...eit Groningen Faculteit der Wiskunde en Natuurwetenschappen

Informatica

B

Visualising Transaction Processing and Connection Management in ATM Networks

Alard de Boer

Supervision:
Prof.dr.ir. L.J.M. Nieuwenhuis
Ir. S. Westerdijk (KPN Research)
Ir. M.R. van der Werff (KPN Research)

August 1997

RuG

Contents

List of Abbreviations

AAL	ATM Adaptation Layer
ACID	Atomicity, Consistency, Isolation, Durability
ATM	Asynchronous Transfer Mode
CDV	Cell Delay Variation
CLP	Cell Loss Priority
CLR	Cell Loss Rate
CORBA	Common Object Request Broker Architecture
DAE	Distributed Application Environment
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DPE	Distributed Processing Environment
GSMP	General Switch Management Protocol
IDL	Interface Definition Language
KPN	Koninklijke PTT Nederland
MaxCTD	Maximum Cell Transfer Delay
MBS	Maximum Burst Size
MCR	Minimum Cell Rate
NE-RM	Network Element Resource Manager
NNI	Network-Network Interface
ODE	Open Distributed Environment
OMG	Object Management Group
OO	Object Oriented
ORB	Object Request Broker
OSF	Open Software Foundation
OTS	Object Transaction Service
PCR	Peak Cell Rate
PVC	Permanent Virtual Connection
QoS	Quality of Service
RFC	Request For Comments
SCR	Sustainable Cell Rate
SNMP	Simple Network Management Protocol
SVC	Switched Virtual Connection
Tcl	Tool Command Language
TINA	Telecommunications Information Networking Architecture
Tk	Toolkit
TM	Transaction Manager
TP	Transaction Processing
UNI	User-Network Interface
UPC	Usage Parameter Control

VC	Virtual Channel
VCB	Virtual Channel Branch
VCC	Virtual Channel Connection
VCI	Virtual Channel Identifier
VCL	Virtual Channel Link
VP	Virtual Path
VPB	Virtual Path Branch
VPC	Virtual Path Connection
VPI	Virtual Path Identifier
VPL	Virtual Path Link
2PC	Two-Phase Commitment

List of Figures

1 Introduction

1.1 Problem Definition

This document reports on a graduate project in Applied Computer Science[1] at the University of Groningen; it was carried out at KPN Research. KPN Research is the research department of KPN, the Royal Dutch PTT. KPN Research is one of the members in the ACTranS project, an international project with participants from many European countries. The ACTranS project demonstrates the use of transaction processing in connection management in ATM networks.

One of the basic operations in computer networks is the setup of connections from one end-terminal to another. A network is a means of data transport, consisting of switches and lines between the switches (see Figure 1). However, coordination of the switches is difficult. When creating a connection over multiple switches, some switches might not be able to create their part of the connection, while others are.

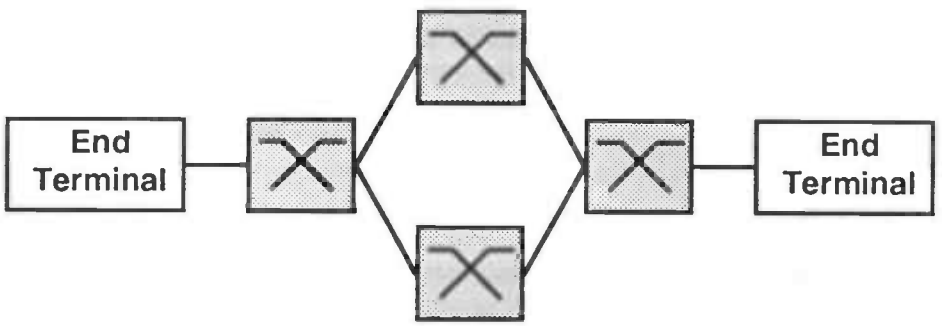

Figure 1 - A network with two end terminals and four switches

A possible solution for the problem of coordinating the setup of connections is the use of *transaction processing*, a technique traditionally used with databases. Using transaction processing, you can guarantee that either *all* switches make their part of the connection, or *none* of them does. The need for coordination is clear: when partial connections are set up, some switches may keep their part of the connection, which is never used and, more importantly, might never be removed. This means a claim on resources will remain. The ACTranS project uses transaction processing for the coordination of the setup of connections in ATM networks.

Figure 2 shows the elements that are built or used in the ACTranS project. At the top, a news-on-demand application is shown that requires the use of an ATM network. The connections this application needs are created by the Connection Management layer. Individual switches are managed by NE-RMs (Network Element Resource Managers) that provide Connection Management with a uniform interface to ATM switches. Different types of ATM switch are managed by different NE-RMs.

Within the ACTranS project, many European companies work together. KPN Research will build the Connection Management component. In order to be able to test Connection Management, you could use an ATM network with software that offers the functionality of the NE-RMs, as described above. However, this would mean that an ATM network has to

[1] In Dutch: Technische Informatica.

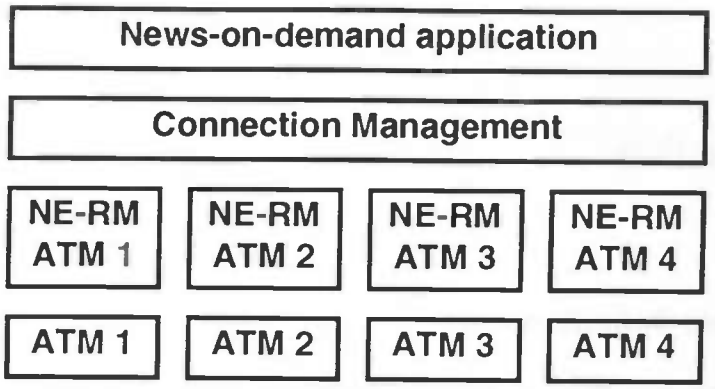

Figure 2 - Overview of the ACTranS project

be built, which is expensive, and the NE-RM software must be available. This software is built by other partners in the project, so that would mean Connection Management could not be tested before they are finished.

To be able to test Connection Management, a *simulation* can be used. The simulation offers the functionality that would normally be provided by the NE-RMs. During this graduate project, such a simulation was built. An important feature of the simulation is that it can be addressed using transaction processing.

Next to the simulation, a network *visualisation* was built. In the visualisation, an overview is given of the layout of the network, with all switches and lines between the switches. Furthermore, the dynamic behaviour of the network is shown, that is, when connections are added to the network, they are visualised, and the status of all elements will be shown. The effect of the operations performed on the simulated network is made clear in the visualisation.

Using the simulation and the visualisation, the Connection Management application can be tested before integration with other software, built by partners in the project. After testing the transition to a hardware network, with the software layers provided by others, should be trivial, since the interface to the simulation and to the NE-RMs is the same.

1.2 Structure of the Thesis

Chapter 2 introduces ATM networks. ATM can transport different kinds of data on the same network by splitting all data into cells. ATM switches offer the functionality to route cells from one line to another.

A network containing elements that can access each other is called a distributed system; such systems are described in Chapter 3. CORBA is a standard that allows clients and servers in a distributed environment to communicate with each other. Next to this basic functionality, CORBA Services have been defined.

Chapter 4 describes Transaction Processing, the CORBA Service that is used by the ACTranS project to coordinate connection management in ATM networks.

When connections are set up using transaction processing, applications become increasingly complex. To gain insight in complex systems, visualisations can be used. Creating a good visualisation, however, is a hard task. Chapter 5 describes methods and ideas for creating visualisations. These ideas were used in the visualisation of the simulated ATM network.

The design of the simulation is described in Chapter 6. The simulation offers the same functionality as an ATM network realised in hardware, as far as connection management is concerned.

A graphical overview of the simulated network is shown in the visualisation, with the effect of the operations performed on it. The techniques used for creating such a visualisation, its design and implementation, and the link between the simulation and the visualisation are described in Chapter 7.

The conclusions are drawn in Chapter 8; this chapter also gives recommendations for future research.

2 ATM Networks

The ACTranS project demonstrates the use of transaction processing in ATM networks. ATM networks are described in this chapter: the transport of data, the switching from one place to another, and the management of the switches. An ATM network, as described here, can be simulated for testing purposes. The design of such a simulation is described in Chapter 6.

2.1 Introduction

In the ACTranS project, connection management is performed in *ATM networks* [21]. The **Asynchronous Transfer Mode (ATM)** is a definition of how data can be transported through a network. One of the basic ideas behind ATM is the capability to transport different kinds of data over one network, with different requirements on that network.

For example, audio is sensitive to the *delay* between sending and receiving, and to *jitter*, the variation in delay. It is less sensitive to minor errors in transmission, since small errors are simply perceived as static. For uni-directional video connections, delay is not very important, but for bi-directional video (video conferencing), it is important. Errors are again of minor importance.

On the other hand, computer data is very sensitive to errors. If a single bit should change, the meaning of a message can change completely. Delay and jitter are irrelevant; usually timing is not important.

To transport these different kinds of data, ATM divides all types of data in so called **cells** at the source. These cells are sent through the network, and reassembled at the destination into their original form. This means that all data is transported in the same way; the only difference is at the end-points where the cells are inserted in or extracted from the network. Here it is of course important what kind of data was transported in the cells.

A network consists of switches, and lines between these switches. To transport the cells from the first switch at the source, to the last switch at the destination, a mechanism called **virtual circuit switching** is used. This means that the route the cells will follow is determined when the connection is set up. The switches on this route will send incoming cells to the next switch on the virtual connection.

This chapter describes how data is transported in ATM networks in Section 2.2, how ATM switching is done in Section 2.3, and how ATM switches can be managed in Section 2.4. The information in this chapter is taken from [1], [2], [3], [4], [13] and [17].

2.2 ATM Data Transfer

As described above, the ability to transfer data with different characteristics is fundamental to ATM. To be able to do this, the parameters for transport through the network are collected in the **Quality of Service** (QoS) of that connection. The QoS is determined when a connection is established, along with the route of the connection.

Different classes of QoS are defined [4]:

- Service Class A: Constant bit rate video, circuit emulation
- Service Class B: Real-time variable bit rate (video, audio)
- Service Class C: Non real-time variable bit rate (data)
- Service Class D: Unspecified bit rate

Next to these four classes, Service Classes are available for "best effort" service and "available bit rate". These classes of QoS have different traffic parameters (PCR, SCR, MBS, MCR) and service parameters (maxCTD, CDV, CLR). Their meaning is described in Figure 3.

PCR	*peak cell rate*	the maximum traffic rate that can be submitted to the network
SCR	*sustainable cell rate*	the average rate of traffic that can be submitted (the average is calculated over a "long" period of time)
MBS	*maximum burst size*	the maximum length of a burst
MCR	*minimum cell rate*	the minimum cell rate the network must always be able to handle
maxCTD	*maximum cell transfer delay*	the maximum delay in transferring a cell from the source to the destination
CDV	*cell delay variation*	the variation in cell transfer delay
CLR	*cell loss rate*	the maximum rate of cells that might get lost because of heavy network traffic

Figure 3 - ATM QoS parameters

The different QoS classes can be defined using these 7 parameters. For example, class A is characterised by low CDV and a PCR close to the SCR. This class can be used for transfer of speech, since speech needs to be transmitted with constant delay. Class C is less concerned with cell delay, but demands a low or zero CLR. This can be used for computer data, since minimal loss of data is important.

Another aspect in the difference of connections, next to QoS, is that both unicast and multicast connections can be set up, as well as uni- and bi-directional connections. However, not all different types of connections have been standardised. In practise many of these will not be possible. There are even contradictions on this subject in the various texts about ATM. [1] and [3] state that ATM connections are unidirectional, while [2] states that they are bidirectional. However, this is not a big problem, since at a switch (see Section 2.3), a bi-directional branch would simply be equal to two uni-directional branches.

ATM defines several layers of communication. The highest layer is the **ATM Adaptation Layer** (AAL). This layer splits up the various kinds of data in cells. Since there are several different kinds of data (speech is sent through a network as a bitstream, data is usually sent in packets), there are several different kinds of AAL to split it up into cells. The cells can then be reassembled by the appropriate AAL at the destination to recreate the original data.

An ATM cell is a block of 53 bytes, consisting of 48 bytes of user data and a 5-byte header, as shown in Figure 4. The most important information in the header is the identification of the virtual connection the cell is part of. Next to this information the cells contain an error detection and correction code, some indication of the contents of the 48 bytes user data, and a code used when congestion occurs.

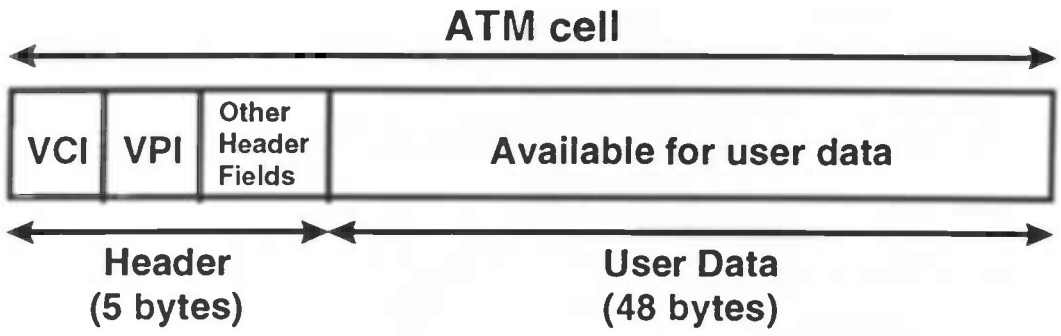

Figure 4 - Contents of an ATM cell

The cells are transported by the **ATM layer**. They are split up into bits by this layer, to be transported by the underlying physical layer. The physical layer uses some method of transferring bits; this is not part of the definition of ATM.

