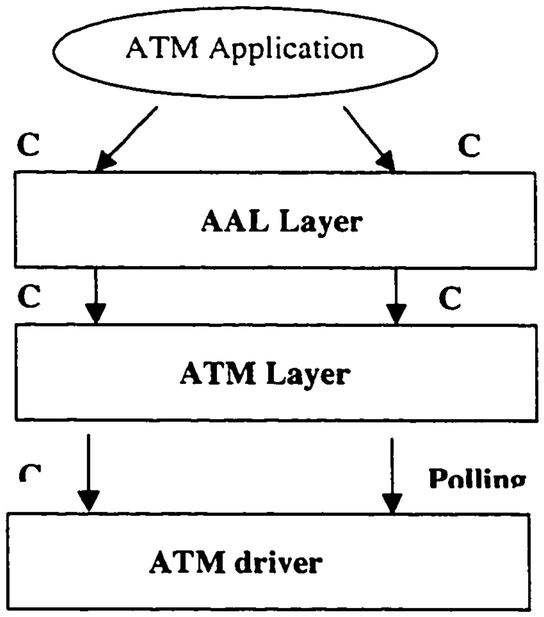


Figure 2.14 ATM Architecture

In the following, we illustrates the skeleton implementations for the ATM protocol for sending/receiving the ATM cells.

(1) skeleton implementation for receiving the ATM cell

```
/* It is an ATM application task */
void ATM_Application_task()
{
   /* enter an infinite loop */
   while (1)
   {
     /* check if there is an ATM cell in shared memory */
     /* it is a non block call */
     if (get_cell(atmCell) == TRUE) {

        validate the received ATM cell

        /* get the port based on ATM header */
        port = get_port_from_cell(atmCell);

        /* downcall the ATM layer to handle the */
        /* received ATM cell */
        ATM_receive(port, atmCell);
```

```
            /* downcall the AAL layer */
            AAL_receive(port, atmCell);

            /* downcall the application layer */
            ATM_app_receive_handler();
         }

       /* check if there is a sending request */
       /* it is a non block call */
       if (get_msg(atmMsg) == TRUE) {

         /* send a AAL message */
         ATM_sender(aalMsg)
       }
}



/* to display the received AAL 5 message */
void ATM_app_receive_handler(aalMessage)
{
   /* display the received the AAL message */
   for (i=0; i < lnegth_of_cell; i++)
   {
     print each aalMessage[i]
   }
}
```

Figure 2.15a The application layer

```
/* AAL_receive is called by the ATM_receive. */
/* The port is from ATM_receive and is used  */
AAL_receive(atmCell, port)
{
   /* handle the received ATM cells */
   assemble the atm cells into aalMsg

}
```

Figure 2.15b AAL layer

```
/* ATM_receive is called by polling task. */
/* The polling task gets the ATM cell and */
/* gets the port based on the ATM header - VPI/VCI */
/* The port is used to get the corresponding handler */
ATM_receive(port, atmCell)
{

  /* handle the ATM cell */
  process the ATM cell

}
```

Figure 2.15c ATM layer

```
void ATM_driver()
{
    get the atm cell from device

}
```

Figure 2.15d ATM driver

(2) skeleton implementation for sending the ATM cell

```
/* This function is called by a sending task */
void ATM_sender(aalMsg)
{
  validate aal message

  /* send message to AAL layer */
  AAL_send(aalMsg)
}
```

Figure 2.16a: The application layer

```
void AAL_send(aalMsg)
{

  segment an aalMsg into ATM cells

  /* sending ATM cells */
  ATM_send(atmCell)
}
```

Figure 2.16b: AAL layer

```
void ATM_send(atmCell)
{

  validate ATM header
  /* send out ATM cell */
  ATM_driver(atmCell);
}
```

Figure 2.16c: ATM layer

```
void ATM_driver(atmCell)
{
   /* send out cell into the device */
   for each bit in ATM cell do send bit
}
```

Figure 2.16d: ATM driver

Figure 2.17 illustrate the control relationship which exist between the various modules cell defined in Figure 2.15 and 2.16 for sending/receiving an ATM. The figure indicates the upcalls and downcalls between layers with arrows, and the interrupt service routine.

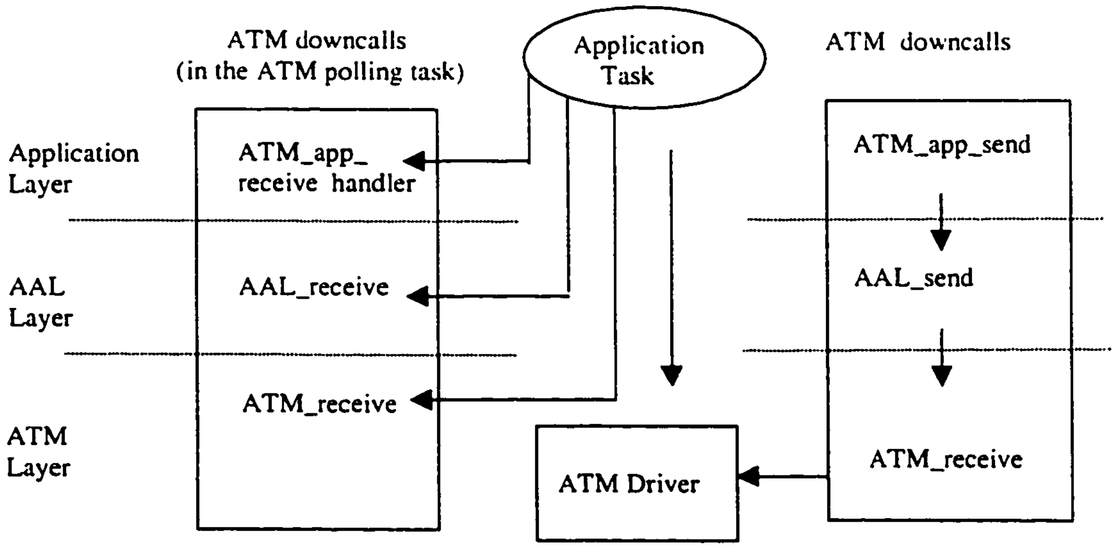


Figure 2.17 Control Flow for Downcall Approach

Chapter 3 Process-based Architecture

3.1 Introduction

A network is specified in terms of layers and protocols. To implement a layered communications software, it is natural to try to map the layers into the processes provided by the supporting operating system or kernel. From point of viewof performance, the inter-layer communication in the process-based approach is our concern because it has an important influence on the performance of the system. The inter-layer communication in the process-based approach is a process-to-process communication and can be performed by one of the following communication mechanisms: message queue, shared memory, mailbox, rendezvous (in Ada) etc. In general, these mechanisms are time-consuming and are not supported by most languages.

A typical example of process-based approach is the communication model proposed by Buhr [BU84, KA90] for a OSI implementation. The main advantages of Buhr's model according to Buhr are that it is intuitively simple and it requires no more rendezvous than his other models (in this model, each package has the minimal number of tasks. So, this model has the minimal number of rendezvous). In this model, each OSI layer is implemented as a package. This package includes the three tasks: one task labeled M in the middle of each package rectangle is the main task for that layer whereas the two tasks labeled T are transporter tasks defined in [BU84]. Buhr's model requires a minimum of two rendezvous to "pass" a message from a higher layer to a lower one or vice versa. In Ada, the rendezvous is very expensive from the performance's view of point. So, the number of Ada rendezvous in a system must be minimized.

In order to minimize the number of tasks in each layer and message passing between the layers, we presented a process-based architecture with one task per layer. As a result only seven tasks are needed for implementing a seven layer OSI protocols, compared to 21 tasks using Buhr's model.

3.2 The Buhr Models

In his book *System Design with Ada*, R. J. A. Buhr[BU84] proposes several "models" for designing layered communications software using Ada tasking to decouple the layers. All of these models are similar in that they introduce concurrency in essentially the same way. Each layer is implemented as an active package (one that contains tasks) and the tasks in a given layer rendezvous with tasks in the layer packages above and below it. The first model that Buhr introduces on page 187 is the simplest and also introduces the least overhead. For this reason, it is considered the "best case" for Buhr's approach. The presentation of what will be referred to as "Buhr's first model" will be in terms of an icon-oriented diagraming technique introduced by Buhr that is both elegant and efficient, and which is a fundamental part of his design methodology. In the discussions that follow, these "Buhrgrams" will be used to illustrate the concepts. Buhr's first model is illustrated in Fig. 3.1. In this diagram, the large rectangles denote Ada packages whereas the small rectangles inside the large ones represent procedures. Those procedure rectangles that touch the outer package rectangles are the procedures that are visible from outside the package (i.e. included in the package specification). The parallelograms inside the package rectangles represent tasks. The arrows without little circles indicate calls to either procedures or tasks. The direction of these arrows indicate which program unit is calling which other program unit. The head of the arrow indicates the program unit that is being called whereas the tail indicates the calling unit. The arrows with little circles at the tail indicate the parameters that are being passed. The dots at the head of some of the calling arrows indicate the presence of guards, i.e., the task entry is conditional.

In Buhr's first model, all calls between layers originate from the higher layer. The advantages of this model according to Buhr are: 1) it is intuitively simple, 2) it requires no more rendezvous than his other models, and 3) it is structurally free from constructs that could lead to deadlock in one form or another. Each OSI layer in this model is implemented as a package. The task labeled M in the middle of each package rectangle is the main task for that layer whereas the tasks labeled T are transporter tasks as defined by chapter 3 of Buhr [BU84]. From Fig. 3.1 it can be seen that it requires a minimum of two rendezvous to "pass" a message from one layer to another. It also requires at least one procedure call. The procedures in this model are labeled p in Fig. 3.1. The main task does whatever processing is required at the layer in which it resides and thus could conceivably have subtasks or procedures. The transporter tasks are merely the decoupling mechanism that allows the main task to work asynchronously from tasks in adjacent layers without having to periodically poll them or be polled by them to see if a message is ready to be passed between layers. In practice, an implementation would probably only pass the protocol header associated with the layer rather than the entire message.

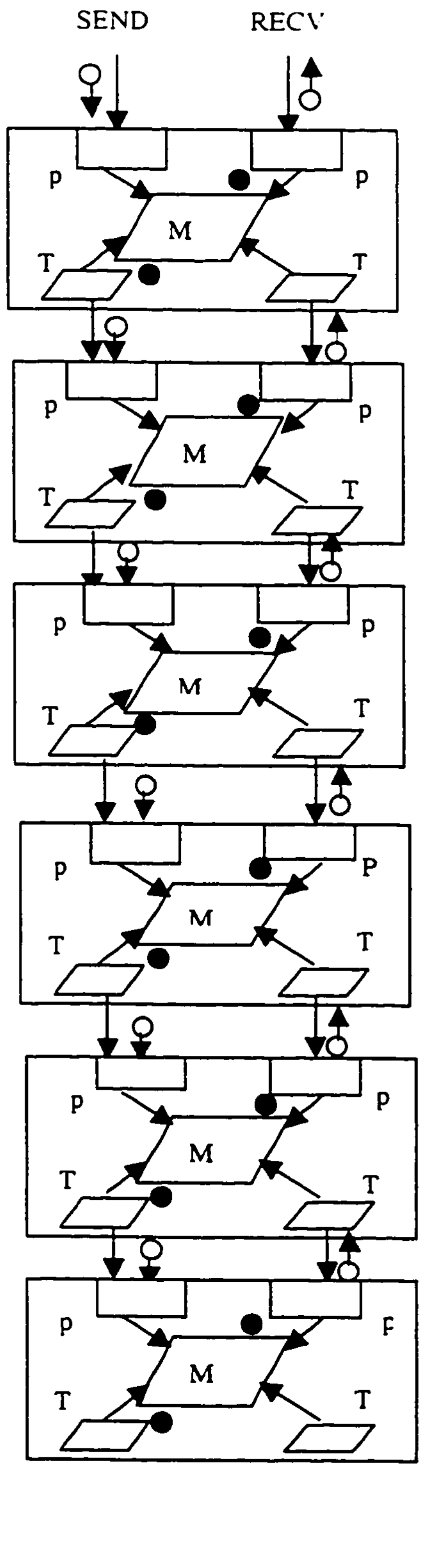


Fig 3.1 Buhr model architecture

3.3 Process-based Architecture

In the above, we have discussed the Buhr's process-based model. In [HW89], Howes and Weaver report on the performance of Buhr's model by simulating it on a VAX 11/785. The results show that the time to services a single message (an integer) attributable to the Ada overhead mainly caused by rendezvous was found to be 11.2 ms. They also reports that on a VAX 11/785, a procedure call takes approximately 19.2 µs, while a simple producer-consumer rendezvous where the only parameter passed is an integer takes 835 µs. This cost of process-based approach is high and not acceptable in a high performance implementation. The performance of the process-based system depends heavily on the number of tasks in the system and the communication overhead between the tasks. In order to achieve this goal, we presented a process-based architecture with one task per each layer. It means that only seven tasks are needed for implementing a seven layer OSI protocols, compared to 21 tasks using Buhr's model. The architecture with one task per layer is shown in Fig 3.2.

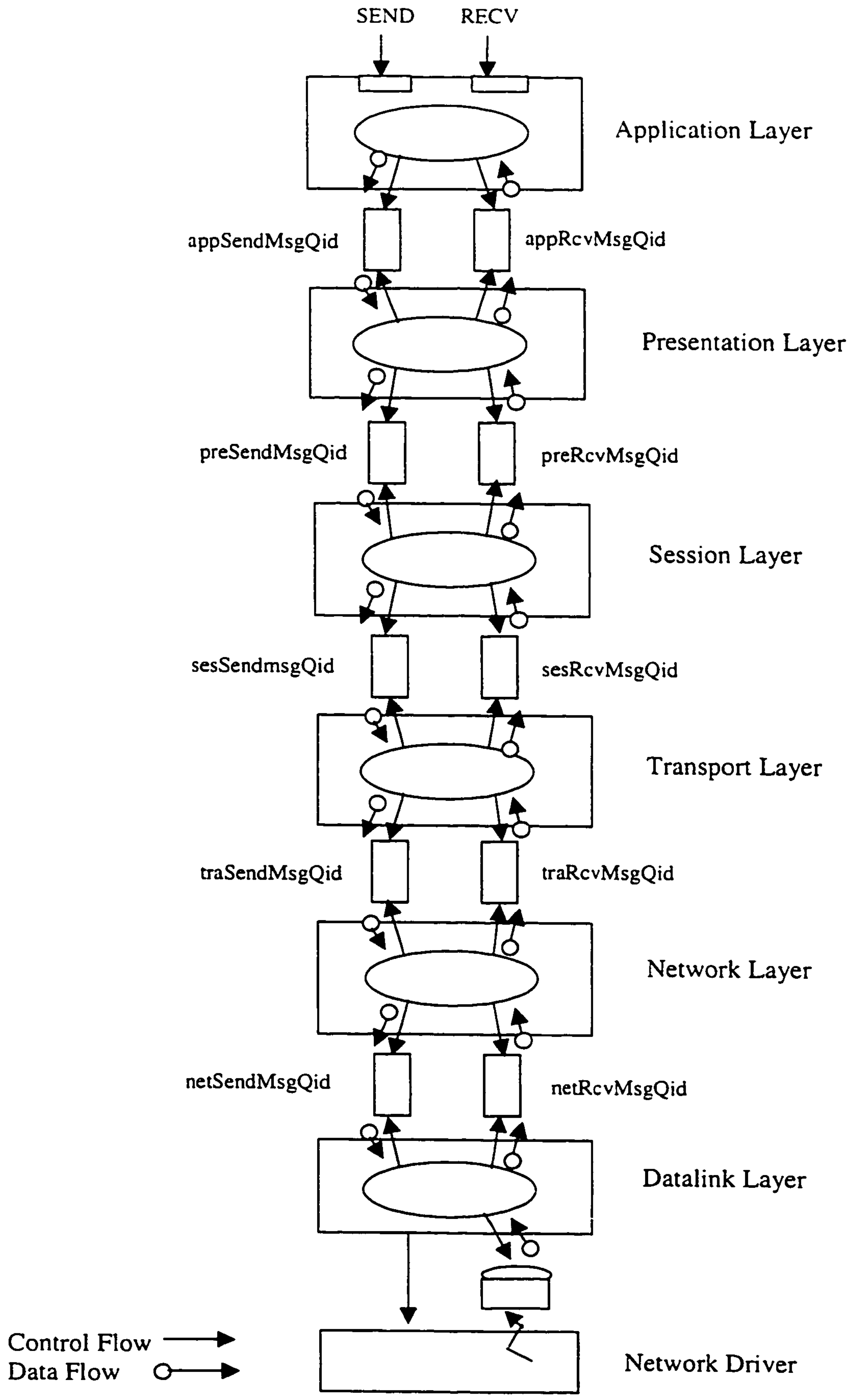

SEND
RECV
Application Layer
appSendMsgQid
appRcvMsgQid
Presentation Layer
preSendMsgQid
preRcvMsgQid
Session Layer
sesSendmsgQid
sesRcvMsgQid
Transport Layer
traSendMsgQid
traRcvMsgQid
Network Layer
netSendMsgQid
netRcvMsgQid
Datalink Layer
Network Driver
Control Flow
Data Flow

Figure 3.2 Process-based OSI architecture

In this diagram, the large rectangles denote a program or a set of programs which performs the layer's functionalities whereas the small rectangles inside the large ones represent procedures. Those procedure rectangles that touch the outer package rectangles are the procedures that are visible from the outside. The big circles inside the rectangles represent tasks. The big arrows indicate calls to either procedures or put/(get) the messages into/(from) the queues. The direction of these arrows indicate which program unit is calling which other program unit. The head of the arrow indicates the program unit that is being called whereas the tail indicates the calling unit. The smaller arrows with little circles at the tail indicate the parameters/data that are being passed. The small long rectangle represents a message queue.

The two adjacent layers communicate by the two message queues. One is the send message queue, the other is the receive message queue. The bottom layer is a network driver. The transmit part of this layer consists of a set of procedures which are called by the data link layer for the packet transmission. The receive part of this layer is in the ISR (Interrupt Service Routine), and it communicates with the data link layer by the mailbox.

As in Buhr's first model, all calls between layers originate from the higher layer. Each OSI layer in this model is implemented as a task. So the layer to layer communication is the task to task communication, which is done by the message queues between the layers. The task in each layer does whatever processing is required at the layer in which it resides and thus could conceivably have procedures. It is also responsible for communication with the layers below and above.

The task in each layer periodically polls the send queue which is above it and the receive queue which is below it. For example, the task in the presentation layer polls the send queue "appSendMsgQid" which is above it and the receive queue "preRcvMsgQid" which is below it. If the task finds some messages in the send or receive queues, then it call the function "removeFromQueue(...)" to get the message. If this message is from above (i.e. get a message from send queue), then it does the processing and appends a header into the message. Finally the task sends this message to the lower layer by calling the function "addToQueue(...) ". If this message is from below (i.e. get a message from receive queue), then it does the processing and removes the header from the message. Finally the task passes this message to the higher layer by calling the function "addToQueue(...)". In summary, the task in each layer polls the send/receive queues to check if the higher layer has requested to send a message or a lower layer has received a message. In either of cases, the task picks up the message and does the processing, and then passes the messages to the lower layer (for the send) or the higher layer (for the receive).

The task in each layer is created by a unix-like system call "fork(process_name)". At the system initialization, the task and the two message queues in each layer are created. The skeleton for the Figure 3.2 is illustrated as follows:

```
void application_open()
{

    /* create application task */
    fork(application_layer);

    /* create sending and receiving message queues */
    appSendMsgQid = cerateMessageQueue(....);
    appRcvMsgQid = createMessageQueuq(…);

    /* invoke the function in presentation layer */
    presentation_open()
}



void presentation_open()
{
     /* create application task */
    fork(presentation_layer);

    /* create sending and receiving message queues */
    preSendMsgQid = cerateMessageQueue(….);
    preRcvMsgQid = createMessageQueuq(…);

    /* invoke the function in presentation layer */
    session_open()
}



void session_open()
{

    /* create application task */
    fork(session_layer);

    /* create sending and receiving message queues */
    sesSendMsgQid = cerateMessageQueue(….);
    sesRcvMsgQid = createMessageQueuq(…);

    /* invoke the function in presentation layer */
    transportation_open()
}



void transportation_open()
{
     /* create application task */
    fork(transportation_layer);
```

```
   /* create sending and receiving message queues */
   traSendMsgQid = cerateMessageQueue(....);
   traRcvMsgQid = createMessageQueuq(...);

   /* invoke the function in presentation layer */
   network_open()
}



void network_open()
{
    /* create application task */
   fork(network_layer);

   /* create sending and receiving message queues */
   netSendMsgQid = cerateMessageQueue(....);
   netRcvMsgQid = createMessageQueuq(...);

   /* invoke the function in presentation layer */
   datalink_open()
}



void datalink_open()
{

   /* create application task */
   fork(datalink_layer);

   /* create the shared memory for the incoming message */
   dadSharedmemQid = cerateSharedmemory(....);

}
```

Figure 3.3a Initiation

```
void application_layer()
{

 /* enter an infinitive loop */
 for (;;)
 {
       if user issued a send request, then get the sending msg;

       /* send the message to the presentation layer */
       addToQueue(appSendMsgQid);

       /* check if there is a message in the queue */
       /* appRcvMsgQid */
       if (queueIsNotEmpty(appRcvMsgQid))
       {
           /* get the message from the queue appRcvMsgQid */
           msgPtr = removeFromQueue(appRcvMsgQid);

           /* strip off an application layer header */
           removeAppHdr(msgPtr);

           /* call the user's receiver handler */
           userRcvHandler(msgPtr);
       }
   } /* for(;;) */
}
```

Figure 3.3b: Application Process

```
void presentation_layer()
{
   /* enter an infinitive loop */
   for(;;)
   {
     /* check if there is a message in the queue */
     /* appSendMsgQid */
     if (queueIsNotEmpty(appSendMsgQid))
     {
        /* get the message from the queue appSendMsgQid */
        msgPtr = removeFromQueue(appSendMsgQid);

        /* append a presentation layer header */
        appendPreHdr(msgPtr);

        /* send this message to the lower layer */
        addToQueue(preSendMsgQid)
     }

     /* check if there is a message in the queue */
     /* preRcvMsgQid */
    if (queueIsNotEmpty(preRcvMsgQid))
    {
        /* get the message from the queue preRcvMsgQid */
        msgPtr = removeFromQueue(preRcvMsgQid);

        /* strip off the presentation header*/
        removePreHeader(msgPtr);

        /* send this message to the application */
        addToQueue(appRcvMsgQid, msgPtr);
     }
   } /* for (;;) */
}
```

Figure 3.3c: Presentation Process

```
void session_layer()
{
   /* enter an infinitive loop */
   for(;;)
   {
     /* check if there is a message in the queue */
      /* preSendMsgQid */
     if (queueIsNotEmpty(preSendMsgQid))
     {
        /* get the message from the queue preSendMsgQid */
        msgPtr = removeFromQueue(preSendMsgQid);

        /* append a session layer header */
        appendSesHdr(msgPtr);

        /* send this message to the lower layer */
        addToQueue(sesSendMsgQid, msgPtr)
     }

     /* check if there is a message in the queue  */
     /*  sesRcvMsgQid */
    if (queueIsNotEmpty(sesRcvMsgQid))
    {
       /* get the message from the queue preRcvMsgQid */
       msgPtr = removeFromQueue(sesRcvMsgQid);

       /* strip off the presentation header*/
       removeSesHeader(msgPtr);

       /* send this message to the presentation layer */
       addToQueue(preRcvMsgQid, msgPtr);
     }
  } /* for (;;) */
}
```

Figure 3.3d: Session Process

```
void transport_layer()
{
  /* enter an infinite loop */
  for(;;)
  {
    /* check if there is a message in the queue */
    /* sesSendMsgQid */
    if (queueIsNotEmpty(sesSendMsgQid))
    {
       /* get the message from the queue sesSendMsgQid */
       msgPtr = removeFromQueue(sesSendMsgQid);

       /* append a transport layer header */
       appendTraHdr(msgPtr);

       /* send this message to the lower layer */
       addToQueue(traSendMsgQid)
    }

    /* check if there is a message in the queue */
    /* traRcvMsgQid */
   if (queueIsNotEmpty(traRcvMsgQid))
   {
      /* get the message from the queue traRcvMsgQid */
      msgPtr = removeFromQueue(traSendMsgQid);

      /* strip off the transport layer header*/
      removeTraHeader(msgPtr);

      /* send this message to the session layer */
      addToQueue(sesRcvMsgQid, msgPtr);
   }
  } /* for (;;) */
}
```

Figure 3.3e: Transport Process

```
void network_layer()
{
  /* enter an infinitive loop */
  for(;;)
  {
    /* check if there is a message in the queue
    /* traSendMsgQid */
    if (queueIsNotEmpty(traSendMsgQid))
    {
       /* get the message from the queue traSendMsgQid */
       msgPtr = removeFromQueue(traSendMsgQid);

       /* append a network layer header */
       appendNetHdr(msgPtr);

       /* send this message to the lower layer */
       addToQueue(netSendMsgQid)
    }

    /* check if there is a message in the queue
    /* netRcvMsgQid */
   if (queueIsNotEmpty(netRcvMsgQid))
   {
      /* get the message from the queue netRcvMsgQid */
      msgPtr = removeFromQueue(netRcvMsgQid);

      /* strip off the network header*/
      removeNetHeader(msgPtr);

      /* send this message to the transport layer */
      addToQueue(traRcvMsgQid, msgPtr);
   }
 } /* for (;;) */
}
```

Figure 3.3f: Network Process

```
void data_layer()
{
  /* enter an infinitive loop */
  for(;;)
  {
    /* check if there is a message in the queue */
    /* netSendMsgQid */
    if (queueIsNotEmpty(netSendMsgQid))
```

```
        {
            /* get the message from the queue netSendMsgQid */
            msgPtr = removeFromQueue(netSendMsgQid);

            /* append a data link layer header */
            appendDatHdr(msgPtr);

            /* send this message to the lower layer */
            sendMsgToNetDriver(msgPtr);
        }

        /* check if there is a message in the shared */
        /* memory */
       if (isNotEmpty(dadSharedmemQid))
       {
          /* get the message from the queue preRcvMsgQid */
          msgPtr = get_message(dadSharedmemQid);

          /* strip off the data link header */
          removeDatHeader(msgPtr);

          /* send this message to the network */
          addToQueue(netRcvMsgQid);
       }
   } /* for (;;) */
}
```

Figure 3.3g: Data Link Process

```
void Network_driver()
{
    get the message from device;

    /* store the received cell into the shared memory */
    put_message(dadSharedmemQid, msgPtr);
}
```

Figure 3.3h: Network Driver

3.4 ATM architecture using process-based approach

3.4.1 ATM architecture

In the following we show how the process-based approach is used to implement a three layer-ATM switch. Figure 3.4 illustrates the ATM implementation.

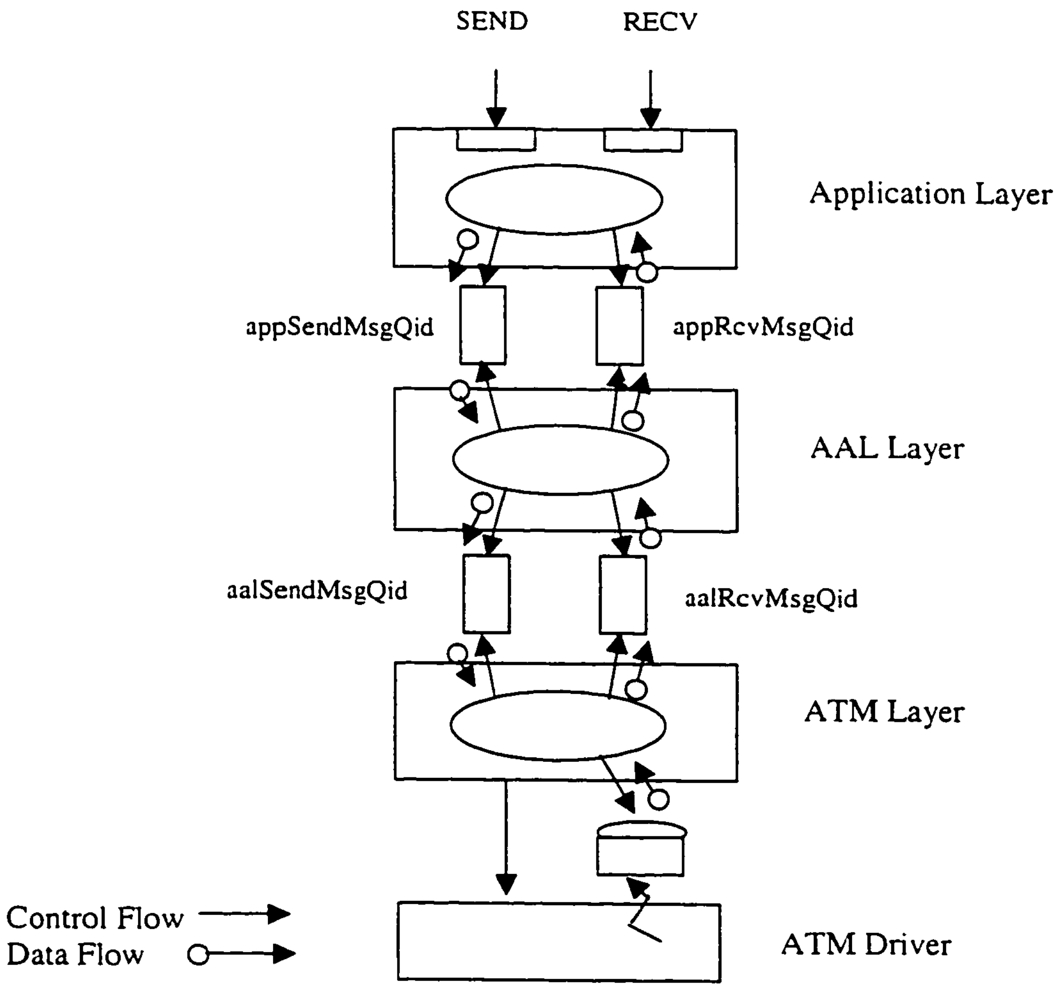


Figure 3.4 Process-based ATM architecture

For the ATM send/receive, the main functionalities in each layer can be described as follows:

- Bottom layer (ATM driver)

The ATM driver consists of the two parts. One is for the transmission part, the other is for the reception part. The former is a set of function calls which are called by the ATM layer for sending an ATM cell. The latter is in the ISR and is trigged by the arrival of an ATM cell. After the ISR receives a ATM cell, it puts the cell into the shared memory.

- ATM layer

The task in the ATM layer periodically polls the message queue "aalSendMsgQid " and the shared memory "atmSharedmemQid" to checks if the AAL layer has sent a request or an ATM cell has received or not. If the task finds that the message queue "aalSendMsgQid " is not empty, it gets the cell from the queue and does the processing, and then calls the functions "`sendMsgToAtmDriver`(cellPtr)" in the ATM driver to send the cell (see Figure 3.5d). If the task finds there is a cell in the shared memory, it gets the cell from the shared memory and does the processing, and then passes this cell to the AAL layer by calling the function "`addToQueue(aalRcvMsgQid, cellPtr)`" (see Figure 3.5d).

- AAL layer

The task in the AAL layer periodically polls the message queues "appSendMsgQid " and "aalRcvMsgQid" to check if the application layer has sent a request or the ATM layer has passed an ATM cell or not. If the AAL task finds that the message queue "appSendMsgQid " is not empty, it gets the message from the queue and does the processing, and then calls the functions "`addToQueue`(aalSendMsgQid, cellPtr)" to send the message to the ATM layer (see Figure 3.5c). If the AAL task finds that the message queue "aalRcvMsgQid" is not empty, it gets the cell from the queue and does the processing, and then calls the function "addToQueue(appRcvMsgQid, msgPtr)" (see Figure 3.5c) to pass the message to the application layer.

- Application

The task in the application layer periodically polls the user send queue (not shown in Figure 3.4) and the message queues "appRcvMsgQid" to check if the user has sent a request or the AAL layer has passed a message or not. If the application task finds that the user sent a request, it gets the message from the user queue and does the processing, and then calls the functions "addToQueue(appSendMsgQid, msgPtr)" to send the message to the AAL layer (see Figure 3.5b). If the application task finds that the message queue "appRcvMsgQid" is not empty, it gets the message from the queue and does the processing, and then passes the message to the user.

In addition, the application `layer` includes an initialization function: `application_open` (see Figure 3.5a). This function performs the initiation code, which creates the application task and the two message queues for communication with the AAL layer. The function `application_open` in turn downcalls the AAL layer (the function AAL_open) initialization code, which in turn downcalls the ATM layer (the function ATM_open) initialization code.