Figure 5 shows the relation between these layers. At ATM Endpoint 1, some application software wants to send data (*User Data*) through the ATM network. This data is split up into cells by the AAL, the cells are split up into bits by the ATM layer, and the bits are transported by the physical layer. At every ATM switch, the bits are regrouped into cells to check the header for identification of the virtual connection (see Section 2.3). The cells are split up into bits again, and they will arrive at the destination, ATM Endpoint 2. Here the bits are grouped into cells by the ATM layer, and the cells are grouped into the original user data by the appropriate AAL.

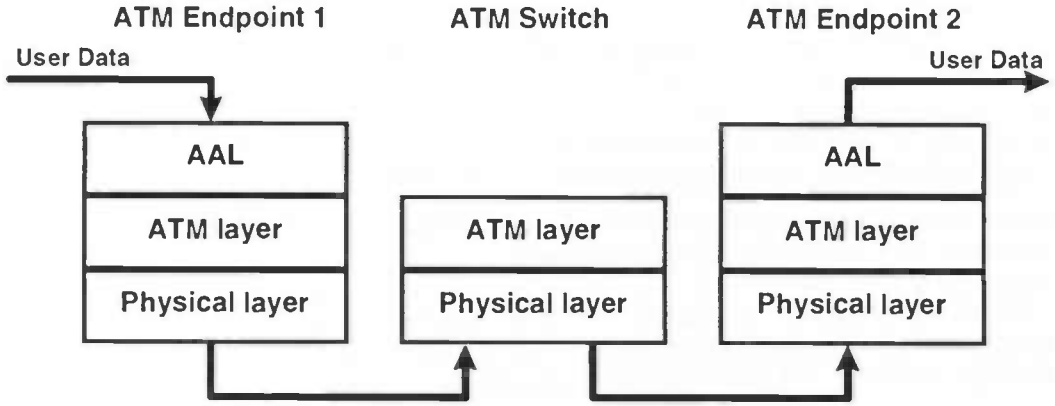

Figure 5 - Layers in an ATM network

2.3 ATM Switching

ATM uses *virtual circuit switching* to transport data. The route the cells will follow is determined when the connection is set up, by telling every switch on the route that a virtual connection is laid through that switch. Every cell arriving at a switch is examined to see which virtual connection it is part of. Subsequently, it is sent on to the next switch. This method of setting up connections is called **virtual circuit switching**.

An **ATM switch** is a node in an ATM network, mainly concerned with building a virtual connection and transporting cells on that virtual connection. However, it has several other tasks as well, such as keeping statistics of every connection, buffering of cells when network traffic is high, and more.

A switch is a piece of hardware, consisting of some sort of switching fabric, and multiple ports to the network, where the lines to other switches are connected. Every physical port is both an input and an output port. Inside the switch, data from an input line is sent to the right output line by the switching fabric. Next to these elements a switch has buffers

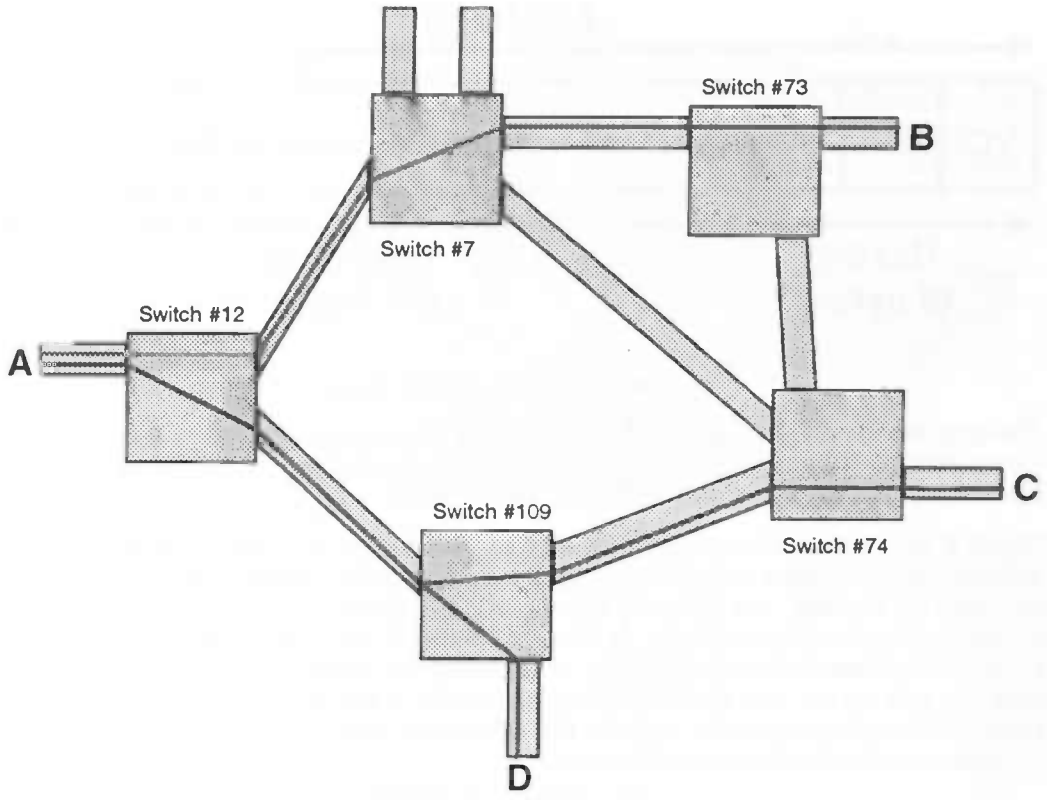

Figure 6 - An ATM network

to be able to deal with temporarily heavy network traffic, and several mechanisms to manage and control the switch and the data it transports.

Figure 6 shows an example ATM network, with five switches. Each switch is identified by a number here, but the identification can equally be an ID string, for example. Here, two connections have been made. The first (shown in red) is a unicast connection between endterminals **A** and **B**, established over three switches: #12, #7 and #73. The second (shown in blue) is a multicast connection from **A** to both **C** and **D**. In switch #109 all cells arriving "on the left" are routed to both the output lines "on the right" and "below".

To transport cells through the network, cells are sent over *Virtual Channel Connections* and *Virtual Path Connections*. Both types of connection are called "virtual", since they are built independent of the physical connections between the switches; they only exist because the network has defined them.

- A **Virtual Channel Connection** (VCC) is a data connection between a source and a destination. A VCC is a concatenation of Virtual Channel Links (VCLs). VCLs are connected at switches; every VCL is identified by a number called Virtual Channel Identification (VCI).
- Many VCLs can be set up over a physical link, and VCLs can be grouped into Virtual Paths. A **Virtual Path Connection** (VPC) is a concatenation of Virtual Path Links (VPLs), each identified by a Virtual Path Identifier (VPI). Every VPL "contains" a group of VCLs. A VPC is an abstraction from the physical connections.

To make these definitions a bit clearer, consider the following analogy. A physical link (glass fibre, copper, etc.) connecting two switches can be seen as a large tube. This tube can contain many smaller tubes, the VPLs. The smaller tubes contain wires (VCLs); the wires transport the data. See also Figure 7.

At the connecting points between the large tubes (the switches), the small tubes can either end, allowing the single wires to be connected (VC switching); or the small tubes

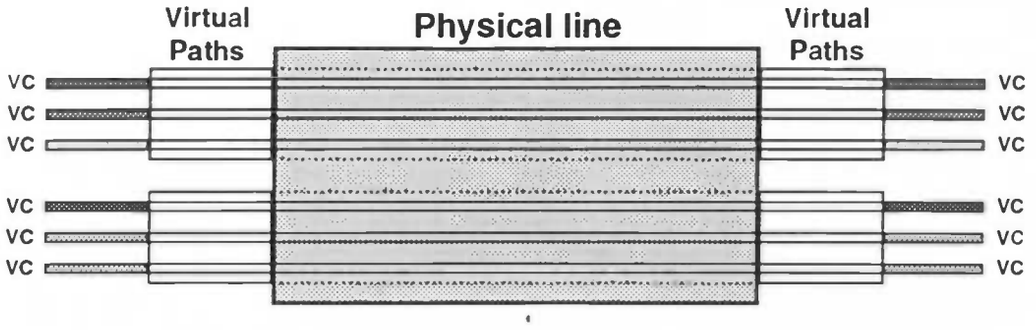

Figure 7 - Relation between physical lines, VPs and VCs

can be connected themselves, including all the wires inside (VP switching). Either way, the wires are connected; the wire in its full length from starting point to endpoint is a VCC.

Using VPs to group VCs has the advantage of allowing a group of VCs between two points in the network to be managed together. For example, bandwidth allocation can be performed within the scope of the VP. If the VP has been allocated a certain bandwidth, VCs can be allocated in that bandwidth, without the need to consider the bandwidth of the physical lines they were set up on. Furthermore, the route a group of VCs follows can be changed simultaneously by changing to a different VP between the same end points, that uses a different route.

When a connection is set up, every switch on the route is told to make an internal **branch**, so the connection is set up one branch at a time. Both VP branches (VPBs) and VC branches (VCBs) can be set up. Note that setting up a connection this way poses problems, when one or more of the switches on the route cannot make the branch. What should happen to the branches already set up? This problem and a possible solution are described in Section 0.

After all branches have been made, cells can be offered to the first switch and they will "reappear" at the last switch. The source and the destination can then simply view the network as a direct connection they can transmit data through.

In the simple case of one source and one destination (a unicast connection), the switch transports the data from one input port to one output port. However, it's possible to set up multicast connections, from one source to multiple destinations (as shown in Figure 8). At a branching switch, the incoming cells are sent to multiple output ports.

As noted in Section 2.2, ATM is uni- or bi-directional. When bi-directional end-to-end connections are set up, every switch must add two elements to its connection table; a uni-directional connection involves the addition of one branch per switch.

Figure 8 shows an example ATM switch. Virtual Path 3 on port #1 is connected to VP 7 on port #6, along with all the Virtual Channels in it. The other VPs end in this switch, and the individual VCs are connected. VC 2 on VP 8 on port #2 is connected to two output VCs, so these VCs are part of a multicast connection.

Two types of virtual connections can be distinguished: **switched virtual connections** (SVCs) and **(semi) permanent virtual connections** (PVCs). Switched virtual connections are established by messaging and negotiation between the switches; (semi) permanent virtual connections are established externally by a network management application. They only differ in the way the switch operations arrive at a switch, in both cases the operations that can be performed are the same. The actual operations are defined in a protocol like the General Switch Management Protocol [3], as described in Section 2.4. To be more accurate, a switch that receives its commands through network management, is called *crossconnect* instead of switch, but the words are used interchangeably.

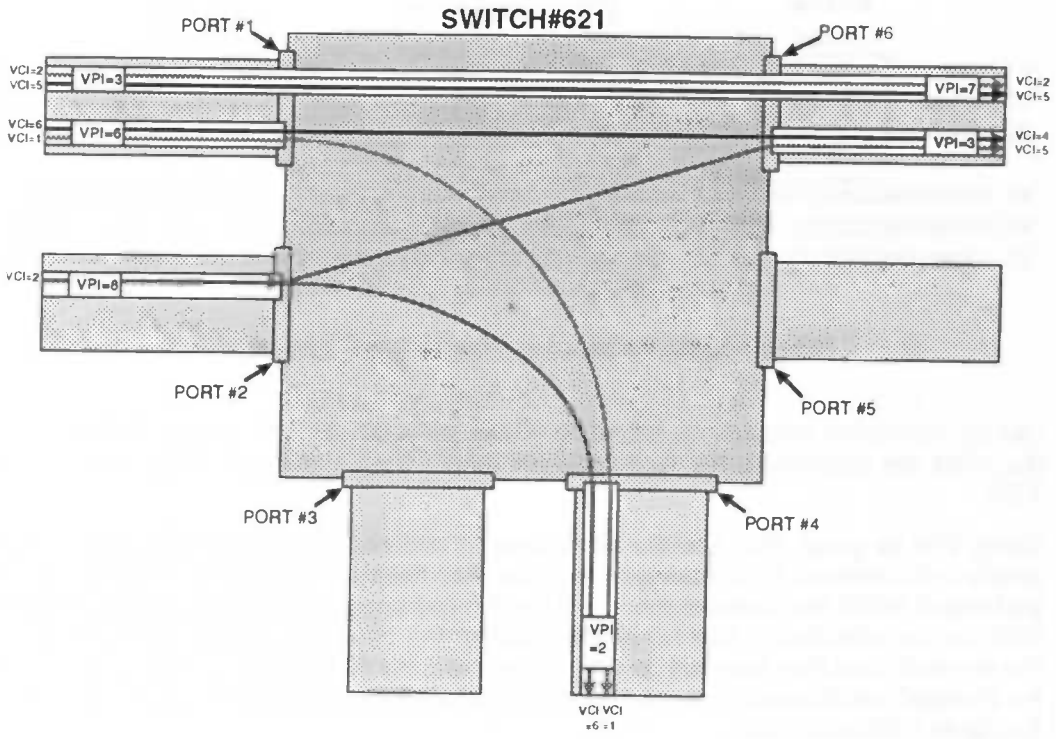

Figure 8 - An ATM switch

At a switch, incoming cells are identified by three parameters: the input port, the input VPI and the input VCI. Given these parameters, the switch will look in its connection database and determine the output port(s), output VPI(s) and VCI(s). VPI/VCI values are only local means of identification; the switch checks the incoming VPI/VCI, decides which VPI/VCI the outgoing VCL(s) must have, and changes the VPI/VCI values in the cell header so the next switch knows what to do with the cells, etc.

Essentially, that is the main task of an ATM switch: wait for an incoming cell, check its header and determine where to send it, changing the VPI and VCI values in the header before forwarding. Note that every *port* has its own set of *VPIs* and every *VPI* in every port has its own set of *VCIs*, so you need all three parameters to uniquely identify a connection. The same VCI and VPI, but on different ports, constitute a different connection.

This also means that the only way to identify an end-to-end connection is to compare the VPI and VCI values in switches that are physically connected. After the route is established, neither the individual switches, nor the endterminals know more than a very small part of it: they only know where to send cells arriving with a given set of parameters.

Figure 9 shows the VPB and VCB tables for the switch in Figure 8. The switch will first try to match incoming cells with the entries in the VPB table, and if that fails it will try to match the cells with the entries in the VCB table.

As said before, next to the actual switching, a switch has several other tasks. First, different cells may have different priorities, depending on their QoS parameters. A cell with a higher priority has a greater chance of being sent on earlier than a cell with lower priority. Next, the switch must maintain several statistics for network management purposes, like the number of cells processed. Furthermore, since it is unknown when data arrives, it is possible that a lot of cells arrive at the same time. The switch must have some sort of buffering capability, and when the buffers overflow, cells have to be discarded. The switch must decide which cells to discard, taking their priority into consideration. The cell loss rate must be kept as another statistic.

VPB Table			
inVPI	inPort	outVPI	outPort
3	1	7	6

VCB Table					
inVCI	inVPI	inPort	outVCI	outVPI	outPort
6	6	1	4	3	6
1	6	1	1	2	4
2	8	2	5	3	6
2	8	2	6	2	4

Figure 9 - Example VPB and VCB tables

2.4 Switch Management

Creating or removing branches is performed by either signalling between the switches, or by some kind of management. At this moment, there is no defined standard for switch management. Some use SNMP for setting and getting switch data, or a proprietary interface, using telnet.

In the ACTranS project an interface is used, based on the *General Switch Management Protocol* [3], a proposed standard for ATM switch management. In this interface, the following operations can be performed on a switch. Chapter 6 describes the exact functionality. Note that this interface is only concerned with connection management operations.

- creating a VP or VC branch;
- deleting a VP or VC branch;
- modifying parameters of a branch, either QoS parameters or input/output parameters;
- changing status of the switch or a port;
- resetting the switch or a port;
- retrieving information about the switch or a port, either statistics or entire connection tables.

2.5 Summary

ATM networks are able to transport many different types of data. Every type of data is split into cells at the source, that are uniformly transported by the network, and re-assembled into their original form at the destination. Cells that arrive at an ATM switch are sent to the next switch until the destination is reached.

In the ACTranS project, ATM networks will be addressed using distributed object technology. More specifically, the NE-RMs will be realised as distributed objects. This subject is described in the next chapter.

3 Distributed Systems

The previous chapter discussed many aspects of ATM networks. Such networks will be addressed in the ACTranS project using a distributed application. This chapter describes what distributed systems are, and how they can be used. The simulation that is described in Chapter 6 will be addressed using this technology.

3.1 Distributed Processing

A *distributed system* is a collection of elements in a network, that can use each other's services. Traditionally, computers were designed for a given set of tasks. A graphical workstation could run applications requiring special graphical capabilities. Supercomputers were designed to run applications requiring massive computational power.

With distributed systems, however, computers are connected to each other in a network, and applications can be distributed over multiple computers. This means that, for example, the capabilities of all these computers can be combined in a single application. With the examples given above, one application might perform its computations on a supercomputer, and display the output graphically on the workstation.

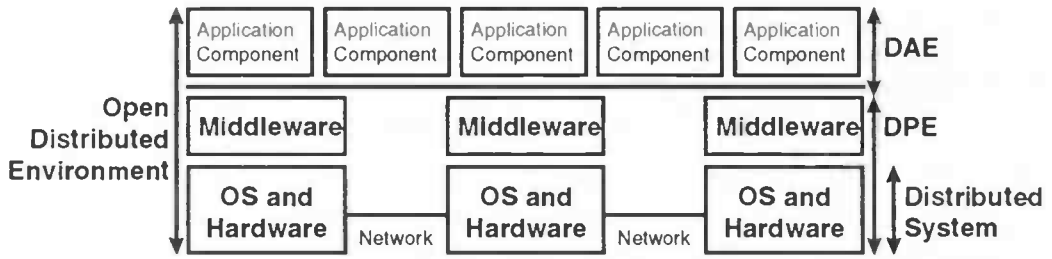

Figure 10 - Terminology in distributed systems

In [19], several terms concerning distributed processing are defined more precisely. Figure 10 shows their relation. In this field, there is no generally accepted taxonomy; here the definitions in [19] are followed.

A **distributed system** is a combination of computing systems and the (tele)communication networks that interconnect them. A computing system consists of the operating system and the hardware of a computer. A (tele)communication network is a means of communication between computing systems that are geographically distributed.

On top of the distributed system, the **middleware** is a layer of software that supports distributed processing where application components may be distributed over several computing systems.

A **distributed processing environment (DPE)** is the combination of a distributed system with some kind of middleware. The DPE supports the **distributed application environment (DAE)**, in which the application components can be run. The entire system of both the DPE and the DAE is called **open distributed environment (ODE)**.

The application components can act in the role of a *client* or a *server* (or both). A **server** is a program that offers a defined set of services. A **client** can make use of the services of one or more servers. Note that a program can act as both a server and a client: when another program asks the program for a service, it may use other servers to deliver that service.

Dividing a system in clients that use services, and servers that offer those services, makes building such systems more flexible. **Distributed objects** have the advantages of object-orientation: *encapsulation* (the objects are only accessible through a defined interface), and *abstraction* (the internal details of the object are unknown to the outside). When the interface between a client and a server is defined, the server can be built without knowledge of the client and vice versa.

An example of a **client-server** system is the following. Consider a collection of servers that each maintain a database with some kind of information. Clients can then request the information from those servers, and combine it for presentation to the user, drawing conclusions, etc.

Using an ODE has many advantages over the use of a monolithic system (a single large computer with some form of access to it, like terminals).

- Expensive and scarce resources such as large storage media, computing power, or colour printers can be shared by many applications and many users, eliminating the need to equip each system with the same set of resources.
- Data can be stored on a single place and still be accessed from multiple places, so it is not necessary to make multiple copies of the data that have to be maintained individually. Furthermore, the user does not need to know where the data is located as long as there is a server that can provide it[2].
- On the other hand, it is possible to replicate data on multiple places automatically, so in case of unavailability of a node in the network the data can be retrieved from somewhere else.
- Distributed environments are flexible: it is possible to add new, different elements to the system and use them just as easily as the elements that were present before.
- Applications can make use of parallel processing, by dividing the computational work over multiple computers in the distributed environment, gaining efficiency.

The communication between clients and servers in a DPE is done by making calls to each other through the middleware. Several standards have been defined with different characteristics. Among them are DCE, created by the Open Software Foundation; DCOM by Microsoft; and **CORBA** by the Object Management Group.

3.2 CORBA

CORBA, the **Common Object Request Broker Architecture** [20], is a middleware standard that allows clients and servers in a distributed environment to communicate with each other. CORBA is object-oriented and universally usable, independent of the hardware or operating system used. CORBA is the standard that is used in the ACTranS project.

---

[2] An important feature of distributed systems is the difference in ways to reach the desired server. In some situations, it is not important which server processes the requests of a client. For example, if some calculation needs to be carried out, you do not care how it gets computed, or by whom. On the other hand, sometimes you want to be able to choose which server handles the requests. For example, when you print a document on a printer in the distributed system, you want to be able to print it on a printer in the next room, not a printer in another country.

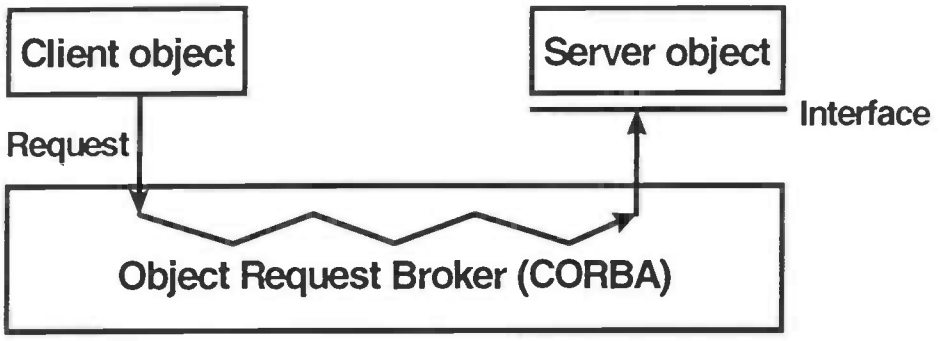

Figure 11 - Client and server can communicate through the ORB

The basic functionality of CORBA is allowing communication between objects in a client and objects in a server, as shown in Figure 11. This communication is independent of details like the operating system the client or server runs on, the byte ordering of the machine, and their physical location.

To accomplish this, the **interface** of a server is defined in advance, in a language called **IDL**, the **Interface Definition Language**. Appendix A gives an example of an IDL file. This interface defines which calls clients can make to the server, with which parameters, etc. If both the client and the server adhere to this interface, they can communicate with each other.

When a client wants to make use of the services of a server, it must *bind* to a server. This is done through an **ORB** (Object Request Broker). Servers register themselves with the ORB, so it knows which servers are active or can be made active. It creates a connection between such a server and the client. After that, the client can make remote calls to the server, just like it is making local calls.

3.3 CORBA Services

To extend the basic functionality of allowing clients and servers to communicate, many *CORBA Services* have been defined [12]. Some of these services are listed here:

1. The *Naming Service* allows objects to be accessed by using structured naming conventions.
2. The *Event Service* allows the system to deal with asynchronous events.
3. The *Transaction Service* enables the use of transactions in accessing objects. See Chapter 4 for a detailed description of the transaction service.
4. The *Concurrency Control Service* coordinates the access to shared resources by multiple objects.
5. The *Licensing Service* provides a mechanism for controlling the use of objects.
6. The *Security Service* allows identification and authentication of users of objects, authorisation and access control of objects, administration of security information, and more security issues.

In the ACTranS project, the Transaction Service is used to coordinate connection management in ATM networks; it is described in the next chapter.

3.4 Summary

Distributed systems are collections of computers and other elements, connected to each other in a network. Applications can make use of such a system using middleware; CORBA is the standard for middleware that is used in ACTranS. The functionality of CORBA is extended by CORBA Services, such as the Transaction Service, that is described in Chapter 4.

4 Transaction Processing

Of the CORBA Services, as described in the previous chapter, the ACTranS project uses the Transaction Service for coordinating connection management in ATM networks. The Transaction Service is described in this chapter.

4.1 Introduction

In Section 2.3 the problem of failing switches and what to do about coordinating all switches was recognised. Suppose you want to set up a connection over four switches, and one of the switches cannot create its branch. This can for example happen, because of a failing port, failure of the switch itself, or simply because the capacity of the switch is filled to the point where the QoS of the new branch cannot be guaranteed.

If there is no coordination over the process, three of the switches will create an internal branch, and the fourth will not, so not the entire connection will be set up. When the network management application notices that the setup was not completed successfully, it might consider the connection non-existent. The three branches will never be used, and more importantly, will never be removed. So, if this happens multiple times, the capacity of the network will slowly diminish because of the "ghost" branches, that are never used but maintain their claim on resources.

What should happen, is that the other switches remove their branches when not the entire connection can be set up. Or, the branches should not be created until is known that all switches *can* make a branch. This can be achieved by using *transactions*.

4.2 Transactions

A **transaction** is a series of operations, guaranteed not to be interrupted by operations from other transaction, and also guaranteed not to be executed partially. If, during execution, it becomes clear that any part of the transaction cannot be completed, all operations already performed have to be undone (this is called "rollback"). After all operations have been completed, the changes made by the operations will be made permanent (this is called "commitment").

For example, consider transferring money from one account to another. The transaction (in a simplified form) consists of two operations: decrease the amount of money on the first account, and increase the amount of money on the second account. It should be clear that it is essential that this transaction is to be executed either completely or not at all; if only the increase or the decrease is performed, money would (dis)appear!

Transaction processing is often summarised in the **ACID properties**:

- Atomicity: the operations in each transaction can be seen as atomic, so either the transaction gets executed entirely, or no effect is seen at all. In the bank example, no money would (dis)appear.
- Consistency: the transaction transforms the data from one consistent state into another consistent state. For example, no money should "disappear" from the bank by partially executed transactions.
- Isolation: during execution of a transaction, no intermediate results can be visible. Only the state before and after every transaction is seen. In the bank example, the

state in which money was deducted from one account and not yet added to the other account can never be seen by other operations.