Figure 3.5 illustrates a skeleton implementation of the ATM protocol described in the above for the initialization, sending cell and incoming cell processing. This figure includes the connection initiation code, which involves the downcall sequence, and sending cell and incoming cell processing.

```
void application_open()
{

   /* create application task */
   fork(application_layer);

   /* create sending and receiving message queues */
   appSendMsgQid = cerateMessageQueue(....);
   appRcvMsgQid = createMessageQueuq(...);

   /* invoke the function in presentation layer */
   AAL_open()
}



void AAL_open()
{
    /* create application task */
   fork(AAL_layer);

   /* create sending and receiving message queues */
   aalSendMsgQid = cerateMessageQueue(....);
   aalRcvMsgQid = createMessageQueuq(...);

   /* invoke the function in ATM layer */
   ATM_open()
}



void ATM_open()
{

   /* create application task */
   fork(ATM_layer);

   /* create the shared memory for the incoming cell */
   atmSharedMemQid = cerateSharedmemory(....);

}
```

Figure 3.5a: Initiation

```
void application_layer()
{

 /* enter an infinitive loop */
 for (;;)
 {
   if user issued a send request, then get the sending msg;

   /* send the message to the presentation layer */
   addToQueue(appSendMsgQid);

   /* check if there is a message in the queue */
   /* appRcvMsgQid */
   if (queueIsNotEmpty(appRcvMsgQid))
   {
       /* get the message from the queue appRcvMsgQid */
       msgPtr = removeFromQueue(appRcvMsgQid);

       /* strip off an application layer header */
       removeAppHdr(msgPtr);

       /* call the user's receiver handler */
       userRcvHandler(msgPtr);
   }
 } /* for(;;) */
}
```

Figure 3.5b: Application Process

```
void AAL_layer()
{
  /* enter an infinitive loop */
  for(;;)
  {
    /* check if there is a message in the queue */
    /* appSendMsgQid */
    if (queueIsNotEmpty(appSendMsgQid))
    {
       /* get the message from the queue appSendMsgQid */
       msgPtr = removeFromQueue(appSendMsgQid);

       /* segment the message into cells */
       segment a message into the cells
```

```
        /* send the cells to the lower layer */
        addToQueue(aalSendMsgQid)
      }

      /* check if there is a message in the queue */
      /* preRcvMsgQid */
     if (queueIsNotEmpty(aalRcvMsgQid))
     {
        /* get the message from the queue preRcvMsgQid */
        msgPtr = removeFromQueue(aalRcvMsgQid);

        /* strip off the cell header*/
        removeCellHeader(msgPtr);

         resemble the cells into a message

        /* send this message to the application */
        addToQueue(appRcvMsgQid, msgPtr);
     }
   } /* for (;;) */
}
```

Figure 3.5c: AAL Process

```
void ATM_layer()
{
   /* enter an infinitive loop */
   for(;;)
   {
     /* check if there is a message in the queue */
     /* netSendMsgQid */
     if (queueIsNotEmpty(aalSendMsgQid))
     {
        /* get the message from the queue netSendMsgQid */
        msgPtr = removeFromQueue(aalSendMsgQid);

        /* append a data link layer header */
        appendDatHdr(msgPtr);

        /* send this message to the lower layer */
        sendMsgToAtmDriver(msgPtr);
     }

     /* check if there is a message in the shared */
     /* memory */
    if (isNotEmpty(atmSharedmemQid))
    {
       /* get the message from the queue preRcvMsgQid */
       cellPtr = get_message(atmSharedmemQid);

       /* strip off the data link header */
       removeDatHeader(cellPtr);

       /* send this message to the AAL layer */
       addToQueue(aalRcvMsgQid, cellCell);
    }
 } /* for (;;) */
}
```

Figure 3.5d: ATM Process

```
void ATM_driver()
{
    get the message from device;

    /* store the received cell into the shared memory */
    put_message(atmSharedmemQid, msgPtr);
}
```

Figure 3.5e: ATM Network Driver

3.4.2 Control Flow

Figure 3.6 illustrate the control relationship which exist between the various modules defined in Figure 3.5 for sending/receiving an ATM cell. The figure indicates the call direction (the task in a layer use both upcall and downcall) and the interrupt service routine with arrows.

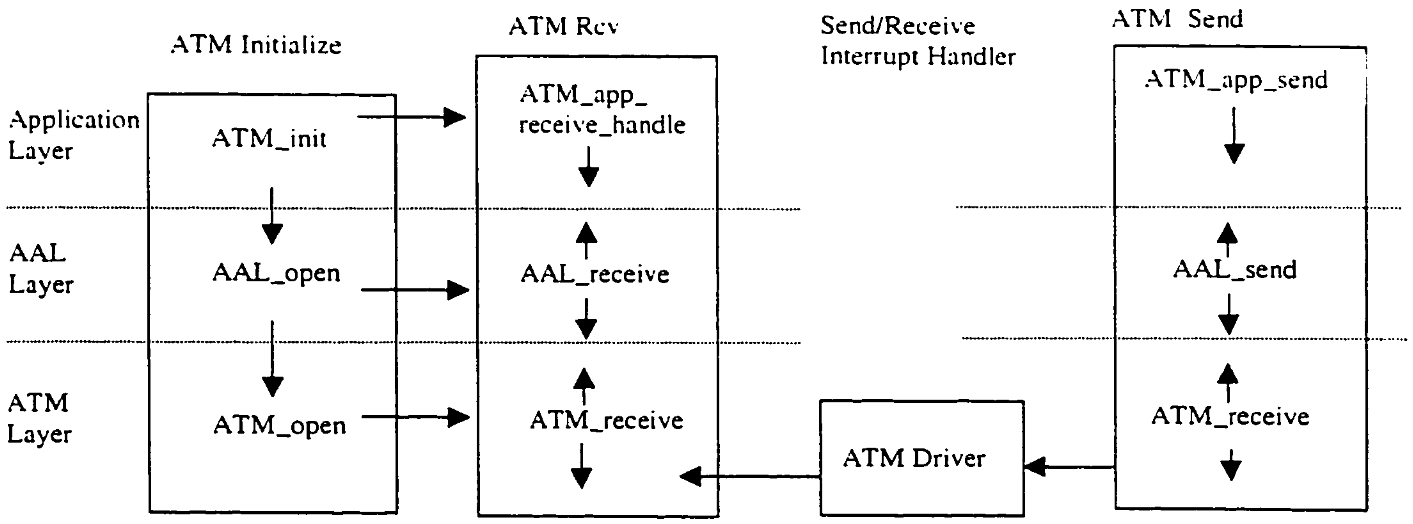


Figure 3.6 Control flow for ATM process-based architecture

For a process-based architecture, the process-to-process communication is a time-consuming procedure. One must reduce the number of processes (e.g., combine multiple layers into process) as much as possible in order to improve the system's performance. In Chapter 5, we will make the performance analysis for the process-based architecture.

The distinguishing feature of process-based is the ability to take advantage of a multiprocessor environment. So, it is particularly suitable for a multiple-processor system, where each processor executes one or several layer processes.

Chapter 4 Hybrid Architectures

4.1 Introduction

A process-based implementation can achieve a maximal parallelism but the inter-layer communication is complex and incurs a high penalty in time. A procedure-based implementation has a simpler, fast inter-layer communication mechanism but the wait time for a user call may be very long. The reason for that is that when a downcall or upcall is invoked by a user or an incoming message, the procedure call continues until the message is sent to the network driver for the case of a user call or the message has been passed to the user handler for the case of an incoming message. It means that a new arrival event (the user call or an incoming message) cannot be processed when a previous procedure call is in the progress.

A hybrid approach can solve this problem. The hybrid architecture is a combination of the process-based implementation and procedure-based implementation, i.e., some layers are implemented as a set of procedures; and other some layers are implemented as processes. So it can benefit from the advantages of the process-based implementation and procedure-based implementation. Inter-layer communication in the hybrid implementation may be based on procedure call, process-to-process communication or ISR-to-process communication.

The key to using the hybrid approach for implementing a layered network software is to choose which layers should be implemented as a task and which layers should be implemented a set of procedure calls. In general, a complex layer (By the complex layer, we mean that this layer includes many functionalities to be done by software. For example, transport layer) should be implemented a task, while a simple layer should be implemented as a set of procedure call. Based on this principle, for an OSI seven model, the transport layer, application layer and data link layer should be implemented as tasks for the following reasons:

- The application layer is complex in that it includes the several protocols (ACSE -- Association control service element, ...). In addition, the user will communicate directly with the application layer. The waiting time for a user call can be reduced if the application layer is implemented as a task.
- The transport layer must guarantee the error-free message transmission and message sequence from the source to the destination.
- The data link layer includes a complex MAC (Medium access control) protocol such as CDMA/CD (Carrier sense multiple access with collision detection), which is used to complete the medium access control.

The presentation layer, session layer and network layer (the network layer in the end system does not the routing function, so it is not complex) is relatively simple compared with the other layers. Figure 4.1 shows the implementation of an OSI seven layer model using hybrid approach.

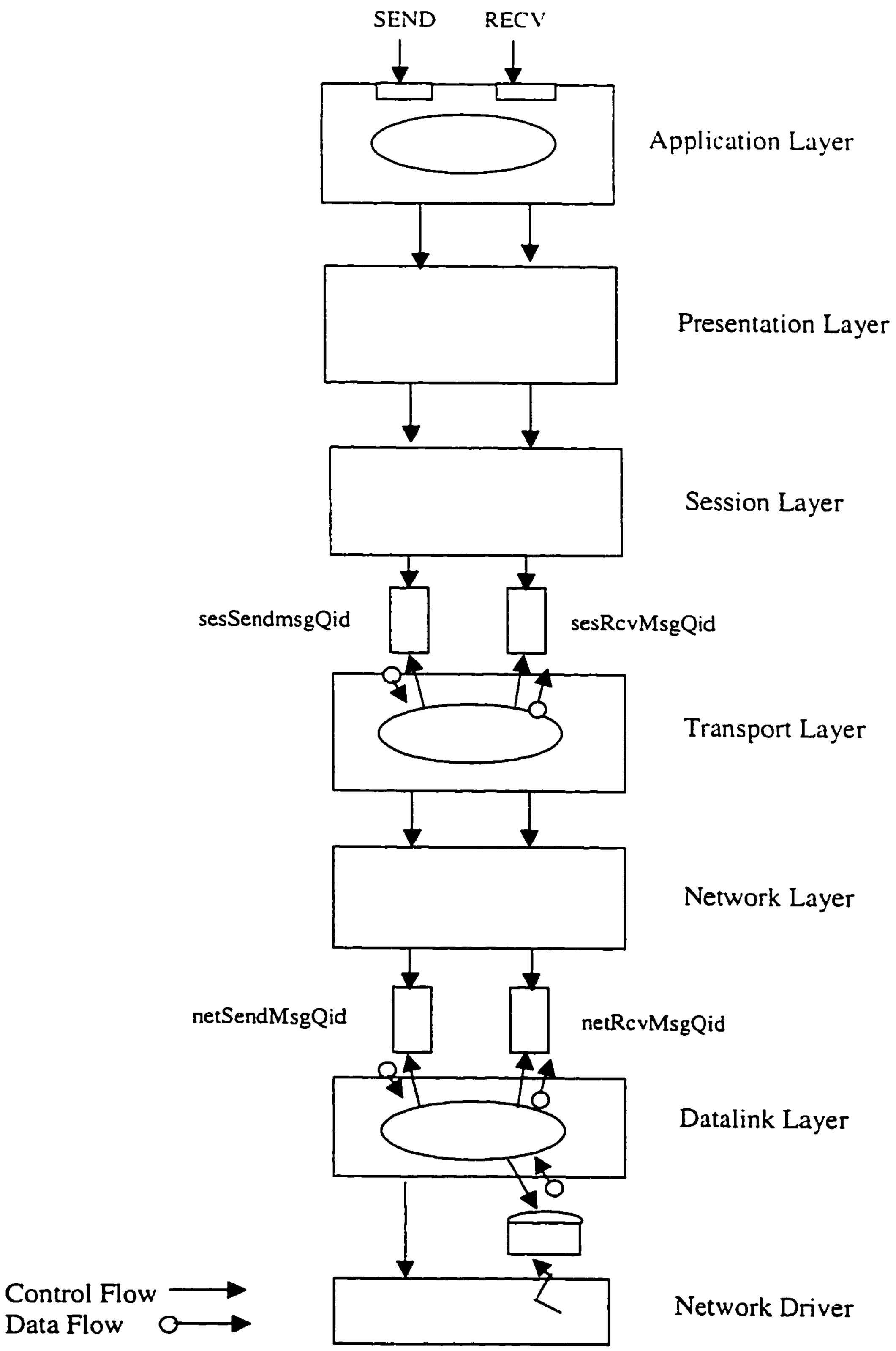


Figure 4.1 Hybrid model for OSI architecture

As in Figure 3.1 of the last chapter, the large rectangles denote a program or a set of programs which performs the layer's functionalities whereas the small rectangles inside the large ones represent procedures. The big circles inside the rectangles represent tasks. The big arrows indicate calls to either procedures or put/(get) the messages into/(from) the queues. The direction of these arrows indicate which program unit is calling which other program unit. The head of the arrow indicates the program unit that is being called whereas the tail indicates the calling unit. The two message queues, send and receive, are needed between the two adjacent layers if they are implemented as tasks. The bottom layer is a network driver. The transmit part of this layer consists of a set of procedures which are called by the data link layer for the packet transmission. The receive part of this layer is in the ISR (Interrupt Service Routine), and it communicates with the data link layer by the mailbox.

As in the process-based architecture, all calls between layers originate from the higher layer. The application layer is implemented as a task. It communicates with the presentation layer by procedure call. Both presentation layer and session layer are implemented as a set of procedure calls. They communicate with each other by the procedure calls. The transport layer is implemented as a task, which communicates with the session layer and network layer by the message queues. The network layer is implemented as a set of procedure calls, which communicate with the transport layer by the procedure call and with the data link layer by the message queues. The data link layer is implemented as a task, which communicates with the network layer by the message queue and with the network driver by the procedure call in the transmission and by the mailbox (shared memory) in the reception.

The task in each layer does whatever processing is required at the layer in which it resides and thus could conceivably have procedures. It is also responsible for communication with the layers below and above.

4.2 ATM architecture using hybrid approach

4.2.1 ATM architecture

In the following we show how the hybrid approach is used to implement a three layer-ATM switch. Figure 4.2 illustrates the ATM implementation.

ATM network software (Figure 4.2) is organized into the tasks (Application Program, ATM receive part), procedures (AAL, ATM send part and ATM driver send part) and interrupt service routines (ATM driver receive part). There is no single communication mechanism for inter-layer communication: message queue (process-process), procedure calls and interrupt to task are used to communicate. For the transmission, the user issues a send request to the application task, which in turn sends request to AAL layer by a procedure call. The AAL layer performs the message segmentation and then requests its transmission by a procedure call to the ATM layer. The ATM prepares its header and passes the resulting cell to the ATM driver by procedure call. After processing by the ATM driver, control returns to the ATM send procedure and then immediately to the AAL layer. When a cell is received by the ATM driver receive part, an interrupt is generated and processed by the ATM driver. This driver put the cell into a mailbox accessible to the ATM task. This mailbox is periodically accessed by the ATM task to extract the incoming cells. The communication between the ATM task and AAL layer is performed by the message queue. The application task needs to periodically poll the message queue "atmRcvMsgQid" to extract the incoming cells. If there is a cell in the message queue. then the application task will call the AAL layer to assemble the cells into the message.

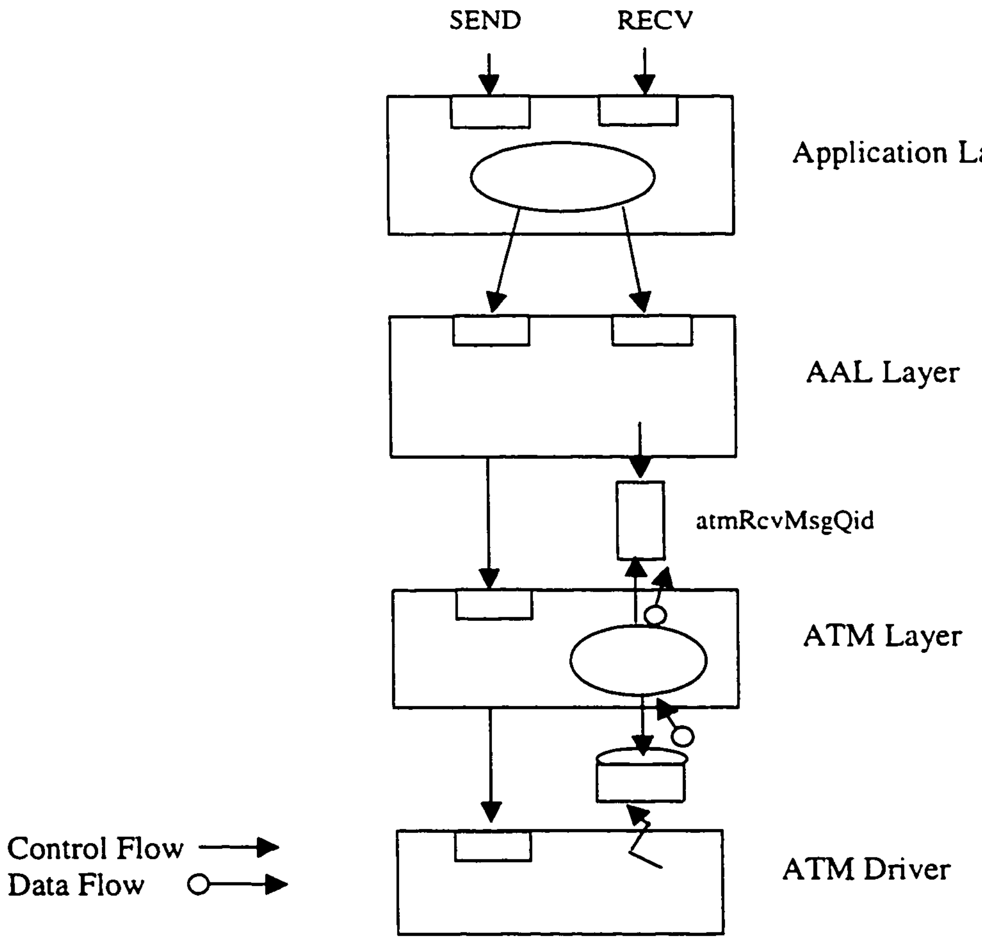


Figure 4.2 Hybrid ATM architecture

In the following we illustrates a skeleton implementation for the Figure 4.2.

```
void application_open()
{

   /* create application task */
   fork(application_layer);

   /* invoke the function in presentation layer */
   AAL_open()
}
```

```
void AAL_open()
{
   /* invoke the function in ATM layer */
   ATM_open()
}



void ATM_open()
{

   /* create ATM task */
   fork(ATM_layer);

   /* create the atm message queue */
   atmRcvMsgQid = createMessageQueue(…);

   /* create the shared memory for the incoming message */
   atmSharedMemQid = cerateSharedMemory(….);

}
```