- Durability: after a transaction is committed, the changes made to resources will be permanent. Failures occurring after a transaction has committed will never undo or alter these changes.

4.3 Distributed Transaction Processing

When used in distributed systems (in applications that run on multiple computers) transaction processing becomes increasingly difficult. Any of the processes working for the transaction (and any of the hosts they run on) is subject to errors, and when an error occurs all changes made by every process have to be rolled back. To allow for **distributed transaction processing**, two "tools" are used: recoverable processes and a commitment protocol.

Recoverable processes are processes that log every operation they perform, and are able to undo the changes they made using that log. For example, a recoverable database will log the changes made during a given transaction, and when necessary it will use the log to undo those changes.

A **commitment protocol** is necessary to coordinate the commitment of every process. The most common protocol is **two-phase commitment** (2PC). With 2PC, every server that is accessed within a transaction, registers itself with a coordinator, the **transaction manager** (TM). When the transaction is ended, the two phases of 2PC are executed.

In the first phase, the TM will ask every registered server if it can commit the changes it has made during the transaction, that is, it can write the new state of the data to permanent storage. The servers answer this question, and the TM collects the answers from all the servers.

In the second phase, the TM will take one of two actions. If all servers responded that their changes could be committed, it will signal all servers to commit. If, however, one or more of the servers responded it could not commit its changes, or did not reply at all, the TM will signal to all servers to rollback the changes.

Either way, the transactional properties will be maintained: all changes will be made permanent, or none of them will.

4.4 The Transaction Service

How does this work in practice? Suppose, you have the following scenario: a client wants to perform operations on multiple servers within a transaction. Next to the client and the servers, the Object Transaction Service (OTS) is present to coordinate everything.

To start a transaction, the client tells the OTS that it will begin a transaction. Every server that is called within the transaction will have some resources (internal state) it manages; for example, a server might manage a switch in an ATM network. The server will register its resource with the OTS.

When the client has done the operations on the servers and is ready to end the transaction, it can do so in two ways. If the client noticed that something has gone wrong during the transaction, or wants to cancel it, it tells the OTS to rollback. The OTS will now send every registered resource a *rollback* call to undo the operations it has performed. If the client thinks everything was OK, it tells the OTS to commit. The OTS will now start the two-phase commitment protocol.

In the first phase, the OTS will send a *prepare* to every registered resource. The resources can respond to this call with

1. *VoteCommit*: the resource is able to commit the changes it has made to the data;

2. *VoteRollback*: the resource cannot commit the changes, and has rolled them back;
3. *VoteReadOnly*: the resource has not altered any data, so nothing needs to be committed.

In the second phase, the OTS will gather all answers and evaluate them, and act in either of two ways. If all resources replied with *VoteCommit*, or with *VoteReadOnly*, the OTS will send a *commit* to every resource that voted *VoteCommit*, and the transaction will be completed with all changes made durable.

If, however, one of the resources responded with *VoteRollback*, the other resources will be sent a *rollback* call. This way, all resources will be rolled back and the transaction will have no effect.

These scenarios can be shown in the following time series diagrams. The time axis runs vertically for the top to the bottom. The figures show the interaction between a client (CLT), two servers (SRV1 and SRV2), and the transaction manager (TM).

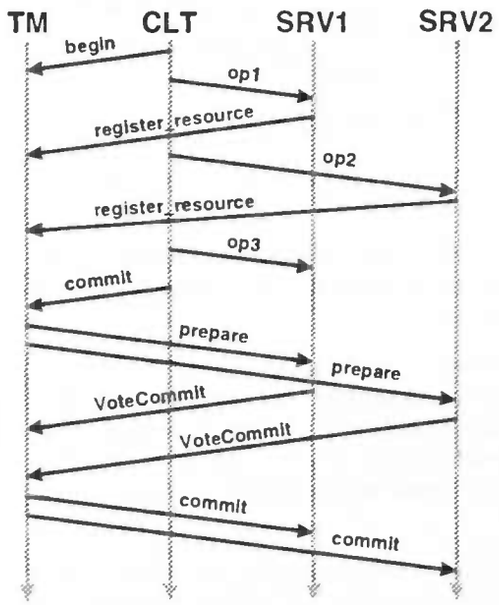

Figure 12 - Interaction between client, servers and TM: commitment

In Figure 12, the scenario depicted starts with the *begin* call to start a transaction. The clients subsequently performs some operations on the two servers (*op1, op2, op3*). The first time a server is accessed within the transaction, it will register itself by issuing a *register_resource* call to the TM. Note that *op3* does not induce such a call, since it is not the first call on that server.

After the client has performed its operations, it will ask the TM to commit, using 2PC. In the first phase, the TM issues *prepare* calls to every registered resource. These resources (servers SRV1 and SRV2) respond with *VoteCommit*, stating they can commit their changes. The TM collects these votes, and decides all resources can commit. So, it sends a *commit* call to both servers, ending the transaction in commitment.

Note that the *commit* call the client sends to the TM is different from the *commit* call the TM sends to the registered resources. The first is the signal to start 2PC, the second is sent in the second phase of 2PC.

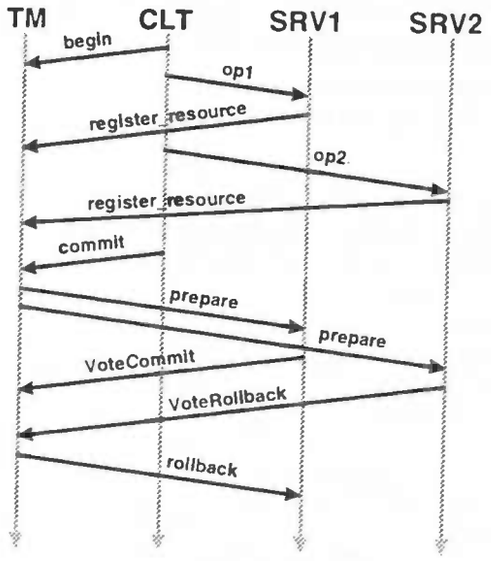

Figure 13 - Interaction between client, servers and TM: server cannot commit

In Figure 13, a similar scenario in depicted. However, during the first phase of 2PC, the second server responds to the *prepare* call with *VoteRollback*, stating it cannot commit its changes. The TM decides that the entire transaction must be rolled back, and it sends a *rollback* call to the other server(s), in this case SRV1. The server that cast the *VoteRollback* vote must rollback before casting that vote, because the TM will not send it a *rollback* call.

Another possible scenario is shown in Figure 14. In this case, the client decides it wants to rollback the transaction. This may for example happen when one of the operations returned a value that was unacceptable for the client. The client issues a *rollback* call to the TM, which sends *rollback* calls to every registered resource. The resources roll back and the transaction will have no effect.

Note that the *rollback* call the client sends to the TM is different from the *rollback* call the

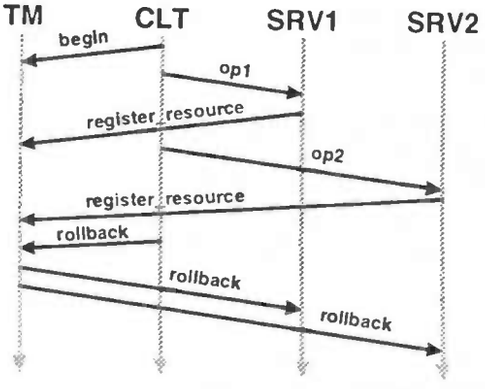

Figure 14 - Interaction between client, servers and TM: client rolls back

TM sends to the registered resources. The first is the signal that the client wants to rollback the entire transaction, the second is the call from the TM to registered resources to indicate they must rollback the changes they made. The last type of *rollback* call is also used during 2PC when one or more resources cannot commit (see Figure 13).

There are some special cases in which this interaction between parties is different. For example, when the client only used one server, so only one resource is registered with the OTS, there's no need for 2PC. In this case, the OTS will send a *commit_one_phase* call to the single resource, which can then try to commit all by itself. The outcome of the transaction is only determined by this resource. This is shown in Figure 15.

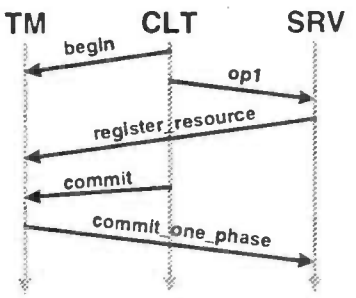

Figure 15 - Interaction between client, one server and TM

When a resource does not reply to the *prepare* call during the first phase within a given timeout, the OTS will send a *rollback* call to every registered resource. It is assumed that the connection to that resource is unavailable, or the resource has crashed itself, so the transaction cannot be committed.

When a resource replies with *VoteCommit* in the first phase, but it cannot commit in the second phase, a so-called **heuristic decision** is taken by that resource. A heuristic decision is taken when resources commit or rollback without being asked to do so by the OTS. This usually only occurs under special circumstances, like entire computers going down. A heuristic decision may not be right when the 2PC is completed with the other resources, resulting in inconsistent states (one resource committed, one resource rolled back). Because using 2PC is not entirely fail-safe, more elaborate algorithms like *three-phase commitment* have been developed; this is not discussed here.

4.5 Operations on Resources

A resource can be part of such a transaction when it supports the operations used above: *prepare, rollback, commit, commit_one_phase, forget*. The *forget* function is called in case of a heuristic decision. You can build your own resources, as long as you implement these five functions correctly.

Next to the interface described here, it is also possible to use the *XA interface* for resources. This interface offers functionality similar to the five functions listed here. For example, Oracle can offer the XA interface and be used as a stable and reliable resource. The obvious advantage is that the programmer no longer needs to implement the interface functions.

In [14] the functionality of the five calls on Resource objects is described.

4.5.1 Prepare

prepare determines which vote the Resource should cast.

1. If no data was modified in the Resource, it casts *VoteReadOnly*, and forgets its

involvement in the transaction; the OTS will not return to it.
2. If data *was* modified, the Resource will check if the changes can be saved to stable storage.
   a) If this succeeds, it will cast *VoteCommit*, write to stable storage that the *prepare* call was processed, write the *recovery coordinator*, and be prepared for a subsequent *commit* or *rollback* call. The recovery coordinator is used by the OTS after a crash, to reconstruct the transaction.
   b) If the Resource cannot write the data to stable storage, it will cast *VoteRollback* and forget all knowledge of the transaction (returning the Resource to its old state); the OTS will not return to it.

4.5.2 Rollback

rollback instructs the Resource to undo all changes performed within the transaction.

1. If no transaction is currently active, no actions will be taken.
2. If a transaction is active, it will rollback all changes made to the data, so that the data in stable storage will be equal to that before the transaction was started. If the Resource cannot rollback all changes and has committed, some or all of them, a *heuristic decision* is taken and an appropriate exception is raised. Otherwise, all knowledge of the transaction is forgotten.

Note that a *rollback* call may be sent to a Resource, even if no *prepare* call was sent before. This happens for example when the client decides to rollback the transaction.

4.5.3 Commit

commit instructs the Resource to commit all changes performed within the transaction.

1. If no transaction is active, no action will be taken.
2. If no *prepare* call was done before, a *NotPrepared* exception is raised.
3. Otherwise, the changes to the data will be written to stable storage, if that wasn't done before (during the *prepare* call).
4. If the Resource cannot commit all changes and has rolled back some or all of them, a *heuristic decision* is taken and an appropriate exception is raised. Otherwise, all knowledge of the transaction is forgotten.

4.5.4 Commit_one_phase

commit_one_phase is used to commit a Resource when it's the only Resource that was registered during the transaction. It instructs the Resource to commit the changed data, if possible, and forget the transaction afterwards. If it cannot commit, roll back all changes and raise a *TransactionRolledBack* exception to signal the OTS that the Resource has rolled back.

4.5.5 Forget

forget is only used after a heuristic decision. In such a situation, created during a *rollback* or *commit* call, the state is kept by the Resource, and it may be called again to commit or rollback. The *forget* call is used to "release" the Resource, and instruct it to forget all knowledge about the transaction.

4.6 Summary

The Transaction Service allows a series of operations to be carried out within a transaction, guaranteeing that either all operations are performed, or no operation is, so no effect is seen. This functionality is used in the network simulation, as described in Chapter 6. When all branches in an ATM connection are set up in a transaction, either all branches or no branch at all will be made. In the simulation, many algorithms are used;

the next chapter describes how complex systems may be made clear by the use of visualisations.

5 Techniques for Visualisation

When many different algorithms are used in a system, it becomes much harder to understand such systems. Visualisation can be used for clarifying algorithms, and gaining an overview of the system. Such a visualisation was built for displaying the state of the simulated network, and the operations on it, as described in Chapter 6. This chapter discusses methods for creating a visualisation.

5.1 Introduction

When using algorithms, the internal working of these algorithms may be difficult to understand. Understanding may involve reading source code, reading long descriptions of the algorithm, and testing the algorithm to see what it does on specific input data.

To make the process of gaining an understanding of an algorithm easier, *visualisation* can be used. A visualisation can show that an algorithm works, and *how* it works. The visualisation shows the data on which the algorithm operates, and during the execution of the algorithm the user can see what happens to the data. The algorithm will modify and use data, and these actions can be shown; the user can see what the algorithm does.

A visualisation of an ATM network will thus show the status of the network, consisting of the switches, the ports, the lines, and most importantly the connections set up. When operations are performed on the network, the results will be shown in the visualisation. When a connection is set up using transaction processing, the visualisation will show how that is done.

So, the goal of the visualisation is to clarify the algorithms used. However, what makes a visualisation clear? Is it possible to construct general guidelines to create a "good" visualisation? These questions are answered in the next sections.

5.2 Overview of Visualisation

Visualisation attempts to provide the user with a mental image of the items visualised. Visualisation is used in many fields within computer science, and is can be divided in three areas [5]:

1. Scientific Visualisation: using graphical representations of data to gain insight in the structure of that data;
2. Program Visualisation: using visualisation to gain insight in the behaviour of a program, to learn how the program works or to make a presentation; also to monitor performance;
3. Visual Programming: specifying a program in a two-dimensional graphical form.

This chapter will address the second area, the use of visualisation to show the behaviour of a program, to show how the algorithms used work. [15] defines Program Visualisation as the use of various techniques to enhance the human understanding of computer programs. This field is also called *Software Visualisation* or *Algorithm Animation*.

Software Visualisation is used for many purposes. Next to gaining insight in the operation of an algorithm, you can use it to find bugs in a program by studying the run-time behaviour, or you can study the performance of a program.

There has been much research in the various areas of visualisation. Some papers focus on which colours you should use, and other drawing techniques [9]; others investigate the use of animations and sound [10]. Other, more specific papers take an algorithm and try to find a good visualisation for that specific case [7], [8], [16].

5.3 General Visualisation Strategies in Literature

In general, there is no fixed set of rules to follow when creating a visualisation. [11] states that good visualisations can only be created by repeatedly evaluating and improving designs. In general a design will not be perfect in the first version, and can be improved in subsequent steps (however, it will never be perfect).

Nonetheless, there are many general ideas and guidelines that can be used to avoid obvious pitfalls. The designer of the visualisation can make use of such ideas, but there will still be a lot of creative work involved. This section presents some ideas taken from literature.

5.3.1 "Color and Sound in Algorithm Animation"

The use of colour and sound in visualisations is described in [10]. It focuses on the use of colour and sound as an enhancement, as compared to simpler black-and-white animations without sound. The examples used are mostly algorithms taken from computer science, and the ideas taken into considerations when creating the visualisations are described.

- Use *multiple views* on a visualised item, instead of a single view. In the simple case, in which you are only interested in one aspect of a simple algorithm one view may be enough, but to gain insight in complex systems you need different views to compare and see what happens.
- Use *state queues*. When something is about to happen to a graphical representation, the user's attention must be drawn to where the action is. When the user does not know where to look, changes on the display may turn into "something happened, but I don't know what or where". For example, you could use highlighting: highlight the area where a change will take place, perform the change itself, and de-highlight the area to show that the operation has completed.
- Create a *static history*, a list built with the subsequent changes during the execution of the algorithm. The user can study the history afterwards, and understand what has happened.
- Take care in the *choice of input data*. Use a small example first, to show how the algorithm works in a simple case. Later, a more elaborate example can be presented to show the general behaviour of the algorithm. Also, when using random values you might not show all possible cases of the algorithm. So to show all cases, use "cooked" data to create a scenario in which they do appear; extreme cases can bring the algorithm to the limits of its possibilities and show its behaviour well.
- Consider if you want *continuous* or *discrete* state changes. When an object moves from one place to another on the screen, you could use a smooth transition in a simple case where the effect is seen. However, in a large display with many small objects and small distances it works evenly well to move the object simply by removing it from the source and replacing it on the destination; the difference will not be noted.
- When visualising algorithms, show *multiple algorithms* next to each other in one display, to allow the user to compare them, for similarities or differences.

5.3.2 "Envisioning Information"

Many aspects of "envisioning information", including computer graphics, maps, time tables, and lots more, are presented in [9]. The author exactly pins down the task of

clearly presenting data by saying: "Designs so good that they are invisible." This work is more general than the other papers discussed in this section, and does not focus on any specific area of visualisation, but gives many examples from many different disciplines.

The following guidelines were originally written for cartography, but apply to the use of colours in general, too.

1. "Pure, bright or very strong colours have loud, unbearable effects when they stand unrelieved over large areas adjacent to each other, but extraordinary effects can be achieved when they are used sparingly on or between dull background tones." Bright colours should not be used for objects that fill large parts of a display (in cartography: do not use a strong blue colour for areas with water, but a dim, light blue colour). They can, however, be used effectively in small colour spots for making small details spring to attention.
2. "The placing of light, bright colours mixed with white next to each other usually produces unpleasant results, especially if the colours are used for large areas."
3. "Large area background or base colours should do their work most quietly, allowing the smaller, bright areas to stand out most vividly, if the former are muted, greyish or neutral. For this very good reason, *grey* is regarded in painting to be one of the prettiest, most important and most versatile of colours. Strongly muted colours, mixed with grey, provide the best background for the coloured theme." This is in parallel with the first point: do not use bright colours for the background of a display; grey or colours mixed with grey are best suited for backgrounds.
4. "If a picture is composed of two or more large, enclosed areas in different colours, then the picture falls apart. Unity will be maintained, however, if the colours of one area are repeatedly intermingled in the other, if the colours are interwoven carpet-fashion throughout the other." Do not divide the display in two regions, each having its own, different, colour if you want to maintain a single display. Note that this effect can be used to divide a display, if that is your intention.

To summarise these four points: use muted colours for backgrounds, large surfaces and objects that do not need the user's attention; use bright, "strong rainbow colours" for important details.

There are many more tips and ideas in the book for creating better displays of information.

Comparisons between objects must be enforced within the scope of the eyespan. So, if the user wants to be able to compare two situations, the two situations have to be showed in the same display. When publishing such displays on paper, do not show the two situations on different sides of a page, forcing the reader to flip the page to compare.

"A design is excellent, when it is governed by good ideas, and executed with superb craft." So, next to good ideas, you also need good execution of those ideas. This means that while your ideas might be excellent, without the proper tools or experience the resulting display may not be able to convey these ideas.

When selecting colours, observe that some colours look a lot like others, so make the difference greater to make the user note the difference. Since yellow is near white, make yellow a little darker to be noticed next to white. Blue is near black, so use a lighter blue when used next to black.

Take care when using closely-spaced grid lines. When the lines are sufficiently far apart, they can be distinguished individually. However, when these lines are brought closer together, they will "melt" into each other and blur into a sort of grey. When you want the grid lines to be clear, do not put them too close next to each other. The blurring effect is called "1+1=3 clutter".

When selecting a palette, it is usually good to use colours found in nature. "Nature's colours are familiar and coherent". This protects users from a display with freakish colours like purple, yellow and red in the same display. However, colours like these can be used to stress the importance of small details, as noted above.

Colours can be used to give different data values an order, but the order of the rainbow colours isn't implicitly associated with an order in value. A gradual change in hue *is* associated with an order.

However, "colour coding by variation of hue, value or saturation is (potentially) sensitive to 'interactive contextual effects'". This means that for example "blue" is observed differently when it is presented on a light blue surface, as compared to the same colour blue on a dark blue surface. Or, conversely, different colours can be made to look alike on well chosen background colours.

This effect is caused by the fact that "any ground subtracts its own hue from colours which it carries and therefore influences". Note that you can use this property: by using the same colour first on white, and then on another colour, you can obtain two different colours, without having to use another colour in your palette. This can be important if you have to use many colours and the graphical capabilities of your output system (be it a computer monitor, be it a colour printer) are limited.

To aid people with "colour-deficient vision", you shouldn't use red and green as distinguishing colour between two data values. Use a light and a dark colour, for example. This is in conflict with the "natural" tendency to use green for "OK", and red for "not OK".

To clarify the transition between two colours (especially two colours close to each other) and the ambiguity that causes, use "redundant methods of data representation". The difference between two tints of blue might not be visible until you draw a line between them to stress the transition. Human cognitive processing is very sensitive for contour information. However, do not overdo it; when the difference between two colours is obvious, you do not need an edge to further clarify the transition.

5.3.3 "Nice Drawings of Graphs are Computationally Hard"

In [7] a discussion is given about what makes a drawing good. The article focuses on techniques for drawing graphs, but some general thoughts on the subject are given, too.

"We say that a diagram is readable, if its meaning is easily captured by the way it is drawn. The pictorial representation shall focus our view to the more important parts of the drawn object and shall illustrate its global structure. This, however, is vague and depends on various features (...)."

The problem is: how can you translate a representation of a graph into a nice drawing? What are the criteria to distinguish a good drawing from a bad one? How can we measure the quality of a drawing of a graph? These criteria cannot be formalised and checkable by an algorithm, and cannot be captured in formal terms. So, the problem is to approximate an unknown goal: how to draw a graph and how to draw it nicely?

The article approaches the problem by using "graph embeddings", a mathematical model to compare features of graphs. A drawing is "good", if the lower and upper bounds of some formal parameters coincide up to some constant factor; a drawing is "nice" if it is sharp for the parameter, and is best possible. A set of parameters of a drawing is defined, like the area used, the total edge length, the number of crossing edges, and more.

However, the article concludes: if you use *this* parameter to measure the quality of a drawing, an algorithm for constructing such a drawing is feasible. However, if your definition of quality uses *that* parameter, constructing such an algorithm may become difficult.

The article tries to solve a specific problem in visualisation, by trying to define the parameters of the quality of this specific visualisation, and then using mathematics to come to a solution. This is, however, both not generally useful, and in my opinion an impossible task since "good" is always a subjective measure.

5.3.4 "A Graphical Representation of the Prolog Programmer's Knowledge"

A visualisation of Prolog programs is presented in [8]. The approach taken to building a good visualisation is to analyse the object to visualise step by step, and finding a visualisation for every part you come across. Note that this resembles object-oriented analysis.

First, they identify a program as a black box. So, visualise the entire program as a rectangle.

Second, a Prolog program has four possible ways to do input/output, so draw four arrows to and from the rectangle, and label them corresponding to the four input/output possibilities.

Third, internally a Prolog program works with deductions, that are essentially implemented as searching through a database of "rules" with possible matches. So, inside the rectangle are "rules" to which the four input/output possibilities are led from the "external" arrows.

So, they quite literally transform the items found inside a Prolog program into objects that can be visualised (rectangles, arrows, etc.). However, their article doesn't consider the use of colours; it does some "animation" by simply placing multiple frames next to each other, not real animation. And, since their article is specific to the Prolog language, it is not very general, but the steps leading to their result are interesting.

5.3.5 "Audiovisuelle Animation von verteilten Algorithmen und Kommunikationsprotokollen"

In [16] visualisations are presented for various distributed algorithms. The difficulty with distributed algorithms compared to sequential algorithms is that in the latter case, you can define a set of states; the algorithm will make transitions between the states. With distributed algorithms, there are multiple processes that communicate by sending some kind of messages. The processes are independent, so the algorithm becomes non-deterministic.

Concerning visualisation, next to the distributed algorithm itself, the communication between the processes has to be visualised in order to give a complete overview of the algorithm.

Some more general issues are discussed as well. When creating a visualisation, the following steps will be taken. You start with an algorithm, or an implementation of the algorithm. The first step is to understand the algorithm. With the understanding, you must make a "mental image" of the algorithm. This image can then be drawn on a screen by using visualisation tools.

For visualising Transaction Processing, two *views* are used. The first is a time series diagram, showing the messages between the elements that are sent in time. Figure 12 through Figure 15 are examples of such diagrams. The second is a graph containing the elements accessed during the transaction, and their actions during two-phase commitment. The elements are coloured according to their actions, for example red for an element that cannot commit.

5.4 Summary

To show how algorithms function, and to show that they do so, visualisations can be used. However, the design of such a visualisation is a hard task. This chapter described general techniques that improve a design, and more specific techniques that can be used for specific algorithms. However, there are no set guidelines for creating good visualisations.

Many articles and books give either very specific examples of visualisations, that are hardly useful for finding general guidelines. Or they give very general but vague ideas about things to do, and things to avoid. It can be concluded that no general guidelines can exist, since every task is different, and every audience is different. A visualisation that has worked well in one place for a specific case, may not be a good when used elsewhere, or with altered objectives.

For the visualisation in this graduate project, the general ideas are useful, but the more specific texts are either on another subject, or have another purpose. [16] is the only article that discusses visualisation of Transaction Processing, but not very elaborately. Sections 7.2 through 7.4 describe how the specific algorithms for this graduate projects can be visualised, and Section 7.5 discusses which of the ideas presented above can actually be used.

Chapter 6 describes how a simulation of an ATM network was designed and realised; Chapter 7 describes the design and realisation of a visualisation of such a network.

6 Simulation of an ATM Network

The information from the previous chapters is combined in a simulation of an ATM network and a visualisation of such a network. The simulation offers the functionality of an ATM network, as described in Chapter 2, and was realised as a distributed application, as described in Chapter 3. It can be addressed using transaction processing, as described in Chapter 4. The visualisation gives an overview of the simulated network, and uses the ideas presented in Chapter 5; it is described in Chapter 7.

6.1 Introduction

To test applications that make use of ATM networks, two options are available. You could use a real network; however, this is both expensive, and inflexible. The other option is to use a simulation of such a network. A realisation in software can be easily tuned, and should not be as expensive as an ATM network realised in hardware.

When a simulation is used, the software making use of the network can be tested in every extreme situation. In a real network, creating such circumstances can be difficult, but for a piece of software this poses no problem. Its functionality can be modified any time.