Figure 4.3a Initiation

```
/* It is an ATM application task */
void ATM_Application_task()
{
   /* enter an infinitive loop */
   for(;;)
   {

   /********************/
   /* for transmission */
   /********************/
   if user issued a send request, then get the sending msg;

   /* send the message to the AAL layer */
   AAL_send(appSendMsg);

   /*****************/
   /* for reception */
   /*****************/
   /* check if there is an ATM cell in message queue */
   /* it is a non block call */
   if (queueIsNotEmpty(atmRcvMsgQid) == TRUE) {

          /* get the cell */
          msgPtr = removeFromQueue(atmRcvMsgQid);

          validate the received ATM cell

          /* downcall the AAL layer to handle the */
          /* received ATM cell */
          AAL_receive(atmCell, msgPtr);

          /* all the user handler */
          ATM_app_receive_handler(msgPtr);
   }
}
```

Figure 4.3b: Application Process

```
void AAL_send(msgPtr)
{
   /* append a AAL layer header */
   appendPreHdr(msgPtr);

   segment the message into the cells;

   /* send the cells to the ATM layer */
   ATM_layer_send(cellPtr);
}

void AAL_receive(cellPtr, msgPtr)
{
   /* handle the received ATM cells */
   assemble the atm cells into aalMsg

}
```

Figure: 4.3cAAL layer

```
void ATM_send(cellPtr)
{
   make the processing;

   append the ATM header;
```

```
    /* call the ATM driver */
    ATM_driver_send(cellPtr);
}


void ATM_receive()
{
   /* enter an infinitive loop */
   for(;;)
   {
     /* check if there is a cell in the shared memory */
    if (queueIsNotEmpty(atmSharedmemQid))
    {
       /* get the cell from the shared memory */
       cellPtr = get_cell(atmSharedmemQid);

       /* strip off the ATM header */
       removeDatHeader(cellPtr);

       /* send the payload to the high layer by message */
       /* queue */
       addToQueue(atmRcvMsgQid);
    }
  } /* for (;;) */
 }
```

Figure 4.3d: ATM Layer

```
void ATM_driver_send(cellPtr)
{

   send cell to the network;

}


void ATM_driver_receive()
{
     get the message from device;

     /* store the revived cell into the shared memory */
     put_message(atmSharedmemQid, cellPtr);
}
```

Figure 4.3e: ATM Network Driver

The main functionalities for the above codes can be described as follow:

- Bottom layer (ATM driver)

For the reception, it is responsible for receiving a ATM cell and putting it into the shared memory. For the transmission, it is responsible for sending a cell to the network.

- ATM layer

For the reception, the polling task **ATM_Task** periodically checks if an ATM cell has arrived or not. If it has, the polling task gets the cell from the shared memory and then insert the cell into the message queue "atmRcvMsgQid". The application task periodically polls this queue to check if a cell has arrived or not. If there are cells in the message queue, then the application task calls the AAL layer to resemble the cells into the message. For the transmission, the ATM subroutine **ATM_send** calls the ATM driver to send a cell.

- AAL layer

For the reception, the subroutine **AAL_receive** is called by the application task **ATM_Application_task** to assemble the received ATM cells into the AAL messages. For the transmission, the AAL subroutine **AAL_send** calls the ATM layer to send a cell.

- Application layer

The application subroutine **application_open** is called during the system initialization, which in turn calls the **AAL_open** in the AAL layer, which in turn calls **ATM_open** in the ATM layer. These subroutines are used to create the application task, the ATM task, ATM message queue and shared memory. After the system initialization the user send request and the incoming cells will be accepted. For the reception, the application task periodically polls the message queue "atmRcvMsgQid" to extract the incoming cells. After getting the cells, the application task calls the **AAL_receive** to assemble the cells into the message. Finally the application task will pass the message to user handler. For the transmission, the application task calls the **AAL_send** in the AAL layer, which in turn calls **ATM_send** in the ATM layer to send a cell.

4.2.2 Control Flow

Figure 4.4 illustrate the control relationship which exists between the various modules defined in Figure 4.3 for sending/receiving an ATM cell. The figure indicates with arrows the upcalls and downcalls between layers, and the interrupt service routine. This example illustrates that the downcalls are used for the system initialization and the transmission for hybrid architecture. For the reception, the application task directly polls the message queue which is between the AAL layer and the ATM layer.

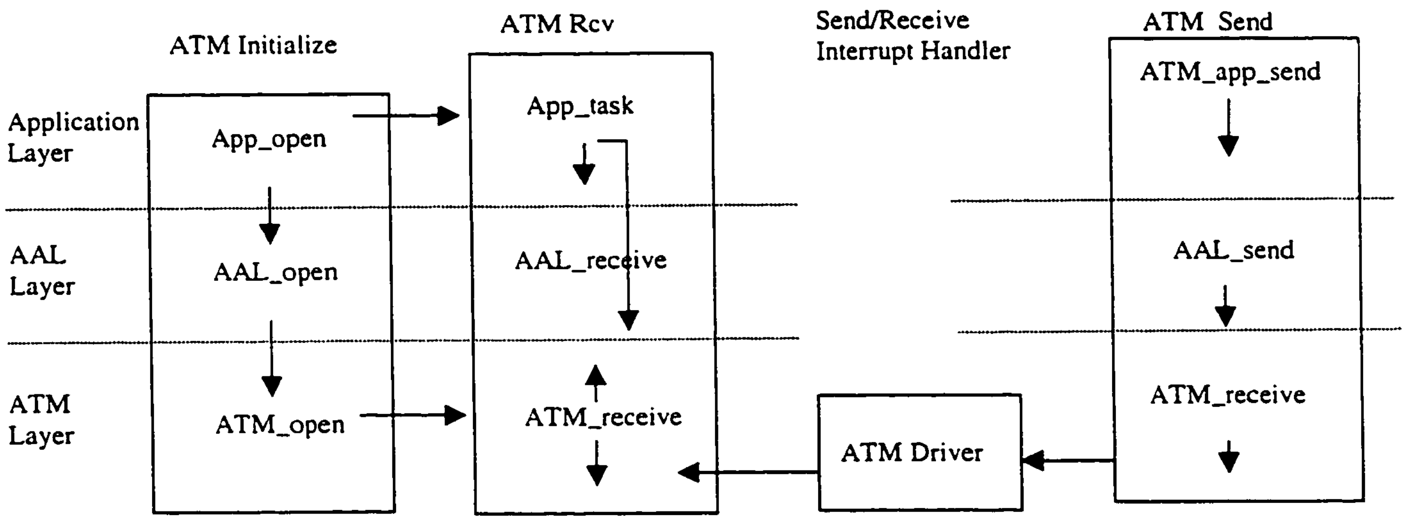


Figure 4.4 Control Relationship for sending/receiving an ATM cell

Chapter 5 Comparison between the different implementation architectures.

In this chapter we focus our attention on the comparison between procedure-based architecture and process-based architecture. For the procedure-based approach, the distinguishing feature of the upcall methodology is that flow of control upward through the layers is done by a subroutine call, which is synchronous, rather than by an interprocess signal, which is asynchronous as in the process-based approach. One obvious advantage of the synchronous flow is efficiency. First, in almost every system the subroutine call is substantially cheaper than an interprocess signal, no matter how cheap the interprocess signal becomes. In a system with many layers, the cost of messages across process boundaries can swamp the processing cost within a signal layer. However, the system overhead of interprocess signaling is not the major source of inefficiency when layer crossings are done by asynchronous signals; the more serious cost is building data buffering mechanisms to hold the information until the next layer is scheduled and runs. This buffering of information at each layer boundary, which in some systems can require copying the data itself, can easily turn out to be the dominant component of execution.

A closely related advantage of upcalls is simplicity of the implementation. Clearly, elimination of code for buffering data at layer boundaries is an important simplification. Perhaps a more interesting simplification results from the ability of one layer to "ask advice" of a layer above it. In classical layering, a lower layer performs a service without much knowledge of the way in which that service is being used by the layers above.

Excessive contamination of the lower layers with knowledge about upper layers is considered inappropriate, because it can create the upward dependency which layering is attempting to eliminate. However, as a practical matter, the lower level often substantially contorts itself to provide a service with reasonable performance for a variety of clients (the higher layers). For example, file systems often provide both a character-at-a-time interface and a block-at-a-time interface, to deal with clients with different requirements. The necessity of providing both of these interfaces, and especially for dealing with a client who changes back and forth between them as part of reading the same file, can often result in a very complicated program. In the upcall methodology, it is considered acceptable to make a subroutine call to the layer above asking it questions about the details of the service it wants. Along with the upcall, we must consider the benefits of the multi-task module. First, most programmers are more accustomed to dealing with subroutine interfaces than interprocess communication interfaces as standards. Thus, the fact that only subroutine interface are exported leads to a layer interface which is less threatening and easier to understand. Second, this methodology eliminates the temptation of architecting a systemwide codification of the format or usage of an intertask message. Different layers, in fact, have drastically different requirements for communicating between the tasks. Some communicate in terms of a work queue, others in terms of modified state variables and others in terms of requests for execution of other tasks after a certain period of time has elapsed. Hiding this variability inside the module makes dealing with each module a simpler intellectual exercise. For example, the ATM layer in Figure 2.5 dispatches a task for the incoming cells. The dispatch algorithm is contained within a single module upcalled by the interrupt handler. If the layer were redesigned to use a different task allocation technique. this change would be internal to the ATM layer rather than requiring a change to an exported interface. The knowledge of how tasks are used: like other design decisions, should be local rather global. A general characteristic of this methodology, which we consider a strong advantage, is that decisions about how tasks are used need not be made until late in the design. In the above example, the decision as to which task should be used to handle an incoming packet is not constrained in any serious way by the example programs. For example, the program could be initially written so that all incoming packets are processed by one task. This decision could be later changed if a performance bottleneck resulted from the initial design, or if a redesign were required in order to meet the reliability. In a system in which layers are realized as tasks, the deployment of tasks within the system is determined as part of the initial architecting of the system abstractions, and it becomes very difficult to rearrange tasks later, in order to deal with problems such as performance.

The distinguishing feature of process-based architecture against the procedure-based architecture is the ability to take advantage of a multiprocessor environment. In the following we summarize the main features for the process-based architecture and procedure-based architecture from the performance's and implementation's point of view.

The features of the process-based architecture are:

1. A kernel or operating system is needed to support process-to-process communication. The time to perform process-to-process communication is a time-consuming procedure. In order to improve system performance, one must reduce the number of processes (e.g., combine multiple layers into process) as much as possible. Task to

task communication in Ada was also measured in [HW89]. Table 3.4 gives the time for simple producer-consumer rendezvous where the only parameter passed is an integer and a pair of put-get operations on a mailbox on a M68020 CPU at 20 Mhz.

2. It is particularly suitable for a multiple-processor system, where each processor executes one or several layer processes.
3. In some operating systems, processes do not share memory space (protection between users). So, copies of messages between layers may be needed, which can result in a degradation of the system performance. Experience [CJ89] has shown that the time for messages copy can reach 50% of the total processing time in a communication system. This is an unacceptable overhead.

The features of the procedure-based architecture is:

1. Compared with the inter-layer communication mechanism in the process-based implementation, inter-layer communication in the procedure-based implementation has the two advantages
   - Because most languages can support the procedure call, inter-layer communication is independent of the kernel and the operating system; so the software has a good portability.
   - Procedure calls take much less time than process-to-process communication. In procedure call, parameters passing between caller and callee are performed through the stack or CPU registers with a better performance for CPU registers.
2. Because all procedures can share a common memory space, copies of messages between layers can be avoided completely. This is a very attractive feature for a real-time system

In the following we present a table (see Table 5.1) to make the comparison between the procedure-based architecture, process-based architecture and hybrid architecture based on the following criteria.

- **Inter-Layer communication**: This criterion indicates the performance of inter-layer communication. Three types of inter-layer communication are considered: procedure call, ISR to process communication and process to process communication.
- **Message copy**: This criterion indicates whether or not message copy is needed between layers.
- **Kernel or OS**: This criterion indicates whether or not a kernel or operating system is needed to support the implementation structure.
- **Wait time for user calls**: This criterion evaluates the wait time for a user call. A short wait time is desirable in a system.
- **Parallelism**: This criterion evaluates the parallelism of the system. It is important to be able to implement the layered communication software in a system with multiple processors.
- **Implementation complexity**: This criterion indicates whether or not the implementation structure can be implemented easily.

The results of comparison are shown in the Table 5.1. From the Table we see that the process-based implementation achieves maximum parallelism and a short wait time for a user call. But the inter-layer communication is slow and copies of PDUs (protocol data units) between layers may be needed.

The procedure-based implementation has a fast inter-layer communication and immediate response. Its performance is the best among the three implementation alternatives. In addition, no kernel is needed in an embedded system. But the wait time for a user call may be long.

The hybrid implementation is interesting because it has a fast immediate response and can achieve the partial parallelism. But if the copies of PDUs between layers is needed, it degrades the system performance.

Feature	Process-Based implementation	Procedure-based implementation	Hybrid implementation
Inter-layer commu.	Slow	Fast	Medium
Message copies	Yes/No	No	Yes/No
Kernel or OS	Yes	No	Yes
Wait time for user call	Short	Long	Medium
Implementation complexity	Complex	Simple	Complex
Parallelism	Yes	No	Partial

Table 5.1 Comparison between the different implementation architectures

Chapter 6 ATM Scheduling and Queuing

6.1 Introduction

In previous chapters we have studied the ATM architecture and we have shown that the architecture is a three-layer one and presented the three different approaches to implemented this architecture. The core layer in this three-layer architecture is the ATM layer which mainly performs (1) connection management (2) traffic management (3) QOS (Quality Of Service) (4) data transfer. Of these, the QOS may be the most important functionality of an ATM layer. It is the QOS that makes the ATM network different from other networks such as IP network. The QOS is a very complicated topic. Because of the limitation of space, we don't discuss the QOS in this project. The interesting reader can find the QOS papers in [BO92, MS98]

In order to achieve the required QOS, the key is to choose an appropriate scheduling and queuing discipline for an ATM switch. The scheduling and queuing discipline will have a direct effect on the ATM network performance for the following reasons:

In an ATM network, cells from different connections interact with each other at each switching node without appropriate control. These interactions may adversely affect the network performance. Figure 1.2 in Chapter 1 shows a change in the peak rate as a result of multiplexing. The scheduling disciplines of the switching nodes, which control the order in which cells are transmitted, determine how cells from different connections interact with each other.

The scheduling discipline at a switching node affects three nearly independent factors:

1. Bandwidth (which cell gets transmitted);
2. Promptness (when does a cell get transmitted);

3. Buffer space (which cells are discarded).

The above three factors in turn affect three performance parameters: throughput, delay, and loss rate.

A desirable scheduling discipline should have the following properties:

- Efficiency. To achieve certain performance guarantees, we need a connection admission control policy to limit the real-time traffic in the network. One scheduling discipline is more efficient than another if it can accept more connections under the same traffic load and meet the same end-to-end performance guarantees. An efficient scheduling discipline will eventually result in higher network utilization.
- Protection. It is essential that a scheduling discipline can provide guaranteed services for well-behavied clients even in the presence of ill-behavied users, network load fluctuation and unconstrained best-effort traffic.
- Flexibility. The ATM network needs to support applications with diverse traffic characteristics and performance requirements. The scheduling discipline should be flexible to allocate different delays, bandwidths and loss rates to different real-time connections.
- Simplicity. We would also like to have a scheduling discipline which is both conceptually simple so that it can be studied with simple analysis techniques and easy to implement in very high speed ATM networks.

In this chapter we will focus our study on the ATM scheduling and queuing.

6.1.1 Motivation

Traditional communication networks offer only a single type of service which supports one type of application. Voice, video and data communications are provided by separate networks. For example, the telephone network has been designed to support interactive voice which requires a low delay, jitter-free, low rate, two way service. The cable TV network has been designed to support analog video which requires a high rate, one way, jitter-free service. The data network has been designed to support communication between computers. However, the current data network only offers a best effort service – there are no guarantees on performance parameters such as delay or throughput. The specialization of each of these networks has allowed the network design to be optimized for a particular type of service.