After the application using the simulated network has been thoroughly tested, the transition to hardware should be trivial. Since the interface to the simulation and to the hardware are the same, no changes should have to be made to the application.

As said in the introduction, such a simulation was constructed during this graduate project. The simulation offers the functionality of an ATM network, as far as connection management is concerned, and can be addressed in a transactional context.

Next to the simulation, a visualisation was built that presents a graphical overview of the simulated network. The visualisation shows the layout of the network, and the result of performing operations on it.

Figure 16 shows the elements of the system and their relation. A client (connection management) issues calls on the simulation, as described in this chapter. These calls are intercepted by a filtering mechanism and appropriate calls are sent to the visualisation. The filtering mechanism and the visualisation are described in Chapter 7.

This chapter describes the design and realisation of the simulation, as well as details concerning the implementation of Quality of Service, statistics and transactional properties. Some technical details are listed in Appendix B.

6.2 Analysis of the Simulation

The functionality of the simulation is that of a collection of ATM switches, as described in Section 2.4. This functionality includes the handling of VP and VC branches, the status of the switch and its ports, and retrieving of information. These operations are related to connection management; this means many aspects of ATM networks are not simulated. No data is transported, for example, so everything concerning cells is irrelevant.

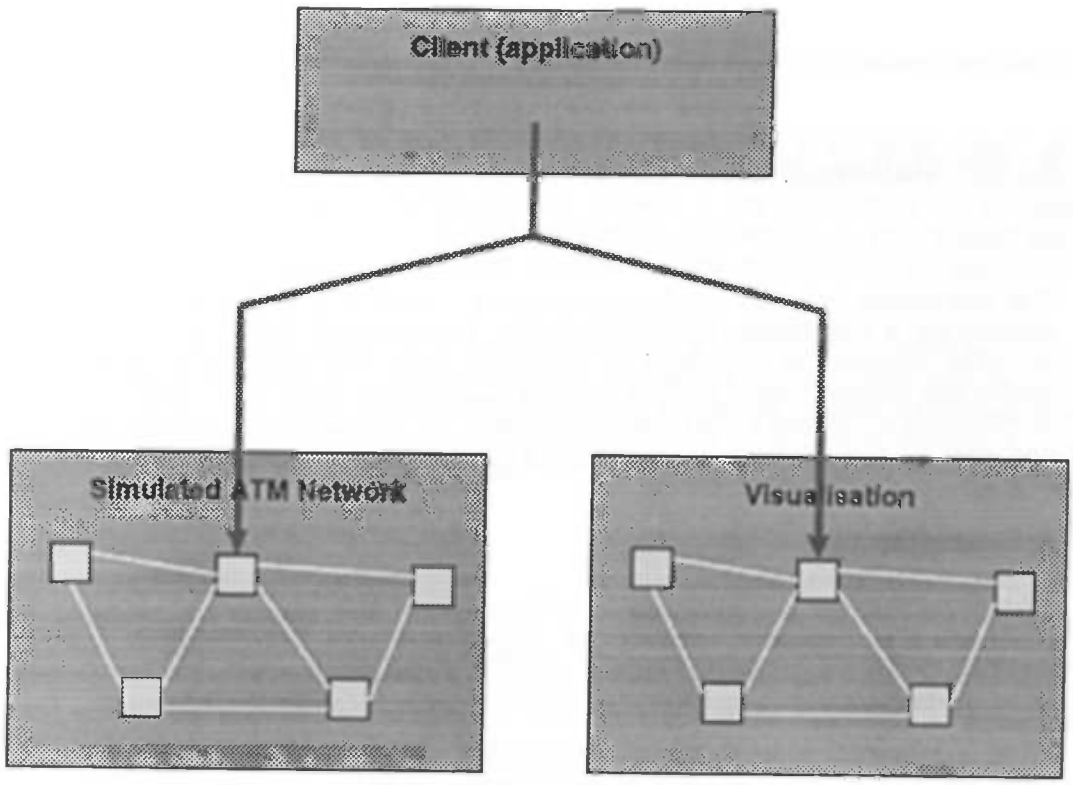

Figure 16 - Interaction of the software components

An object-oriented analysis of an ATM network yields the following list of objects present in such a network. Their relation is shown in Figure 17.

- Switches
- Ports on these switches
- Physical lines connecting the switches through their ports
- VC and VP branches

The *Switch* objects deal with the handling of VC and VP branches that can be added, deleted and modified in the connection tables inside the switch. To offer its functionality, every switch must store information concerning its state:

- a table of ports with their statuses and the bandwidth currently in use;
- the switch status;
- the VP connection table, containing the VP branches, the QoS parameters of every branch, and the time the branch was created for obtaining statistical information (see Section 6.5);
- the VC connection table, containing information similar to that in the VP table.

Every switch has a number of *Ports*; this number is assumed to be even. Every port has a status, and its input and output ports each have a bandwidth that is used for reacting adequately to QoS parameters. Ports are too small to be realised as objects themselves, so they are stored as entries in a table in the switches.

The *physical lines* are assumed to have enough bandwidth to support the connections established on the ports on either end of the line. Since connection management in ATM only deals with the setup of branches within switches, not with the lines between the switches, the lines are irrelevant to the simulation. They are only used in the visualisation, as described in Chapter 7, and by connection management itself (to determine which branches have to be set up).

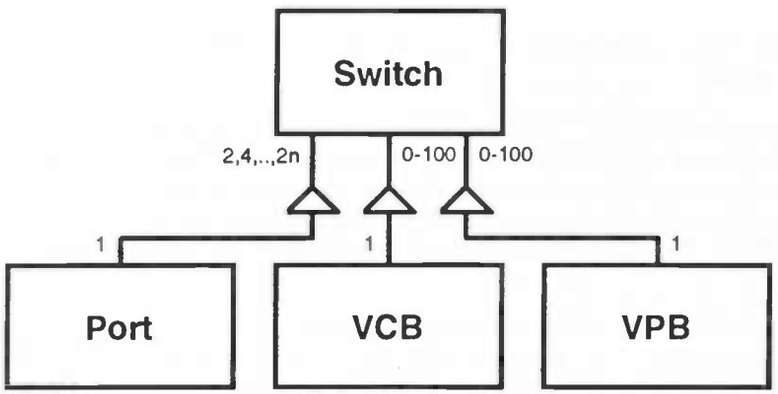

Figure 17 - Objects in the simulation

The *VC* and *VP branches* are identified by their input and output VCI/VPI values, and the input and output ports. No restrictions are laid upon the actual VCI and VPI values[3]. The input and output port numbers are used by the switch to check if sufficient bandwidth is available on the ports so the branch may be created. The branches will not be used for transporting data; the simulation does not need to implement that functionality. Therefore, branches are kept as entries in a table in every switch.

6.3 Realisation of the Simulation

The simulation is realised as a server in a DPE. This server offers the functionality of all switches in the simulated network. It can be addressed using CORBA calls, as described in Chapter 3. However, many details concerning CORBA and Transaction Processing are hidden, since the ACTranS project uses a template to build CORBA servers; only the functionality of the interface needs to be implemented.

Every object created with this template has two interfaces, as shown in Figure 18. Next to the interface to manage the switch, every switch object has a transactional interface. On this interface the five Resource calls can be performed that are described in Section 4.5.

The template also provides a way to realise multiple switches. During run-time, switches

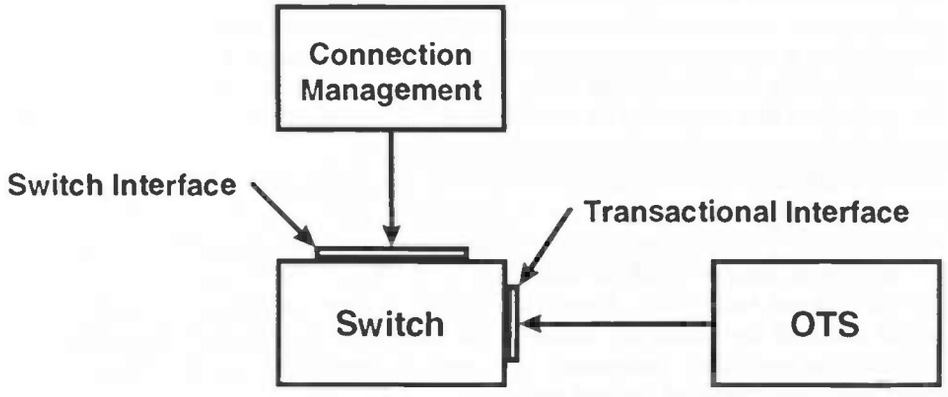

Figure 18 - Interfaces to the Switch server

can be created and removed. When a switch is created, it is given a name for

[3] With hardware ATM switches, there may be limitations on the VCI and VPI values of connections, depending on the type of switch. Also, certain VCI and VPI values may be reserved and therefore unavailable.

identification; this is a character string to be used in the filtering mechanism described in the next chapter.

For the switches and ports, an extra state was introduced, next to the three states "up", "down" and "locked". In this fourth, intermediate, state, called "buggy", operations seem to be executed normally, but during 2PC (at the end of the transaction) the changes made will not be committed, causing the transaction to be rolled back. This "buggy" state can be used for testing the transactional properties of the switches and the application using it.

Every switch offers the same interface; this interface is described in Figure 19, along with its functionality. The interface has been defined in the ACTranS project. The functionality follows the GSMP specification [3] as closely as possible. However, the *ResetSwitch* operation does not exist in GSMP, so it was implemented similarly to the *ResetPort* operation, extrapolated to all ports and the switch itself. The IDL file is given in Appendix A.

6.4 Quality of Service

Quality of Service is implemented by looking at the peak rate of the branches. Every port is given a certain input and output bandwidth. When branches are created, their peak rate will be subtracted from this value. It is possible that more bandwidth is requested than is available; in that case lower bandwidth is granted.

In general, for a multicast connection, at the input the largest bandwidth of the branches in the connection will be claimed; at the output the bandwidth of each branch is claimed seperately. VP and VC branches are completely independent of each other; this is enforced when adding or moving branches. For the following operations, QoS must be taken into consideration.

- For *AddVCBranch* and *AddVPBranch*, the bandwidth in use by the other branches in the same multicast connection is determined; this bandwidth is released at the input. How much bandwidth must be reclaimed is determined by looking at the requested bandwidth and the bandwidth in use by the other branches; this is limited by the bandwidth available at the input and output port. The bandwidth is then claimed at the input and output.
- With *DelVCBranch* and *DelVPBranch*, bandwidth must be released. At the input, the bandwidth of the other branches in the multicast connection is determined; if this is less than the bandwidth in use by the branch to be deleted, the difference is released. At the output, the bandwidth of the branch to delete is released.
- The QoS of a branch will change with *MoveVCBranch* and *MoveVPBranch* if the new output port has less bandwidth than the old output port. If the bandwidth is reduced, the largest of the branches in the multicast connection will be claimed at the input, and the new bandwidth will be claimed at the output.
- For *ChangeVCQoS* and *ChangeVPQoS*, the old bandwidth is released, and after that the actions taken are similar to those taken when adding a new branch.
- When resetting a switch with *ResetSwitch*, or a port with *ResetPort*, the bandwidth of the branches deleted must be released. With *ResetSwitch*, all input and output port bandwidths are reset; with *ResetPort*, the input port bandwidth is reset, and at the output ports of the branches deleted, the bandwidth is released. Note that multicast connections are either completely removed or untouched, so no checks have to be made with branches that are not removed.
- When a switch or port status is set to "up" using *ChangeSwitchStatus* or *ChangePortStatus*, branches will be removed. QoS considerations are equal to those with *ResetSwitch* or *ResetPort*.

Nr.	*Operation Name*	Functionality
1.	*AddVCBranch*	Add a VC branch on the switch, with specified QoS. The QoS granted is returned and may be less than requested.
2.	*AddVPBranch*	Add a VP branch on the switch, with specified QoS. The QoS granted is returned and may be less than requested.
3.	*DelVCBranch*	Delete a VC branch from the switch. If the VC branch is not present in the VCB table, the operation has no effect; otherwise, the branch is removed and the bandwidth it used is returned.
4.	*DelVPBranch*	Delete a VP branch from the switch. If the VP branch is not present in the VPB table, the operation has no effect; otherwise, the branch is removed and the bandwidth it used is returned.
5.	*MoveVCBranch*	Move a VC branch, that is, change its output parameters.
6.	*MoveVPBranch*	Move a VP branch, that is, change its output parameters.
7.	*ChangeVCQoS*	Change the QoS parameters of a VC branch.
8.	*ChangeVPQoS*	Change the QoS parameters of a VP branch.
9.	*GetAllVCBranches*	Retrieve the entire VC connection table from the switch.
10.	*GetAllVPBranches*	Retrieve the entire VP connection table from the switch.
11.	*ResetSwitch*	Reset the switch: remove all VC and VP branches, reset the switch status to "down", reset all port statuses to "down", and reset the bandwidth of all ports.
12.	*ResetPort*	Reset a single port on the switch: remove all VC and VP branches originating in the port, release the bandwidth they used, and reset the port status to "down".
13.	*ChangeSwitchStatus*	Change the switch status. If it is set to "up", all VC and VP branches are removed, and the port bandwidths are reset.
14.	*ChangePortStatus*	Change the status of a port. If it is set to "up", all VC and VP branches originating in the port are removed, and the bandwidth they used is released.
15.	*GetSwitchInfo*	Retrieve statistical information about the switch; see Section 6.5 for details.
16.	*GetPortInfo*	Retrieve statistical information about a port; see Section 6.5 for details.