ATM networks will have to support applications with multiple traffic characteristics and performance requirements. These applications can be classified into at least two classes: those that have strict performance requirements in terms of delay, delay jitter and loss rate, and those that do not. Examples of the first class are video conferencing and scientific visualization. Examples of the second class are file transfer, distributed computation and electronic mail. Corresponding to these two classes of applications, there are two classes of services to be offered by the ATM networks-real-time service and non-real-time service.

The non-real-time service corresponds to the best effort service provided by the current data network, but the real-time services provided by the ATM networks are far from the simple merging of services provided by the current voice and video networks because of the vast diversity of traffic characteristics and performance requirements of existing applications as well as the uncertainty about future applications. Real-time applications generate network traffic that requires stringent performance guarantees. These performance guarantees are generally not provided by the conventional first-come-first-serve (FCFS) scheduling discipline. In order to guarantee the real-time traffic, priority scheduling can be used. However a single ill-behaved source, sending cells to an ATM switch at a sufficiently high speed and a high priority, can capture an arbitrarily high fraction of the bandwidth of the outgoing link.

So both priority scheduling discipline and FCFS scheduling discipline do not meet the needs of the congested networks and the network with the real-time traffic as indicated in section 6.3.1. It is the primary motivation for new scheduling disciplines.

Following a similar line of reasoning, Nagel [NA85] proposed a fair queuing (FQ) algorithm in which the switches maintain separate queues for the cells from each individual source (connection). The queues are serviced in a round-robin manner. This prevents a source (connection) from arbitrarily increasing its share of the bandwidth or the delay of other sources. In fact, when a source (connection) sends cells too quickly, it merely increases the length of its own queue. Nagel's algorithm, by changing the way cells from different sources (connections) interact, does not reward, nor leave others vulnerable to, anti-social behavior. If the real-time traffic is considered in the ATM network then the weighted fair queuing should be used. The weighted fair queuing is an extension to fair queue and to apply the weights to identify the different traffic flows. In here, the weight can be seen as the priority in the priority discipline, so the weight fair queuing discipline combines the fair queuing discipline with the priority scheduling discipline.

A simple comparison for the *fair queuing* and the *weighted fair queuing* is given below. The fair queuing refers to a class of algorithms which evenly share the available bandwidth on a link among a number of sources (connections), while the weighted fair queuing algorithm can share bandwidth unevenly among sources (connections), e.g. assigning twice as much bandwidth to one source (connection) as to another.

In the next sections, we will overview some scheduling algorithms and analyze which scheduling algorithm is suitable for ATM switch.

6.1.2 Scheduling and Queuing in a ATM switch

In a ATM switch, the cells may be queued in (1) Input Port Queue (2) Connection Queue (3) Output Port Queue. In the following we will give the detail description for each of them.

1) Input Port

Each input port contains a dedicated queue which is used to store the incoming cells until the scheduling algorithm decides to serve the queue. The switching transfer medium then moves the ATM cells from the input queues to the connection-based queue (i.e. central queue described in the next section). In an ATM switch, the number of ports is small (in general, it is less than 128). The scheduling algorithm for input ports can be as simple as round-robin or can be as complex as priority, fair queuing and weighted fair queuing, or can be more complex such as taking into account the input queue filling levels as indicated in [BO92]. However, these schemes have Head of Line (HOL) blocking problem i.e. if two cells of two different ports contend for the same output, one of the cells is to be stopped and this cell blocks the other cells in the same port which are destined for a different out port. This queuing discipline can be shown by the following figure.

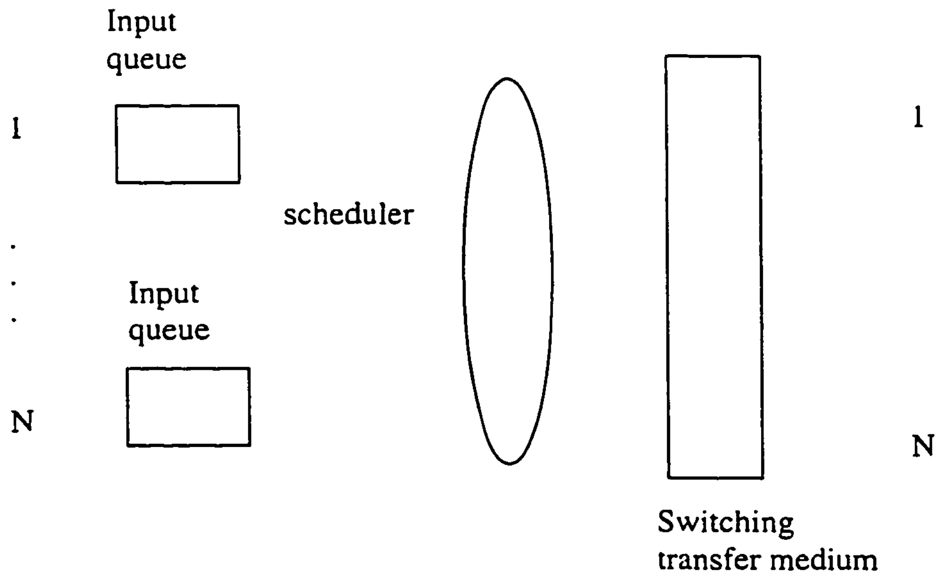


Figure 6.1 Input Port

2) Central queue (connection queue)

In this scheme, the queuing buffers are shared between all input ports and output ports. In an ATM switch, each connection has a queue. So the number of queues in an ATM switch is vary vast (in general, the ATM connections in an ATM switch may be larger than 100K). Because of the large number of connections, the scheduling and queuing must be implemented in hardware for a high performance ATM switch.

All the incoming cells are stored in the central queues based on the connections. The scheduling algorithm will make the decision as to which connection queue is serviced at a particular time. After choosing a queue to be serviced, the scheduling algorithm will move the cells in this queue to the corresponding output port. Currently, the most popular scheduling disciplines for connection queues are fair queuing and weighted fair queuing. The following figure shows this mechanism.

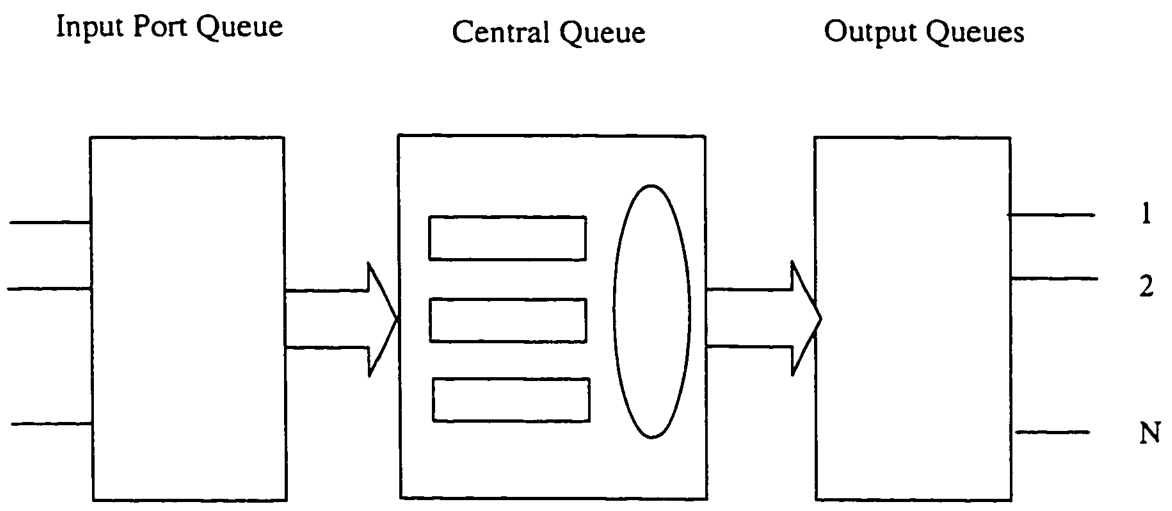


Figure 6.2 Central Queue

3) **Output Port**

In this queuing discipline, queues are located at each output port of the ATM switching. Each output port contains a dedicated queue which is used to store the outgoing cells. The output contention problem is solved by these queues. The cells arriving simultaneously at all input port destined for the same output are queued in the buffer of the output queue). In an ATM switch, the number of port is small (in general, it is less than 128). Like the input queue, the scheduling algorithm for output port can be as simple as round-robin or can be complex as priority, fair queuing and weighted fair queuing or can be more complex such as taking into account the output queue filling levels as indicated in [SV]. The following figure illustrates this mechanism.

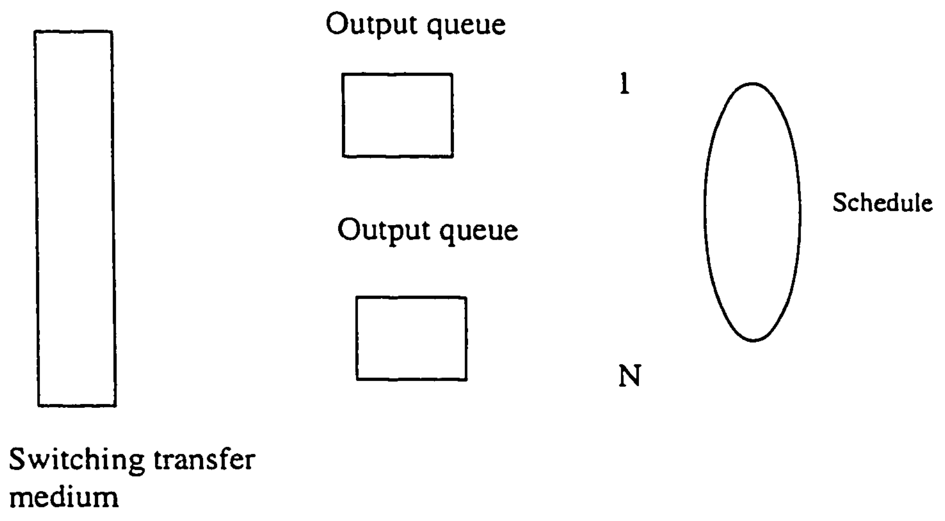


Figure 6.3 Output port

6.2 Previous work

A number of scheduling disciplines have been proposed in recent years. The benefit of traffic regulation at the switching nodes for achieving desired delay bound and throughput was analyzed in [KE91, SV, ZH90]. It was shown that an effective scheduling discipline could reduce maximum network delay. A scheduling discipline can be classified as either work-conserving or non-work-conserving. In a non-work-conserving discipline, each cell is assigned an eligibility time. No cell will be sent if none of the cells is eligible for transmission, even when the server is idle at that time.

In this chapter, we will use *flow* concept in the following sections. The notion of a flow is quite general and applies to datagram networks (e.g. IP, OSI) and Virtual Circuit networks like X.25 and ATM. In ATM, a flow could be identified by a connection, so the *flow* has the same meaning as *connection*.

In this section, we will look at the following previously proposed disciplines: Virtual Clock [ZH90], Fair Queuing [NA85], Delay Earliest-Due-Date (Delay-EDD) [MA], Jitter Earliest-Due-Date (Jitter-EDD), Stop-and-Go, Hierarchical Round Robin (HRR)[OS, CPU], and Rate-Controlled Static Priority (RCSP) [OS]. The first three belong to the work-conserving class and the rest are non-work-conserving.

(1) Virtual Clock

The basic idea of Virtual Clock was inspired by Time Division Multiplexing (TDM) systems. Each cell is assigned a virtual transmission time, which is the time at which the cell would have been transmitted if the server were actually doing TDM. Cells are transmitted in the increasing order of virtual transmission times. For example, if a connection is to get a service rate of 10 cells per second, incoming cells from that connection are stamped with virtual transmission time 0.1 second apart.

The Virtual Clock algorithm assigns each cell a virtual transmission time based on the arrival pattern and the bandwidth reservation of the connection to which the cells belong. In this way, it ensures that each well-behaving connection gets required guaranteed services. The Virtual Clock provides throughput guarantees, but no specific delay bounds.

(2) Fair queuing/weighted fair queue

The Fair Queuing emulates a bit-by-bit (a bit corresponds to a connection) round-robin service among the connections belonging to the same output link. It is a simple control strategy that provides all users with an equal allocation of network resources. The basic idea of Fair Queuing is to transmit data from each connection in turn. Each connection receives a portion of the bandwidth and the remaining bandwidth is equally distributed among all the connections. A simplified example: if N connections share the same output link, then each connection should receive 1/N of the total bandwidth. If any connection uses less than its share, the slack is equally divided among the rest. This could be achieved by doing a bit-by-bit round robin (BR) service among all the connections.

In [KS89], Demers presented a formal description of a fair queuing as shown below:

The term fair has a clear colloquial meaning, but it also has a technical definition (actually several, but only one is considered here). Consider, for example, the allocation of a single resource among N users. Assume there is an amount μ_{total} of this resource and that each of the users requests an amount ρ, and under a particular allocation, receives an amount μ_i. What is a fair allocation? The max-min fairness criterion [HA86] states that an allocation is fair if (1) no user receives more than its request, (2) no other allocation scheme satisfying condition 1 has a higher minimum allocation, and (3) condition 2 remains recursively true as we remove the minimal user and reduce the total resource accordingly, $\mu_{total} = \mu_{total} - \mu_{min}$. This condition reduces to $\mu_i = \text{MIN}(\mu_{fair}, \rho_i)$ in the simple example, with μ_{fair}, the fair share, being set so that $\mu_{total} = \sum \mu_i$. This concept of fairness easily generalizes to the multiple resource case [RA87]. Note that implicit in the max-min definition of fairness is the assumption that the users have equal rights to the resource.

To emulate bit-by-bit round robin (BR), the Fair Queuing algorithm assigns each cell a finish number, which is the round number at which the cell would have received service if the server was doing BR. By servicing cells in the increasing order of the finish numbers, it can be shown that Fair Queuing emulates BR. Different fractions of the bandwidth can be allocated to the connections by assigning each connection a weight.

The larger the weight, the more the connection receives the services. As with Virtual Clock, Fair Queuing provides only throughput guarantees but no delay bounds.

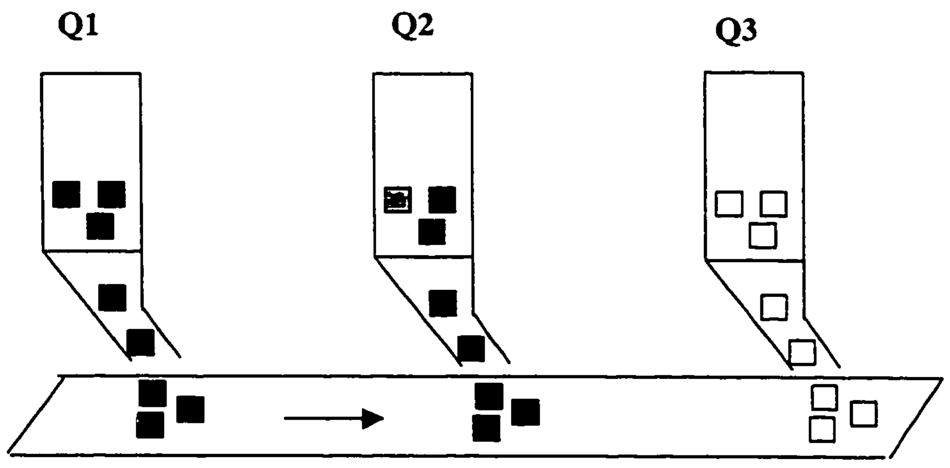


Figure 6.4. Fair Queue

An extension to fair queue is to apply the weights to identify the different traffic flows. In here, the weight can be seen as the priority in the priority discipline, so the weight fair queuing discipline combines the fair queuing discipline with the priority scheduling discipline. It can not only guarantee the bandwidth needs of the critical traffic, but also can avoid the starvation problem caused by priority discipline (i.e., it allocates some bandwidth to the low priority flow while guaranteeing the bandwidth need of critical traffic). So the weighted fair queuing has been widely used in router and switches. The weighted fair queuing discipline can be shown in Figure 6.5:

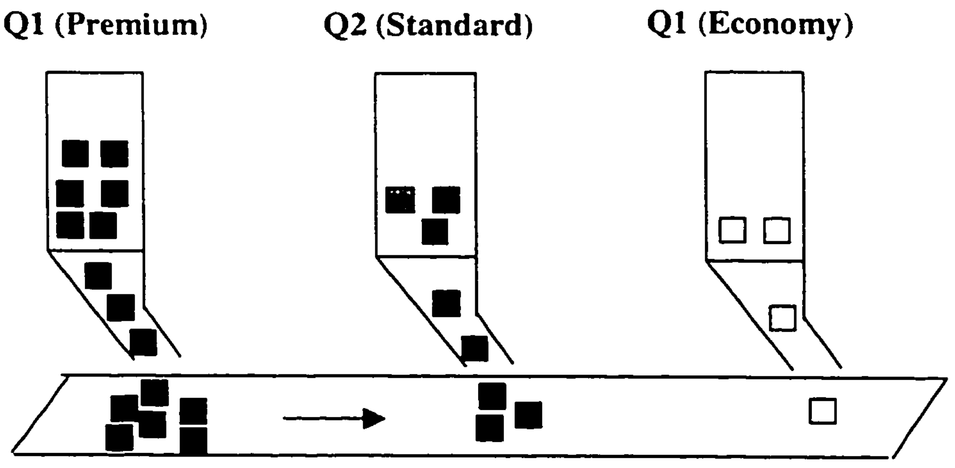


Figure 6.5 Weight Fair Queue

In fact, the fair queue is a special case of the weight fair queue where the weight of each queue is equal.

(3) Delay Earliest-Due-Date

Delay Earliest-Due-Date (Delay-EDD) algorithm extends the idea of the classical Earliest-Due-Date-First (EDD or EDF) scheduling. EDD assigns a deadline to each cell of a periodic traffic stream. The deadline of a cell is the sum of its arrival time and the period of the traffic stream.

In the Delay-EDD algorithm, a deadline is assigned to each cell according to a service contract negotiated between the server and each traffic source. The service contract states a source's promised traffic specification (such as peak and average sending rates) and the guaranteed delay bound provided by the server if the source keeps its promise. By reserving bandwidth for the peak rate, Delay-EDD guarantees a delay bound. The key point of the assignment of deadlines is that the deadline of a cell is the sum of the expected arrival time according to the service contract and the delay bound at the server.

For example, if a source promises that its sending rate is no more than 10 cells per second, and the delay bound at a server is 1 second, then the kth cell from that source will get a deadline at $0.1k+1$.

Unlike Virtual Clock and Fair Queuing, which can give throughput guarantees, Delay-Edd provides explicit end-to-end delay bounds.

(4) Jitter Earliest-Due-Date

Jitter Earliest-Due-Date (Jitter-EDD) algorithm is an extension to Delay-EDD. Jitter-EDD provides bounds on both the minimum and the maximum delays, which are usually called delay-jitter bounds.

Right before a cell is transmitted from each switching node, it is stamped with a *Holding Time* which is the difference between its assigned deadline and actual transmission time. A regulator at the next switching node will hold that cell for the period of the *Holding Time* computed at the previous node before this cell is made eligible for transmission. Since a cell will thus experience a constant delay at each switching node except the last node, Jitter-EDD can achieve a delay-jitter bound.

(5) Stop-and -Go

Stop-and-Go queuing makes use of cell frames. The underlying idea of the framing strategy is to confine cells within certain time frames, which are defined as consecutive time intervals of some fixed duration T and are viewed as traveling with the cells from one end of the link to the other end. The duration T is typically a few milliseconds. At the broadband transmission speed, several hundred ATM cells can fit into a frame of this size.

The central idea of Stop-and-Go scheme is to ensure that cells received during an arriving frame on the incoming link, for subsequent transmission over the outgoing link, will be sent exclusively during the next outgoing frame, and not any other time intervals. If a cell fails to find an empty slot in that time frame, it will be discarded rather than wait for an available slot in the future.

Ensuring that at each switching code cells are switched from an arriving frame exclusively into the adjacent departing frame of the corresponding outgoing link, Stop-and-Go keeps cells on the same frame at the source stay in the same frame throughout the network. It maintains the original smoothness of the traffic by preventing cells of different frames from clustering together as they travel in the network. Stop-and-Go can achieve a desirable bound on the end-to-end delay and the delay-jitter of each cell by proper choice of the frame size *T*. for example, for a frame size of 0.5 milliseconds, the delay-jitter of any cell will be limited to from –0.5 to +0.5 milliseconds, and the maximum delay will be limited to 1 millisecond at each switching node.

The framing strategy introduces the problem of coupling between delay bound and bandwidth allocation granularity, because the delay and delay-jitter bounds are linked to the length of the frame time. A low delay bound and fine granularity cannot be achieved at the same time in a framing strategy like Stop-and-Go. A multiple frame size version of Stop-and-Go has also been proposed.

(6) Hierarchical Round-Robin

Hierarchical Round-Robin (HRR) also adopts the multi-level framing strategy. At each level round-robin services are provided to a fixed number of slots. The time a server takes to service all the slots at a level is called the frame time at that level. A slot can either be allocated to a connection or a low level frame. If it is assigned to a connection, that connection will be served. If it is assigned to a low level frame, one slot of that low level frame will be served. If a slot is assigned to a connection with no cell waiting at that slot time when it is served, it will keep the server idle rather than giving the service time to other slots. In this way, HRR has the ability to give each level a constant share of the bandwidth. The frame time at a higher level is smaller than the frame time at a lower level, which means higher levels get more bandwidth than lower levels.

The Hierarchical Round-Robin (HRR) is non-working-conserving, because even if it serves a slot which is assigned to a connection with no cell waiting at that slot time, it will keep the server idle rather than serve the other slots. HRR can provide maximum delay bound but no delay-jitter bound.

(7) Rate-Controlled Static Priority

A Rate-Controlled Static Priority (RCSP) server consists of two components: a rate controller and a static-priority scheduler. At each switching node, a rate-controller conceptually consists of a set of regulators corresponding to the connections going through the switching node. The regulators for each connection shapes the input traffic into the desired traffic pattern by assigning an eligibility time to each cell upon its arrival and holding that cell till the eligibility time before handing it to the scheduler. In this way the regulators control the interaction between switches and eliminate jitter. Two types of regulators were proposed in:

- *Rate-jitter controlling regulator* (RCSP/RJ), which partially reconstructs the traffic pattern.

- *Delay-jitter controlling regulator* (RCSP/DJ), which fully reconstructs the traffic pattern.

The scheduler orders the transmission of eligible cells from all the connections going through the switch. It consists of a number of prioritized real-time queues and a non-real-time queue. A connection is assigned to a particular priority level during the connection establishment phase. All the cells from the same connection, after being released from their regulators, will be inserted into the real-time queue at the priority level to which that connection is assigned. Different connections can share the same priority level. The RCSP scheduler adopts a non-preemptive strategy, always selecting the cell at the head of the highest priority queue which is not empty.

For each priority level in the RCSP scheduler, there is a corresponding delay bound. By limiting the number of connections at each priority level using the admission control conditions, the waiting time for each cell at a priority level can be controlled within the delay bound associated with that priority level. With a delay-jitter controlling regulator, RCSP provides an end-to-end delay bound and delay-jitter bound. With a rate-jitter controlling regulator, RCSP can provide an end-to-end delay bound and rate-jitter bound. The rate-jitter is defined to be the maximum number of packets in a *jitter averaging interval.*

6.3 ATM Scheduling and Queuing

6.3.1 Problem

A lot of routers and switches use first-come-first-serve (FCFS) service on output links. In FCFS, the order of arrival completely determines the allocation of packets to output queue buffers. The presumption is that congestion control is implemented by the sources.

In feedback schemes for congestion control, connections are supposed to reduce the rate at which they send when they sense congestion. However, a rogue flow can keep increasing its share of the bandwidth and cause other (well-behaved) flows to reduce their share. With FCFS queuing, if a rogue connection sends packets at a high rate, it can capture an arbitrary fraction of the outgoing bandwidth. First Come First Serve (FCFS, or FIFO) traffic sequence does not meet the needs of congested networks. The earliest widely known case in point was the 1988 congestive collapse of the 56 KBPS NSFNET Backbone, which resulted from a standing load of only 47% utilization (as reported by Dr. David Mills in a talk to the Monterey InterOp in 1987). As the Figure 6.6 shows, congestion brings with it increasing mean queue depth, mean latency, and mean probability of congestive discard. Latency also becomes more variable, as queue depths instantaneously vary between empty and the upper bound. There are three possible solutions to the problem: increase bandwidth, signal senders to slow down (congestion avoidance), or manage the queue. Historically, most vendors have used manually programmed access lists to prioritize traffic; more recently, many vendors have worked on the basis of distributing bandwidth among the applications (for example, voice, data, video, ...). Either approach, however, is error-prone - the network manager cannot identify and prioritize network traffic in real time, and what he has programmed may not be relevant to (or, worse, may exacerbate) ambient network problems.

Consideration of the behavior of congested systems is not simple and cannot be dealt with in a simplistic fashion, as traffic rates do not simply rise to a level, stay there a while, and subside; in the words of Paxson and Floyd [FL94]:

"[FL1] examines the bursitness of data traffic over a wide range of time scales, and discusses the impact of this burstiness for network congestion. Their conclusions are that congested periods can be quite long, with losses that are heavily concentrated; that, in contrast to Poisson traffic models, linear increases in buffers size do not result in large decreases in packet drop rates; and that a slight increase in the number of active connections can result in a large increase in the packet loss rate. They suggest that, because the level of busy period traffic is not predictable, it would be difficult to efficiently size networks to reduce congestion adequately. They observe that, in contrast to Poisson models, in reality traffic 'spikes' which cause actual losses ride on longer-term 'ripples,' that in turn ride on still longer-term 'swells.' They suggest that a filtered variable can be used to detect the low-frequency component of congestion, giving some warning before packet losses become significant."

What is needed is a mechanism, however crude, which will identify and manage network traffic latency in real time without human intervention, prioritizing traffic that requires low latency and protecting it from other traffic streams that would otherwise crowd it out, and forcing high bandwidth sessions to share the remaining bandwidth reasonably. Fair Queuing, first proposed by Nagel [NA85] in 1985, and considered in further depth by Shenker, Keshav, Clark [KS89], and Zhang [ZH90], proposes to do exactly this. As extended to real time traffic (by applying priority or weights to identified traffic) in [SZ92], it further addresses the specific issues of latency sensitive real time traffic. Parekh and Gallagher, in [GA93], demonstrate that to construct a network with predictable end to end latency, it is necessary to implement a Fair Queuing algorithm in the switches and use an algorithm such as sliding window or token bucket to limit the amount of traffic that any given session keeps "in flight."

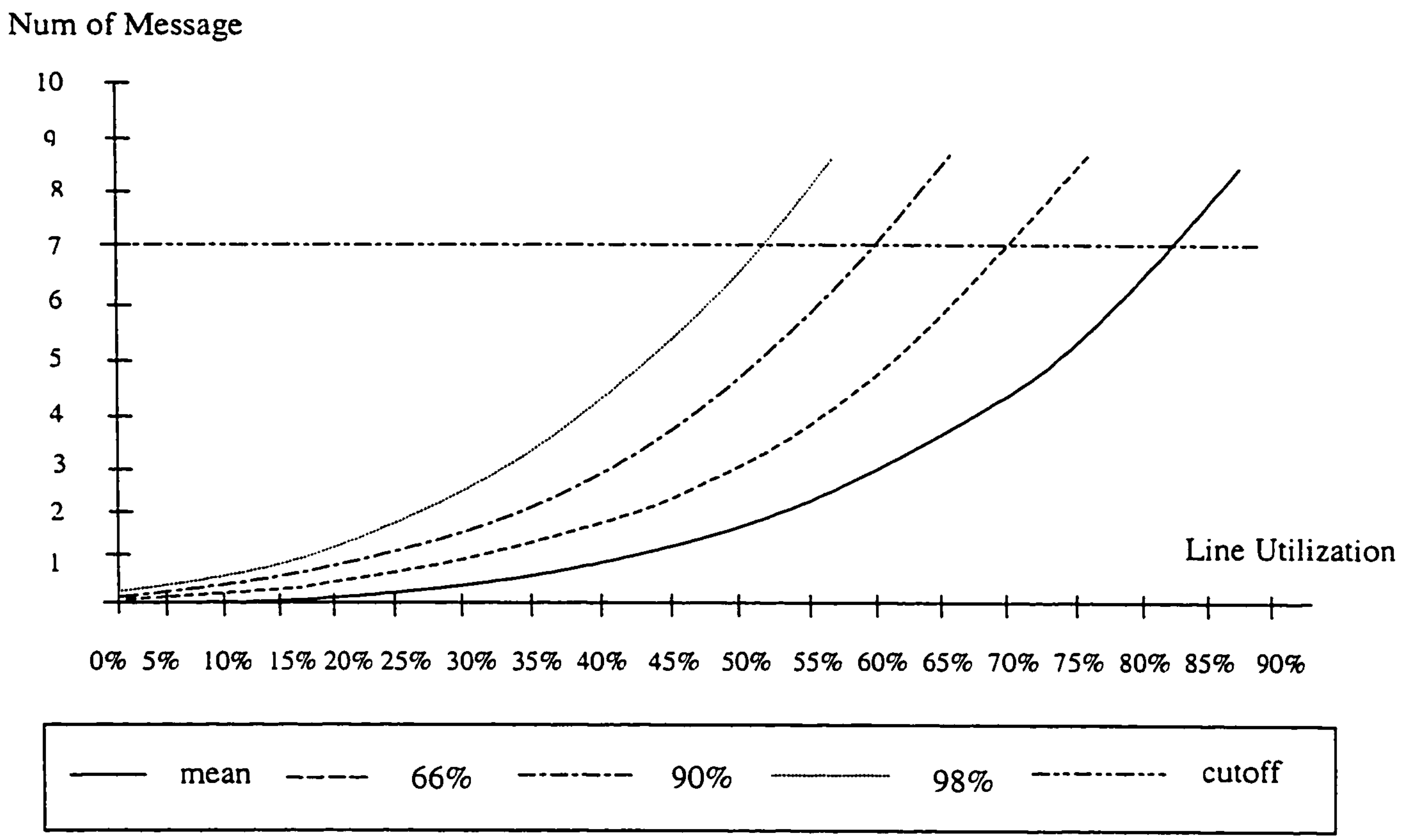


Figure 6.6 Mean Queue Depth on Enqueue (M/M/1)

6.3.2 Scheduler for ATM switch

The scheduling discipline at a network switch determines the order of cell transmission, and thus controls the variable delays of cells. In an ATM network, cells from a particular connection traverse the network on a fixed path of switches and links. Each switch has the three scheduler to schedule the input port queues, connection queues and output port queues. Since the cell scheduler can transmit only one cell at a time, it selects the next cell to transmit from a number of queues containing all cells eligible for transmission. Considerable research efforts have resulted in the development of various schedulers for ATM switch [LDW, SZ92, GA93]. The selection of a particular scheduling discipline for a cell scheduler involves a tradeoff between the need to support a large number of connections with diverse delay requirements and the need for a simple and efficient scheduling algorithm. In the following, we will present and analyze the scheduling algorithms which can be used in the ATM network

- the Static-Priority scheduler, which provides a good balance between simplicity or implementation and flexibility in allocating delay bounds and also achieves reasonably high average utilization for real-time traffic. A Static-Priority scheduler distinguishes P priority levels and maintains one FIFO (first-in-first-out) queue for each priority. Each connection is assigned an integer priority p with $1 \leq p \leq P$, and cells arriving on a connection are inserted into the FIFO queue for this connection's priority. After completing the transmission of a cell, the Static-Priority scheduler always selects the first cell in the non-empty FIFO queue with the highest priority for transmission.
- FCFS scheduler -- As for implementation simplicity, FCFS (first-come-first-serve) is by far the simplest one. That is the reason why most switches today implement the FCFS discipline. But the FCFS scheduler can only have one delay bound for all connections, and it is unlikely to be acceptable for future networks since different applications have diverse performance requirements.

- The EDF (earliest-deadline-first) scheduler, which always selects the cell with the earliest deadline for transmission, supports different delay bounds for different connections. The EDF scheduling is complex since it involves a search operation for the cell with the earliest deadline. It has yet to be demonstrated whether EDF, or other schedulers based on sorted queues can be implemented at a very high speed.
- Fair Queue/Weight Fair queuing – As for implementation, the fair queuing and weighted fair queuing are relatively complex, but they can provide the fairness which is a very important feature for a congested network. Especially, the weighted fair queuing not only can guarantee the bandwidth needs of the critical traffic, but also can avoid the starvation problem caused by priority discipline. This is the main reason why the weighted fair queuing has been widely accepted as a very efficient and strong scheduling discipline in router and switches

6.3.3 FCFS Queuing

One uses a single queue for cells from each input/output port waiting for the link to transmit them. This scheme is simple, efficient, and offers optimal average delay per cell (since it always utilizes the entire link bandwidth). Its drawback is that it does not distinguish among the different traffic streams-the more traffic in a stream, the larger its share of the link bandwidth. Furthermore, it has a large cell delay variance per stream-a larger stream's packets at the front of the queue delay a smaller stream's packets. Simply, this scheme does not prevent larger traffic streams trampling over smaller ones.