Figure 19 - Operations on the simulation

6.5 Statistics

Since no real data is transferred, the switch will have to estimate the statistics for the *GetSwitchInfo* and *GetPortInfo* operations that retrieve these statistics. The statistics to be returned are, for both input and output, the number of octets (bytes) processed, the number of discards, the number of errors and the number of unknown protocol cells encountered.

The *GetSwitchInfo* operation will check all VC and VP branches set up in the switch. When a branch is created, it is given a timestamp. This timestamp can be compared with

the "current" time, to determine how long the branch has been up. This number multiplied with the average rate will give a reasonable estimate of the number of bytes processed at the input.

Every branch has associated QoS parameters. In these parameters, the loss rate and the error rate are given. These parameters are used to determine the input discards and input errors, as a fraction of the total number of octets processed.

This means that all octets are sent to an output port, except for the discards and the errors. So, the output number of bytes processed is determined as the input number of bytes processed with the discards and errors subtracted.

The other statistics are set to zero. At the output, it is assumed that there are no discards or errors, since the peak rate of the branches is used at setup. This means that high traffic will not result in discards or errors. Both "unknown protocol" field are set to zero, since their meaning has not yet been defined in the ACTranS project.

The *GetPortInfo* operation is implemented similarly to the *GetSwitchInfo* operation. For this operation, only branches ending or originating in the specified port are taken into consideration.

This method of estimating statistical information has some drawbacks. For example, when a branch is added, then removed, and statistics are requested after that, that branch will not be counted. Only branches that are in use at the time one of the *GetInfo* operations is performed will be taken into consideration.

Another example where things go wrong, is when the QoS parameters are renegotiated. Only the last bandwidth will be used since no history is kept for a branch.

Both problems can be solved by keeping statistical data in the switch when branches are modified. The statistics are computed as shown above, these values are stored, and the branches are given a new timestamp. When a *GetInfo* operation is performed, both these old values as well as the new values must be added for a better estimate.

6.6 Transactional Properties

An important feature of the simulation is its ability to be addressed in a transactional context. This means that a series of operations on one or more switches, such as the setup of an end-to-end connection, can be performed within one transaction.

To guarantee the transactional properties as described in Section 4.2, the switches must be registered as Resource objects, so the five operations listed in Section 4.5.1 through Section 4.5.5 can be performed. Furthermore, these five operations must be implemented themselves.

When the first operation within a transaction is performed on a switch, it will register itself as Resource with the transaction manager. Also, to guarantee isolation, mutexes must be locked for the switch, so other operations will not change its state during the transaction.

For every switch, two copies of its data will be kept. Operations during a transaction will be performed on a "temporary" copy, so rollback can be done very easily: just discard the copy on which the changes were made. Commitment can be done just as easily: discard the unchanged copy, and use the changed version as permanent version.

Next to the "internal" state of the switch, a durable version must be kept, as described in Chapter 4. This is a file containing all information present in a switch, so the switch can be re-initialised from that file.

The five operations are implemented according to their description in Section 4.5.1 through Section 4.5.5. A flag is used to indicate whether the Resource received a *prepare* call.

- *prepare* will vote *VoteRollback* if no switch is associated with the Resource, or the Resource has received a *prepare* call before. Otherwise, if no data was changed, it will cast *VoteReadOnly*. If data was changed, the (changed) switch data and a recovery coordinator will be saved to a temporary file. If this succeeds, *VoteCommit* is cast and the "prepared" flag is set; if not *VoteRollback* is cast.
- *rollback* will remove the temporary file created during *prepare*, if the Resource was prepared. The changed copy of the internal data will be discarded. The "prepared" flag will be reset.
- *commit* will make a heuristic decision if no switch was associated with the Resource, or if the Resource was not prepared. Otherwise, the temporary file created during *prepare* will be copied to a permanent file, without the recovery coordinator. If this fails, an exception will be thrown. Otherwise, the changed version of the internal data will be used as permanent version, and the "prepared" flag will be reset.
- *commit_one_phase* will try to save the internal data to a permanent file. If this succeeds, the changed version of the internal data will be used as permanent version; if not the changed version will be discarded and an exception will be thrown stating the Resource has rolled back.
- *forget* will simply remove the temporary file.

These functions need to be able to save the state of a switch to file. Such a file contains the entire data of a switch. This means the timestamp, the switch status, the number of VP and VC branches, and the number of ports are written. After that, for every VP branch the branch itself, its QoS, and its timestamp are written; for every VC branch the same information is written. For every port, the port status, and the input and output bandwidth left are written.

6.7 Summary

This chapter described the first part of the software built during this graduate project: the simulation of an ATM network. Its design and realisation were described, as well as details concerning the implementation of Quality of Service, statistics and transactional properties. The next chapter describes the other parts of the software: the visualisation of an ATM network, and the filtering mechanism that can be used to link the simulation to the visualisation.

7 Visualisation of an ATM Network

Next to the simulation, as described in the previous chapter, a visualisation of an ATM network was realised for this graduate project. This visualisation was constructed using the ideas found in literature, described in Chapter 5. Its design and realisation are described in this chapter.

7.1 Introduction

As described in Chapter 5, a visualisation can be used to gain insight in the operation of algorithms. A visualisation of an ATM network can show the layout of that network, and the results of operations on the network; such a visualisation was built during this graduate project, to function next to the simulation described in the previous chapter.

This chapter first describes which general visualisation techniques can be used for visualising Transaction Processing and Connection Management in ATM networks, in Sections 7.2 through 7.5. After that, the design and realisation of the visualisation is described in Sections 7.6 through 7.9. Some technical details are presented in Appendix B.

To link the visualisation to the simulation, that is, to let the visualisation reflect the network that is simulated, a filtering mechanism can be used that intercepts the calls to the simulation, and sends appropriate calls to the visualisation. This is described in Section 7.10.

7.2 Visualisation of Transaction Processing

In transaction processing, two algorithms can be visualised. The two-phase commitment protocol is the algorithm used to ensure either all changes made within a transaction are committed, or all changes are rolled back. The interaction with the transaction manager concerns the order in which the calls between the TM and the other elements active in the transaction occur.

7.2.1 Visualisation of the Two-Phase Commitment Protocol

The *two-phase commitment protocol* is the method used by transaction managers, to ensure that either all changes to data are committed, or all changes are rolled back (see Chapter 4). This mainly means the order in which the calls described in Section 4.5.1 through Section 4.5.5 are issued.

When some data is modified, the new (temporary) state will be shown in a grey colour, for "unsure". During 2PC the changes will either be committed or rolled back; the colour will change accordingly. The colour will change to white (for "good") if the changes can be committed, or the colour will change to black (for "bad") if the changes have to be rolled back.

When all changes are committed, the colour will change to a permanent one (a bright colour, as suggested in Section 5.3); when all changes are rolled back, the black colour will disappear, just as the changes made will disappear.

An example of how and when the colours change is given in Section 7.4, when a scenario of using transaction processing in the setup of ATM connections is described.

7.2.2 Visualisation of the Interaction with the Transaction Manager

The *interaction with the transaction manager* can be visualised by tracking all calls between clients, servers and the transaction manager. When the client starts a transaction, the TM will be notified. Every resource used is registered with the TM. When the client decides to end the transaction, it notifies the TM, which will start the 2PC protocol.

A way to visualise these calls, is by creating a time series diagram, which plots the calls as arrows between the elements involved, against time. When a call is issued at a given time, it is drawn as an arrow from the source to the destination, labelled with the name of the call. Subsequent calls are drawn further along the time axis. Figure 12 through Figure 15 are examples of these diagrams.

7.3 Visualisation of an ATM Network

An ATM network is composed of switches and physical lines between the switches. The switches are composed of some sort of switching fabric in which virtual branches are set up, and of ports which are the endpoints of these branches. On the edges of the network, there are endterminals which are connected to switches, again by physical lines.

So, the straightforward way to visualise an ATM network is as follows. The switches in the network, with the lines between them, are placed on a surface. The switches can be shown as rectangles and the ports on them as smaller rectangles attached to the switches. The physical lines run between the ports; they can simply be drawn as lines between the small rectangles.

The status of the switches and the ports can be shown by their colour. If a switch or port is "up", it can be shown in green, or in a light colour; if it is "down" it can be shown in red, or in a dark colour.

When branches are set up, they are placed inside the switches as arrows running from the input port to the output port. Since many branches can be set up in one switch, even between the same two ports, this can become a mess quite easily.

Therefore, the use of colours is essential to distinguish one branch from another. For every branch set up a new colour will be chosen, with the exception of branches that are part of the same (multicast) connection. The setup of connections in ATM networks is performed one branch at a time, as described in Chapter 2; when two branches are set up that are part of the same connection, they will be shown in the same colour. This results in end-to-end connections with all branches in one colour.

To distinguish between VP and VC branches, different arrow shapes can be used. Since VPs "contain" VCs, VPs can be drawn as open arrows, and VCs can be drawn as closed arrows. Figure 20 gives an example of how a switch in an ATM network can be visualised.

7.4 Visualisation of TP in ATM

As discussed in Chapter 4, transaction processing can be used in ATM networks for setting up an end-to-end connection over many switches. When one or more of the switches cannot comply to the request made, the other switches must not add a branch, since these branches will not be used and might not be removed afterwards.

Setting up such a connection will then be carried out as follows. A transaction will be started before accessing the switches. Next, all switches are told to add a branch.

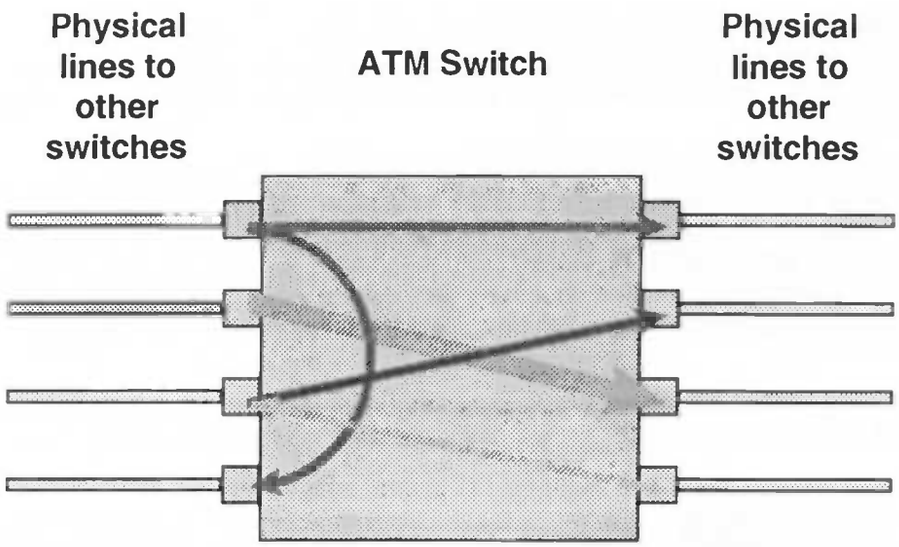

Figure 20 - Visualisation of an ATM switch

Finally, when the transaction is ended, the 2PC takes place, which decides whether all switches will commit the newly added branch, or all switches will roll back.

There are several scenarios that can occur here. As an example, the addition of a branch to one or more switches is used to show how the use of Transaction Processing in ATM networks can be visualised; Figure 21 and Figure 22 show the result.

1. The client starts a transaction.
2. The client issues a call to create new branches on one or more servers. The servers will register their resources with the TM. This will create an arrow in the switches for every branch added, all coloured grey, for "unknown outcome". After this, the client has two options:

   a) Rollback the transaction. The TM will issue a *rollback* call to every registered resource, which will undo the changes. On the display, the branches set up are made black, and will be removed after a short period of time.

   b) Commit the transaction. The TM will now start 2PC.

When 2PC is started, every registered resource will receive a *prepare* call from the TM. The resources will try to write the changes to the data to permanent storage. Every resource will respond with one of the following choices.

1. *VoteReadOnly*, if no data was changed. This will not occur when a branch is added, but will occur when data was retrieved, not modified. In this case, no branch was set up, so on the display nothing has to be changed.
2. *VoteCommit*, if the data can be saved to permanent storage. This is the normal case, in which branches can be set up without problems. When this vote is cast, the branch set up in grey will be changed to white, for "OK".
3. *VoteRollback*, if the data cannot be saved to permanent storage. This happens when there is for example some error with the switch, or some communications problem. The branch set up will be changed to black, for "not OK", and it will be removed after a short period of time.

The TM waits for the votes from all registered resources. There are two choices now:

1. If all resources responded with either *VoteReadOnly* or *VoteCommit*, everything went OK, so the TM will send a *commit* call to the resources that responded with

VoteCommit. The branches will change from white (set after the *prepare* call) to a permanent colour; which colour is chosen will depend on whether this is the only branch in a given connection, or a branch that is part of a larger connection (see above).

2. If one or more resources responded with *VoteRollback*, the TM will send a *rollback* call to all other registered resources. The other branches already set up will be changed from either white (if the resource already voted *VoteCommit*) or grey (if the resource didn't receive its *prepare* call yet) to black, for "not OK", and will be removed after a short period of time. If one resource doesn't reply at all (within a given timeout period), the same procedure will be followed. This happens if the switch can no longer be reached.