The dispatcher (scheduler) always picks the first one. This method does not emphasizes throughput, since long flow are allowed to monopolize the link bandwidth. For the same reason, the response time with FCFS can be high (with respect to transmission time. In summary, FCFS has the following features:

- FCFS is the simplest CPU scheduling algorithm.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- The average waiting time is often long.
- Simplest Algorithm, widely used in Internet Routers
- Scheduling is done using first-in first-out discipline
- All flows are fed into the same queue
- flows can interfere with each other
- is NOT fair (malicious monopolization, packet size bias)

FCFS implementation

The FCFS discipline implementations based on the following three functions :

```
struct  queue_type {
        struct  queue_type   *next;
        void                         *data;
};
struct  header_type {
struct  queue_type   *Head;
struct  queue_type   *Tail;
int      Count;
int      Maximum;
};
struct  header_type  Header;
```

1. Enqueue

```
/* Add an item to the end of a queue. */
void  enqueue  (queue_type  *new_item)
{
     struct  queue_type  *tmp;
      /* Determine whether a queue is full */
      if (queuefull (Header.Head) == TRUE)
          /* Queue is full */
           return;
       else
      {
          /* Queue is not full, add an item to the end */
          /* of queue */
          tmp = Header.Head;
          /* Find the end of queue */
          while (tmp->next != NIL )
                  tmp = tmp -> next;
          }
         /* Add an item to the end of queue */
           new_item -> next = NIL;
           tmp -> next = new_item;
     }
 }
```

2. Dequeue

```
/* Remove the first item from the beginning of a queue */
void  dequeue (queue_type  *first_item)
{
     /* Determine whether a queue is empty */
     if (queueempty (Header.Head) == TRUE)
        /* Queue is empty */
        return;
     else
       {
          /* Queue is not empty, remove the first item */
          /* from the beginning of queue */
          first_item = Header.Head;
          Header.Head = Header.Head -> next;
      }
}
```

3. IsEmpty

```
/* Determine whether a queue is empty. */
#define QUEUEEMPTY          TRUE
#define QUEUENOTEMPTY       FALSE
boolean queueempty (queue_type  *queue)
{
    if (queue == NIL)
       return  QUEUEEMPTY;
    else
       return  QUEUENOTEMPTY;
}


4.  IsFull
/* Determine whether a queue is full. */
#define QUEUEFULL         TRUE
#define QUEUENOTFULL      FALSE
boolean queuefull (queue_type  *queue)
{
    if (queue < Header.Maximum)
       return  QUEUENOTFULL;
    else
       return  QUEUEFULL;
}
```

6.3.4 Static Priority scheduling implementation

The Static-Priority scheduler can be implemented with a fixed number of FCFS queues, i.e. one FCFS queue for each priority level, or as a linked list providing a variable number of queues. Insertion and deletion operations in Static-Priority scheduler can both be accomplished by a constant number of steps, and the complexity of scheduling is very low. However, at most one delay bound can be associated with each priority level, thus limiting the flexibility of Static-Priority scheduler for providing different delay bounds to connections. Due to its simplicity, which enables cell scheduling at very high data rates, Static-Priority scheduler is suitable for bounded delay services. A danger of priority scheduling is starvation, in which flow or connection with lower priorities are not given the opportunity to be serviced. In order to avoid starvation, in preemptive scheduling, the priority of a flow or connection is gradually reduced while it is being serviced. Eventually, the priority of the flow or connection being serviced will no longer be the highest, and the next flow will start to be serviced. This method is called aging. The Priority Queuing technique is designed to give all mission-critical flows higher priority than less critical flows.

In summary, the priority scheduling is a multi-queuing mode where two or more queues exist, each receiving a user-specified share of the link bandwidth. However unlike fair queuing , one queue gets priority over all others. Priority queues allow the ATM switch to minimize the packet-delay variance for delay sensitive traffic, such as live voice and video. Another advantage of priority queues is that they are more resilient to congestion than their non-priority counterparts. This is because during system-wide congestion, where too many links have transmission traffic even though each of the link is within its bandwidth, we gives preference to the priority-queue packets at the expense of the non-priority queue packets.

(1) Static Priority Scheduling Implementation

There are the two methods to implement the static priority scheduling. One is to use a linked list, in which the cells are sorted in the list according to their priorities. The other is to use the multipl- queues such that, the cells in a queue has the same priority.

1) Linked list

In this implementation, one uses a single sorted queue for packets/cells from all traffic flows waiting for the link to transmit them in the order of their arrival as shown in Figure 6.7.

This scheme is simple and efficient when the traffic is light. If the network occurs the congestion, the linked list may become very large. In this case, maintaining a sorted list may be very expensive. For example, inserting an element into the linked list with the large number of elements may be a timing-consume procedure. In order to solve this problem, the multiple FCFS queues are used for implementing the static priority scheduling discipline. In this section, we present the implementation of the linked list. In the next section we will present the implementation of multiple FCFS queues.

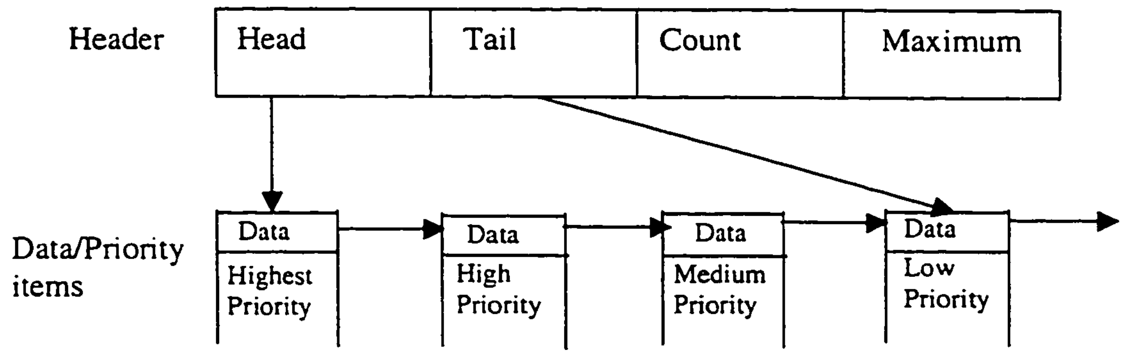


Figure 6.7 Static priority scheduling implementation with single queue

```
 struct  queue_type {
        struct queue_type   *next;
        void                *data;
        int                 priority;
};
struct header_type {
struct  queue_type   *Head;
struct  queue_type   *Tail;
int      Count;
int      Maximum;
};
struct header_type  Header;


(1)  Enqueue
/* Add an item to a  priority queue */
Void  enqueue (queue_type  *new_item)
{
     struct  queue_type   *tmp;
     tmp = Header.Head;
      /* Find appropriate place to insert */
      for (;;) {
            if ((new_item->priority < tmp->priority)
               && (tmp->next != NIL))
               tmp = tmp -> next;
            else
               break;
       }
      if (tmp->next == NIL)
        Header.Tail = new_item;
      new_item -> next = tmp -> next;
      tmp -> next = new_item;
}


(2) Dequeue
/* Remove an item from a priority queue */
void  dequeue (queue_type  *highest_priority_item)
{
   /* Remove the highest priority item from a queue */
   highest_priority_item = Header.Head;
   Header.Head = Header.Head -> next;
}
```

2) Multiple FIFOs

The Static-Priority scheduler can be implemented with a fixed number of FCFS queues, i.e. one FCFS queue for each priority level as shown in Figure 6.8.

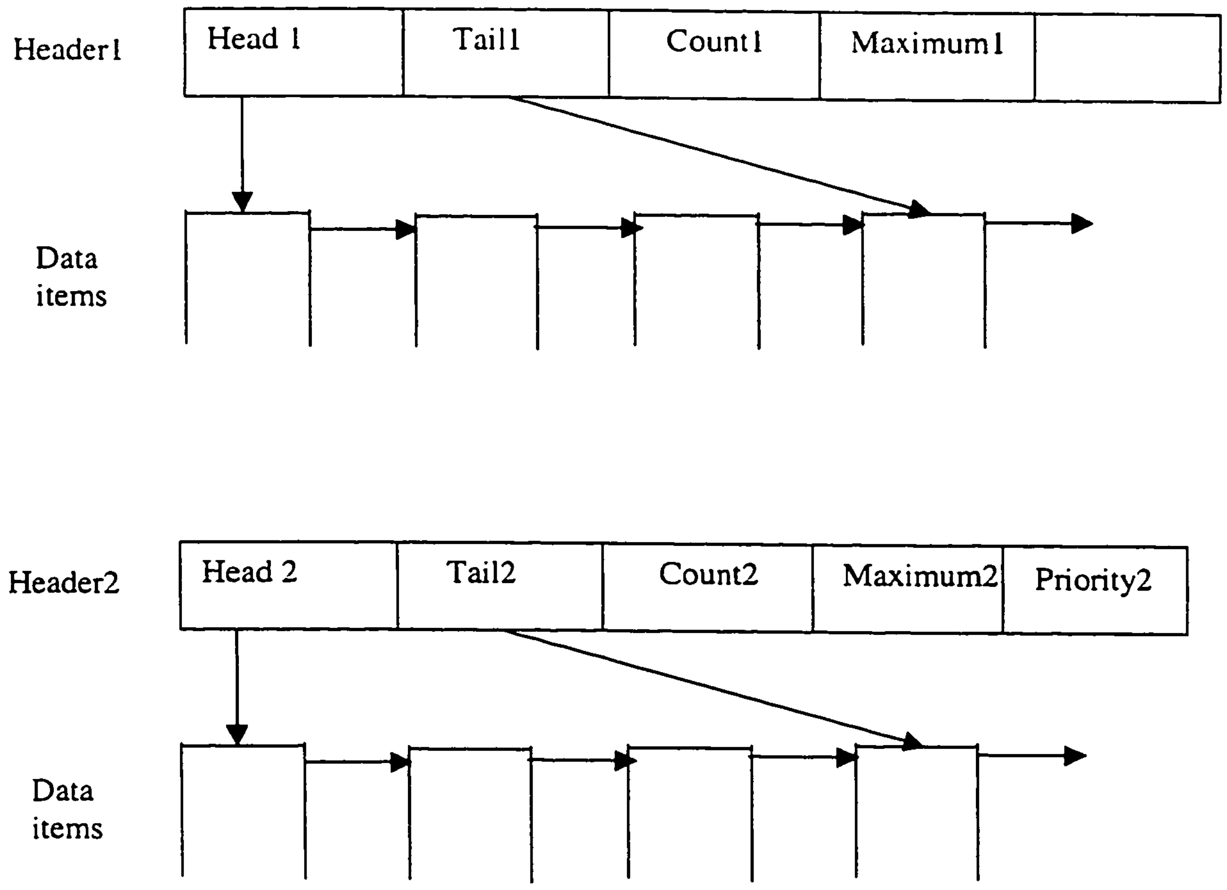


Figure 6.8 Static priority scheduling implementation with multiple queues

```
struct  queue_type {
        struct queue_type   *next;
         void                *data;
         int                 priority;
};
struct header_type {
struct  queue_type   *Head;
struct  queue_type   *Tail;
```

```
int       Count;
int       Maximum;
int       Priority;
};
struct header_type  HeaderArray[1..NumOfPriQ];


(1)  Enqueue
/* Find the corresponding priority queue and */
/* add an item to the end of the queue. */
void  enqueue  (queue_type  *new_item)
{
      struct  queue_type  *tmp;
      /* Find the corresponding priority queue */
      for (i = 1; i < NumOfPriQ;  i++)
      {
           if (HeaderArray.Priority == new_item-> priority)
           {
                   /* Determine whether a queue is full */
                   if (queuefull (Header.Head) == TRUE)
                   /* Queue is full */
                          return;
             }
             else
             {
                    /* Queue is not full, add an item to */
                    /* the end of queue */
                    tmp = Header.Head;
                    /* Find the end of queue */
                    while (tmp->next != NIL )
                       tmp = tmp -> next;
               }
               /* Add an item to the end of queue */
               new_item -> next = NIL;
               tmp -> next = new_item;
     }
   }
}
```

(2) Dequeue

```
/* Find the corresponding priority queue and */
/* Remove the first item from the beginning of a queue */
void  dequeue (queue_type  *first_item)
{
      /* Find the corresponding priority queue */
      for (i = 1; i < NumOfPriO;  i++)
      {
            if (HeaderArray.Priority == new_item->priority)
           {
```

```
                        /* Determine whether a queue is empty */
                        if (queueempty (Header.Head) == TRUE)
                            /* Queue is empty */
                            return;
                          else
                          {
                              /* Queue is not empty, remove */
                              /* the first item from the */
                              /* beginning of queue */
                              first_item = Header.Head;
                              Header.Head = Header.Head -> next;
                          }
                        }
                    }
}
```

6.3.5 EDF Queue

The Earliest-First scheduling discipline has received a lot of attention in the past. For example, Georgiadis, Guerin and Parekh [GGP97] and Liebeherr, Wrege and Ferrari [LWF96] showed that, for a single node, EDF is worst-case optimal in the sense that if any scheduling discipline can meet a set of delay bounds with certainty, then EDF can meet these delay bounds with certainty. These papers also present necessary and sufficient conditions under which a set of delay bounds can be met. Low-complexity schemes that approximate EDF are described by Liebeherr et al. in [LWF96, WL97].

For the ATM network, The EDF (earliest-deadline-first) scheduler always selects the cell with the earliest deadline for transmission. It supports different delay bounds for different connections. The EDF scheduling is complex since it involves a search operation for the cell with the earliest deadline. It has yet to be demonstrated whether EDF, or other schedulers based on sorted queues can be implemented at a very high speed.

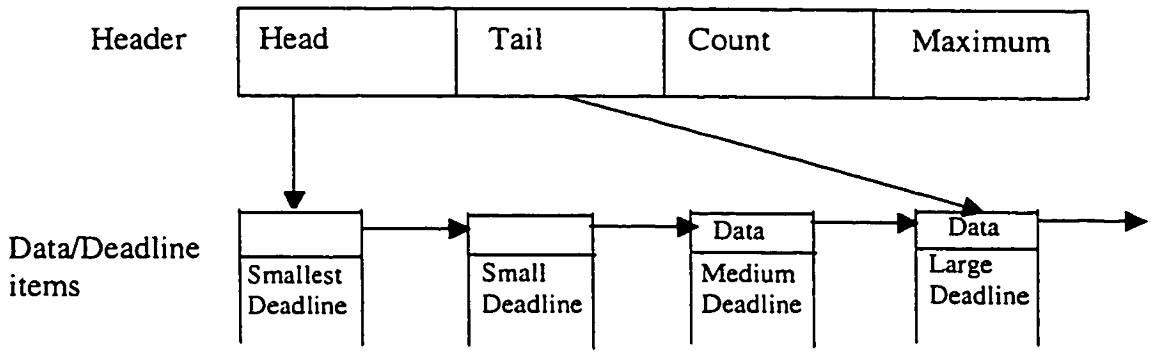


Figure 6.9 EDF queue implementation

The following codes describe EDF queue implementation. The `queue_type` defines the structure of EDF queue which includes the pointer to the next node, data and deadline. The `header_type` defines the header and tail for EDQ queue.

```
struct  queue_type {
        struct queue_type   *next;
        void                 *data;
        float                deadline;
};
struct header_type {
struct  queue_type   *Head;
struct  queue_type   *Tail;
int       Count;
int       Maximum;
};
struct header_type  Header;
/* last_time is the time when last enqueue is done */
float  current_time, last_time, time_difference;
```

(1) Enqueue

```
/* Add an item to a  EDF (sorted) queue */
Void  enqueue (queue_type  *new_item)
{
     struct  queue_type   *tmp;
     /* adjust the deadline */
      adjustdeadline (Header);
      tmp = Header.Head;
      /* Find appropriate place to insert */
      for (;;) {
           if ((new_item->deadline > tmp->deadline)
              && (tmp->next != NIL))
              tmp = tmp -> next;
           else
              break;
   }
  if (tmp->next == NIL)
```

```
        /* The queue has reach the tail, so the new item */
        /* is the last item */
         Header.Tail = new_item;
     /* insert the new item after the item pointed by tmp */
     new_item -> next = tmp -> next;
     tmp -> next = new_item;
}
```

(2) Dequeue

```
/* Remove an item from a priority (sorted) queue */
void  dequeue (queue_type  *earliest_deadline_item)
{
        /* Remove the item from the sorted queue */
         earliest_deadline_item = Header.Head;
         Header.Head = Header.Head -> next;
}
```

(3) AdjustDeadline

```
/* To adjust the deadline */
void  adjustdeadline (struct header_type  queHeader)
{
       /* get current time, it is a system call */
       current_time = get_time ();
       /* get the time period between the current enqueue */
```

```
    /* call and the last enqueue call */
    time_difference = current_time - last_time;
    tmp = queHeader.Head;
    /* adjust the deadline of all items in the queue */
     for (i = 1; i < NumOfQueue; i++)
     {
          tmp->deadline = tmp->deadline - time_difference;
          if (tmp -> deadline < 0)
            tmp -> deadline = 0;
     }
     last_time = current_time;
}
```

From the above EDF implementation, we can see that it is very similar to the static priority implementation. But there are the two important differences between them

- EDF is a dynamic scheduling algorithm, while the priority scheduling is a static scheduling algorithm. So, the deadline of each item in the sorted queue must be adjusted dynamically
- EDF must be implemented as a sorted queue. As indicated in the above, this can make the insert operation very slow when there are many items in the sorted queue. While the priority scheduling can be implemented as the multiple-FIFO queues. Greater efficiency can be achieved by using multiple FIFO queue

6.3.6 Fair Queue

One solution to service traffic with fairness is to divide the link bandwidth among the different streams, which each queue stream's throughput limited to its fair share - a rate smaller than the link bandwidth. This way the delay variance of packets of a particular stream is minimized as the traffic on other streams cannot block its packets. The drawback in this scheme is the reduced link utilization as the link bandwidth is wasted when a queue has nothing to send. It also means that this scheme has an average delay per packet larger than the single-queue scheme as no queue receives the full link bandwidth.

To achieve fairness while fully utilizing the link bandwidth, one allows multiple queues each receiving its fair share of the link bandwidth divided among all non-empty queues. This way, we do not waste any bandwidth associated with a non-empty queue. Rather, we divide the unused bandwidth to the queues with packets to send. Hence, this scheme has the same average delay per packet as the single-queue scheme with the advantage of fairness.

So we see that the *Fair Queuing* is a multi-queue mode where two or more queues exist. Each queue receives a share of the link bandwidth that the user specifies. No queue gets priority over another.

The actual throughput of a fair queue may be higher or lower than the user specification. It may be higher when one or more of the other queues on the link are idle; it may be lower during high-traffic conditions when the router cannot serve all links at their maximum speeds.

A change to the bandwidth of a fair queue takes effect after the queue gets idle, until such time the queue continues to receive the old rate. This is due to the software limitations.

The user may add or remove a fair queue at any time with immediate effect. When adding a fair queue, the existing packets/cells belonging to the new queue that are waiting in other queues continue to remain there while the new packets/cells use the newly-created queue. When removing a fair queue, the existing packets waiting in the queue to be removed continue to wait there until they dequeue, while the new packets utilize the default queue of the link. The fair queue to be removed remains active on the link until the system finishes processing all the packets in the queue. The system finally removes the queue when it is empty.

In this project, we use a modified bit-by-bit round robin (MBR) to implement the fair queuing implementation. The ideal bit-by-bit round-robin (BR) which solves the flow is presented by Nagle [NA85]. In the BR scheme, each flow gets to send one bit at a time in round robin fashion. In order to implement the BR scheme, a very precious clock is needed, it is impractical to implement such a scheme in the current system. They suggest approximately simulating BR. Also, they calculate the time when a packet would have left the router using the BR algorithm. The packet is then inserted into a queue of packets sorted on departure times.

Our modified algorithm (MBR) assumes that the basic time unit is a cell tick (time to service a cell). This assumption is acceptable because (1) ATM cell is a fixed length, so the service time is a constant (2) ATM cell's length is short (53 bytes) (3) In most of the ATM switch, the cell tick is a basic service unit for scheduling discipline. With this assumption, we can simplify the fair queuing implementation (i.e. use a cell tick timer to replace a virtual clock).

A bit map is used in which a bit corresponds a queue. Each time, the scheduler only services a cell for one queue. The data structure for the fair queuing is shown as in Figure 6.10:

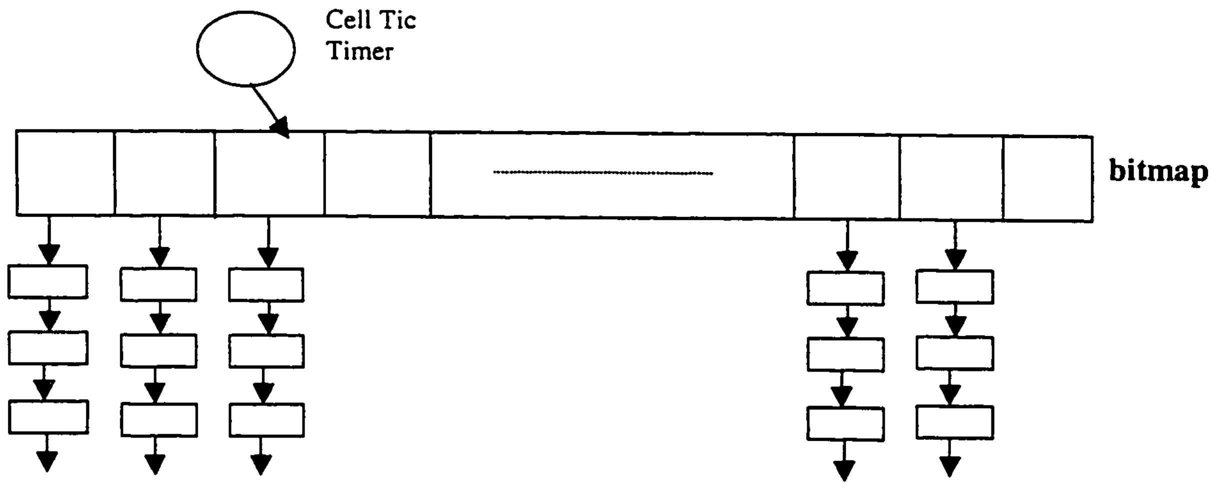


Figure 6.10 Fair queue implementation with a virtual clock

Fair queuing implementation:

```
/*NUM_OF_BITS can be divided by 8 */
#define  NUM_OF_BITS  128
char  bitmap[NUM_OF_BITS / 8];
struct  queue_type {
        struct queue_type   *next;
        void                 *data;
```

```
};

struct header_type {

struct  queue_type   *Head;

struct  queue_type   *Tail;

int      Count;

int      Maximum;

};

struct header_type  HeaderArray[NUM_OF_BITS];


(1)  Enqueue

/* Add an item to a queue */

void  enqueue (queue_type  *new_item,  int  index)

{

     struct  queue_type  *tmp;

     /* Determine whether the queue is full */

      if (queuefull (HeaderArray[index].Head) == TRUE)

           /* Queue is full */

            return;

      else

      {

           /* Queue is not full, add an item to the end */

           /* of queue */

           tmp = HeaderArray[index].Head;

           /* Find the end of queue */
```

```
            while (tmp->next != NIL )
                    tmp = tmp -> next;
            }
          /* Add an item to the end of queue */
             new_item -> next = NIL;
             tmp -> next = new_item;
      }
      /* Set 1 for corresponding bit in bitmap */
      setBitMap(bitmap, index, TRUE);
}
```

(2) Dequeue

```
/* Remove an item from a queue */
void  dequeue (queue_type  *new_item,  int  index)
{
      struct  queue_type  *tmp;
      tmp = HeaderArray[index].Head;
      new_item = tmp->next;
      HeaderArray[index].Head->next;
}
```

(3) Set/clear bitmap

```
/* set or clear one bit in the bitmap */
void setBitMap (char* bitmap, int index, boolean set_clear)
```

```
{
Uint32  *myBitMap;
Uint32  set_clear_bit = 0;
/* We suppose that the max number of bit in the bitmap */
/* is 32*8 = 256 */
MyBitMap = (Uint32 *)bitmap;
If (set_clear == TRUE)
{
  /* set one bit */
  set_clear_bit = index << 1
  *MyBitMap = *MyBitMap | set_clear_bit;
}
else
{
  /* clear one bit */
  /* set one bit */
  set_clear_bit = index << 1
  *MyBitMap = *MyBitMap & ~set_clear_bit;
}
```

(4) Scheduler

```
void scheduler()
{
   for (;;)
   {
        for (i = 1; i < NUM_OF_BITS; i++)
        {
          if (IsSet(bitmap,i) == TRUE)
         {
            dequeue(item,i);
            service the item;
            if (IsEmpty(ArrayHeader[i].Header) == TRUE)
               /* clear the bit in the bitmap */
               setBitMap(bitmap, i, FALSE)
         }
    }
}
```

6.3.7 Weighted Fair Queue implementation

For illustrative purposes, we assume a calendar queue implementation of WFQ (Weighted Fair Queue). Normal operation is as follows:

There are two levels of queues. At the first level are the per-flow queues, one per flow. At the second level are the calendar queues, number 1 through n. The queuing component has no per-Connection queues (In general, several flows are transmitted over a single connection) - just per-flow queues and a set of n calendar queues. The calendar queues sit "closest" to the shaper (a shaper is used to limit the amount of traffic sent on a connection), and when a packet is removed from a calendar queue, it will be delivered to the shaper unless the connection for this packet is backlogged. When a packet is removed from a per-flow queue, it is placed on a calendar queue. There can only ever be one packet from a given flow queue in the complete set of calendar queues.

The calendar queues are served as follows: we begin at calendar queue 1 and serve all packets from it; now move to calendar queue 2 and serve all packets from it; and so on. After serving all packets from calendar queue n, return to calendar queue 1 and begin again. We can think of each of the queues being a sort of time slot. The queue that is being served represents the current time, and other queues represent times in the future. Only one packet from a given flow can be on any of the calendar queues at any given time. Each flow has an associated weight which determines what its share of the link should be. For example, if flow X has a weight of 2 and flow Y has a weight of 6, X should get 3 times as much bandwidth as Y if both flows always have packets waiting in the flow queues.

Packets are moved from the flow queues into the calendar queues in the following way:

If a packet from flow X arrives in the system, and there are currently no packets from flow X in the system (i.e. neither the flow queue for X nor the calendar queues have a packet from X) then the packet is placed on the calendar queue that is W slots away from the calendar queue that is currently being served, where W is the weight associated with flow X. That is, we schedule the packet for transmission W timeslots in the future.

If a packet from flow X arrives in the system, and there are packets from flow X already in the system, then it is simply placed on the per-flow queue associated with flow X.

Whenever a packet from flow X is served from a calendar queue, the per-flow queue for flow X is examined and, if non-empty, the first packet in the queue is removed and placed on a calendar queue. The calendar queue on which it is placed is the one that is W slots away from the calendar queue from which the flow X packet was just removed. Again, we schedule the packet for transmission W units in the future.

Chapter 7 Conclusion

This conclusion is composed of a summary of contributions made in this report and the description of the future work.

7.1 Contributions

In this report, we have discussed the most important issues on implementing a high performance ATM switch. They are: Network Software Architecture and Cell Scheduling and Queuing.

In order to implement a high performance ATM network, an appropriate software architecture must be chosen. Three software architectures – procedure-based, process-based and hybrid architectures for ATM protocol implementation are presented and justified. We have made detailed studies on these three approaches and have shown how these approaches can be used to implement the ATM communication stack.

In the procedure-based architecture, we have presented the ATM architecture based on the procedure-based approach and described the ATM. We have also studied two variants for the procedure-based ATM architecture.

In the process-based architecture, we have shown how to apply Buhr's process-based approach to the ATM protocol implementation.

A process-based implementation can achieve a maximal parallelism but the inter-layer communication is complex and incurs a high penalty in time. A procedure-based implementation has a simpler, fast inter-layer communication mechanism but the wait time for a user call may be very long. A hybrid approach can solve this problem. The hybrid-based architecture is a combination of the above two implementation. In this report, we have shown how the hybrid-based approach is used to implement a ATM switch and have compared the three different architectures based on the implementation's complexity and the performance.

In an ATM network, cells from different connections interact with each other at each switching node. These interactions may adversely affect the network performance without appropriate control.

In this report, we have summarized the previously proposed disciplines on cell scheduling algorithm and made the comparison between them. We have analyzed ATM switch requirements for the scheduling algorithms, presented the scheduling algorithms which can be used in the ATM network, compared the different scheduling algorithms based on the implementation's complexity, and the performance.

We have also given the guidelines to choose a cell scheduling algorithm for an ATM switch and presented the implementation of the scheduling algorithm.

7.2 Future work

(1) How are the time-critical traffic and non time-critical traffic scheduled within an ATM switch? What scheduling algorithm should be used?

In this report, we present the several scheduling algorithms which can be used in a ATM switch. They are: Fair queuing/weighted fair queue, EDF, Stop-and-Go, Rate-Controlled Static Priority, FCFS and etc. However, these scheduling algorithm are good for either time-critical traffic (EDF and static priority) or non time-critical traffic (FCFS, Fair queuing and weight fair queuing). In many applications, we need to consider to schedule both time-critical traffic and non time-critical traffic.

(2) Study of an efficient implementation architecture for accommodating a mix of the time-critical traffic and non time-critical traffic

In this report, we present the three architectures – procedure-based, process-based and hybrid architectures. These architecture are suitable for the non time-critical traffic. If we consider the time-critical traffic, it may be necessary to introduce a priority mechanism into the architecture.

References

[BO92] Jean-Yves Le Boudec: The Asynchronous Transfer Mode: a tutorial, *Computer Networks and ISDN Systems* 24 p. 279-309 (1992).

[BU84] R. J. Buhr, *System Design with Ada.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

[CJ89] D. Clark, V. Jacobson, et al. An Analysis of TCP Processing Overhead, *IEEE Communications Magazine*, vol.27, no.6, p. 23-29, June 1989.

[CL85] David D. Clark. The Structure of System Using Upcalls. *Proc. 10th ACM Symp. On OS Principle*, p. 171-180, 1-4 Dec 85, USA.

[CPU] CPU Scheduling,
http://infocom.cqu.edu.au/Units/win2000/85349/Resources/Lectures/5/1/index.html

[CY78] R. Cypser, *Communications Architecture for Distributed Systems*, Addison Wessley, 1978.

[FL94] Paxson, Floyd; Wide Area Traffic: Failure of Poisson Modeling, *Comupter Communication Review*, vol.24, no.4, p. 257-268.

[GA93] Parekh, Gallagher; A Generalized Processor Sharing Approach to Flow Control – the Multiple Node Case, *IEEE INFOCOMM*, 1993.

[GGP97] L. Georgiadis, R. Guerin, and A. Parekh. Optimal multiplexing on a single link: delay and buffer requirement. *IEEE Transactions on Information Theory*, 43(5) : p. 1518-1535, Sep. 1997.

[HA86] E. Hahne, Round Robin Scheduling for Fair Flow Control in Data Communication Networks, Report LIDS-TH-1631, Laboratory for Information and Decision Systems, MIT Dec. 1986.

[HW89] N. R. Howes and A. C. Weaver, Measurements of Ada overhead in OSI-Style communication systems, *IEEE Trans. Software Eng.*, vol 15, no. 12, p. 1507-1517, 1989.

[KA90] Gerald M. Karam, Comments on "Measurement of Ada Overhead in OSI-Style Communication Systems", *IEEE Trans. On Software Eng.* Vol. 16, no 12. p. 1435-1439, Dec. 1990.

[KE91] Keshav, On the Efficient Implementation of Fair Queuing, Internetworking: *Research And Experience*; vol 2, p. 113-131 (1991).

[KHBG91] H. Kroner, G. Hebuterne, P. Boyer, A. Gravey, Priority Management in ATM Switching Nodes. *IEEE JSAC*, Vol. 9, No. 3, p. 418-427, April 1991.

[KS89] Demers, Keshav, and Shenker; Analysis And Simulation of a Fair Queuing Algorithm, *Proceedings of ACM SIGCOMM*, p. 1-12, 1989.

[LDW] Todd Lizambri, Fernando Duran and Shukri Wakid, Priority Scheduling and Buffer Management for ATM Traffic Shaping. *Proceedings of 7th IEEE Workshop on Future Trends of Distributed Computing Systems – FTDCS'99*, p. 36-43, Dec. 1999.

[LWF96] J. Liebeherr, D. Wrege, and D. Ferrari, Exact admission control for networks witj a bounded delay service. *IEEE/ACM Transactions on Networking*, 4(6) p. 885-901, Dec. 1996.

[MA] Peter Marwedel, Software and Compiler Generation for Embedded Systems, http://ls12-www.cs.uni-dortmund.de/publications/slides/dak_forum/tutorial/sld001.htm

[MS98] David E. McDysan and Darren L. Spohmn, ATM theory and Application. McGraw-Hill 1998.

[MZ92] N. Malcolm and W. Zhao. Advances in Hard Real-Time Communication with Local Area Networks. *17th Conference on Local Computers Networks*, p. 548-557, Sept., 1992.

[NA87] J. Nagel, On packet switches with infinite storage, *IEEE Transactions on Communications*, COM-35(4), p. 157-173, April 1987.

[OS] Operating Systems: Process Control & Scheduling, http://www.cs.jhu.edu/~yairamir/cs418/os2/sld001.htm

[RA87] K. K. Ramakrishnan, D.-M. Chiu, and R. Jain, Congestion Avoidance in Computer Networks with a Connectioness Network Layer; DEC Technical Report TR-510, DEC, Nov. 1987.

[RE90] Recommendation I.121, Broadband Aspects of ISDN. *CCITT SG XVIII, Report R34*, June 1990

[SI84] A. Silberschatz, Cell: A distributed computing modularization concept. *IEEE Trans. Software Eng.*, vol. SE-10, no. 2, p. 178-185, 1984.

[SV95] M. Shreedhar and George Varghese; Efficient Fair Queuing using Deficit Round Robin, *Proceedings of ACM SIGCOMM*, p. 231-242, Aug., 1995.

[SZ92] Clark, Shenker and Zhang; Supporting Real-Time Applications in an Integrated Services Packet Networks: Architecture and Mechanism, Proceedings of ACM SIGCOMM, p. 14-26, 1992.

[SZ93] K. G. Shin and Q. Zheng. Mixed time-constrained and non-time-constrained communications in local area networks. *IEEE Transactions on Communications*. Vol. 41, No. 11, p. 1668-1676, 1993.

[WL97] D. Wrege and J. Liebeherr. A near-optimal packet scheduler for QoS networks. In *Proceedings of IEEE INFOCOM'97*, p. 23-29, 1997.

[ZH90] Zhang, Virtual Clock; A new Traffic Control Algorithm for Packet Switching Networks; *Proceedings of ACM SIGCOMM*, p. 19-29, 1990.