After 2PC, either all branches are removed, or all branches will have a permanent colour, indicating which branches are part of the same (possibly multicast) connection.

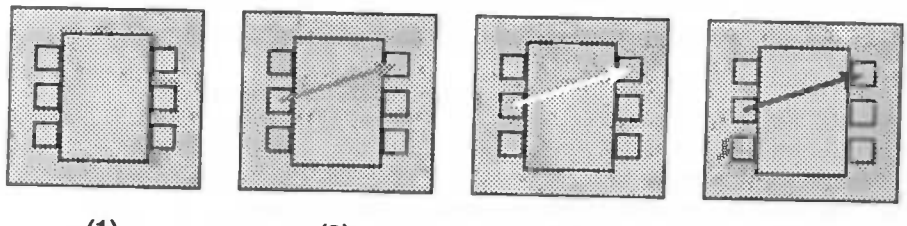

Figure 21 - Visualisation of TP in ATM : commitment

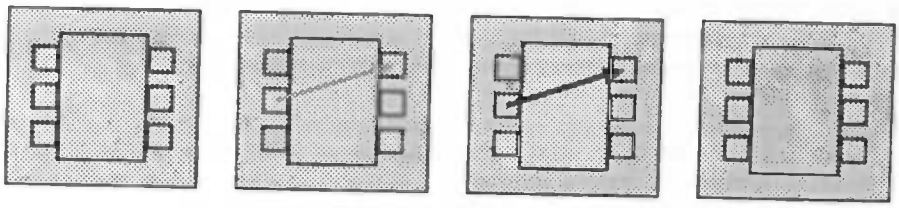

Figure 22 - Visualisation of TP in ATM : rollback

7.5 Application of Visualisation Techniques

The techniques and ideas presented in Chapter 5 have been useful in creating the visualisation. This section describes the application of these techniques.

Some of the ideas from [10] can be used. *Multiple views* can be created by zooming in on parts of the network, and observing the global behaviour in the normal display, and the specific behaviour of one switch in the zoomed display.

Highlighting the branch that will be added is difficult to do in real-time, since you cannot predict when it will be added. You can, however, create a thick or blinking arrow when it is created, and change it to a normal arrow after some time.

A *static history* can be created by "recording" all events, and using "play", "step" and "pause" buttons, etc. This way you can first view the events in real-time, and replay interesting parts afterwards.

The *choice of input data* is not the task of the visualisation, but the task of the user trying to show how transaction processing takes place in ATM connection management. The visualisation must perform the specified tasks in the best possible way, but the user must use it in a clear and correct way.

In the visualisation, no choice between *continuous* or *discrete* state changes has to be made, since no "moving" objects are present. VC and VP branches simply appear or disappear. The last point given in [10], the use of *multiple algorithms* next to each other, is not applicable, since the task of the visualisation is to show the operation of one defined combination of algorithms.

The ideas presented in [9] are more general, and mostly apply to the use of colours, the choice of which colours to use, etc.

Background colours will be grey or greyish. Important data items (most notably the VC and VP branches in switches) will be shown in bright colours.

The switches, shown as rectangles, should not be filled with bright colours, but a dim colour is more appropriate. The visualisation uses four colours for four different switch states: light green for "up", light red for "locked", darker red for "down", and yellow for "buggy" (an intermediate state between "OK" and "not OK").

When selecting colours for the branches, care must be taken to avoid colours that look like one of the four colours used for switch states. Colours close to light green, light or darker red, or yellow must not be chosen.

The various tips for using colours to show an order in value are not used, since this kind of data is not used in the visualisation. Also, red and green *are* used for distinguishing; the audience is limited and this will not be a problem.

Many other ideas were not used, in most cases because they were not applicable in this specific visualisation. For example, there are no multiple algorithms to show and compare; one algorithm for TP and one for ATM connection management is shown.

From the more specific texts, the "object-oriented" approach, used in the visualisation of Prolog program [8], is used in the visualisation too. The elements of an ATM network, and the operations on them are defined and visualised.

The ideas in Section 7.2 through Section 7.4 are not taken from literature, but were conceived while creating the visualisation. However, not all ideas were used. The interaction between the transaction manager and the clients and servers was not visualised. This would require a completely different type of visualisation, other than the view of the ATM network.

Another point is the actual choice of colours when the branches are created in the network, after 2PC. All branches that are part of the same (multicast) connection will have the same colour. Initially, the most logical choice would seem that the branch added will get the colour already used in the rest of the connection.

However, this poses a problem, since the branches are added in a random order. What colour should the newly created branch get when it is a branch connecting two already coloured but (so far) disjunct collections of branches?

The solution chosen, is to select an unused colour for every branch added, and use that colour for the other branches on the same connection, too. This means that the colour of the branches will change with every branch added to it. But, the colour will be the same for the entire connection after every addition.

7.6 Analysis of the Visualisation

The visualisation offers the user a graphical overview of the simulated network. Both static information, concerning switches, ports, and the lines between them, and dynamic

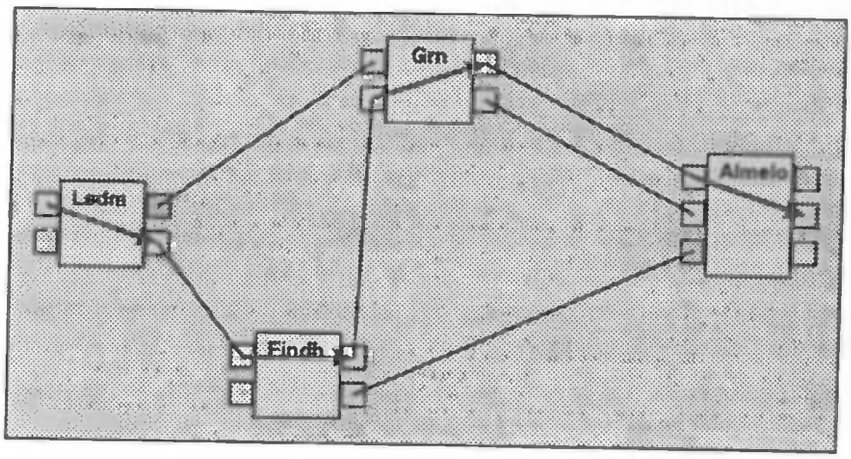

Figure 23 - Example display of the visualisation

information, concerning branches being set up and the status of the switches and port, is presented.

The switches are drawn as rectangles, with smaller rectangles on the left and right that represent the ports on the switch. The colour of these rectangles represents the status of switches and ports: green for "up", dark red for "down", light red for "locked", and yellow for "buggy". Figure 23 gives an example of what the visualisation presents.

Branches are represented as arrows drawn in the switches, pointing from their input port to their output port. VC branches are shown as "normal" arrows, and VP branches are shown as "open" arrows.

Next to presenting an overview of the network, the visualisation shows how the various algorithms used work. The following two algorithms are depicted.

- Transaction Processing will be made clear using the colours of the branches for the different "phases" of setting up the branches. During the transaction, branches will be shown in grey. When the transaction is ended, during the first phase of 2PC every branch will either be committed or rolled back; this is shown by changing the colour to white when committing, or black when rolling back. After branches are committed, their colour will change to a permanent, bright colour.
- Connection Management in ATM networks is shown implicitly. Setting up a connection is performed by instructing switches to create a branch; these branches are set up independently. After commitment of the transaction during which the branches were set up, they will be part of an end-to-end connection. This is shown by using the same colour for all branches in one connection. Also, multicast connections can be set up; these connections will also be coloured equally in all switches.

To accomplish this, the visualisation needs largely the same information as the simulation. The same objects are present: switches, ports, physical lines and branches. Their relation is shown in Figure 24.

The *Switch* object implements the functionality of switches, as far as the visualisation is concerned. This means every switch has VC and VP connection tables, a list of ports, and the switch status. Furthermore, information about the representation on the display is kept: the coordinates where the switch is located, its dimensions, an identification string (the name of the switch) and a "handle" to be able to manipulate the switch.

Every *Port* object represents a port on a switch; ther number of ports on a switch is assumed to be even. The status of the port is kept, along with information about the representation on the display: its coordinates and a handle.

The physical lines are implemented in one *Link* object, containing information about all lines. A table is kept, containing an entry for every line. A line is implemented as a 4-tuple (s1, p1, s2, p2) for a line from port p1 on switch s1, to port p2 on switch s2.

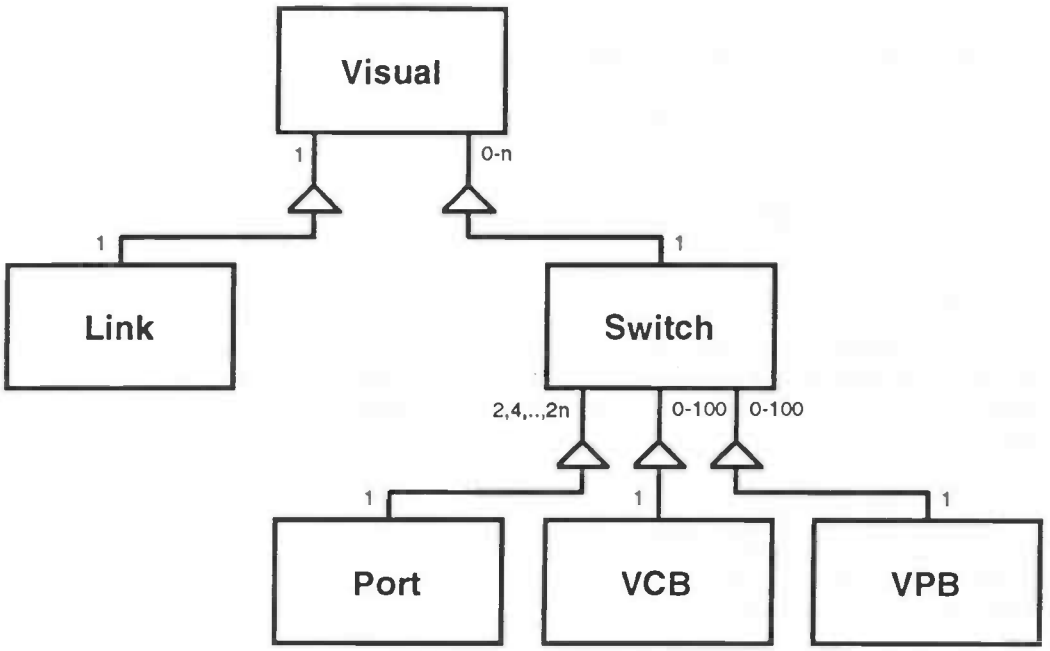

Figure 24 - Objects in the visualisation

Like in the simulation, VC and VP branches are not implemented as seperate objects. The branches are kept as entries in the connection tables in the switches, like in the simulation (see Chapter 6). Every entry contains the branch itself; the phase (see Section 7.8); if the colour is permanent, the colour of the branch; and a handle.

For the visualisation, an "overall" object was created. This *Visual* object contains a list of switches, a Link object, and a colour table for use in the propagation algorithm as described in Section 7.8.

Every object has two representations: one in the "internal" data, and a graphical representation on the *canvas* [18]. A canvas is a "drawing board" used for the graphical display of the visualisation; it reflects the internal data.

At initialisation, the network topology is read. First, the number of switches in the network is read. Then, for every switch, its (x, y) coordinates are read, the number of ports on the switch, and the name of the switch. For every switch read, a rectangle is drawn on the canvas with smaller squares on the sides for the ports. The first half of the ports are on the left, the second half on the right of the switch.

Second, the number of physical lines in read. Then, for every line, its parameters are read: s1 p1 s2 p2 for a line from port p1 on switch s1 to port p2 on switch s2. For every line read, a line is drawn on the canvas between the two squares that represent the right ports.

The visualisation is also a user interface to the simulation. Specific information about objects present on the display can be obtained using the mouse. For example, ports are normally only shown as rectangles. The port number can be obtained using the right mouse button. Information about switches, ports, lines and branches can be obtained this way.

The layout of the network is shown on the display, and this can be modified using the mouse. After a new layout has been constructed, it can be saved for future use. Algorithms exist to automatically place the nodes of a graph; such an algorithm was not used.

7.7 Realisation of the Visualisation

The interface to the visualisation is largely similar to that to the simulation. The operations are listed below, with their functionality, in Figure 25.

In general, the operations involve two actions: changing the data in the switch, and changing the representation on the display. The functionality of the operations is on one hand simpler than that of the operations on the simulation, and on the other hand more complex.

The ATM aspects are simpler, since many tests are no longer necessary. The filter only sends commands to the visualisation if the operation was successful on the simulation (see Section 7.10). Furthermore, the visualisation does not have to consider QoS or statistics.

However, the simulation does not have to deal with transaction processing in the implementation of the interface functions; that part is taken care of elsewhere. The visualisation must handle these aspects for displaying branches in different colours. And, most importantly, the visualisation must create the graphical output. This involves computing the location of switches, branches, etc. on the canvas.

All operations on the visualisation are performed on the "overall" Visual object. This means that the operations identify the switch on which the operation must be performed in the parameters to the operation. For example, if switch 3 receives an *AddVCBranch(VCB, QoSi, QoSo)* call, the visualisation will receive a *VisAddVCBranch(3, VCB, phase)* call.

7.8 Transaction Processing

The visualisation shows the operation of two algorithms: connection management in ATM networks, and the use of transaction processing. Connection management is shown implicitly, as described in Section 7.6.

Transaction processing is visualised as follows. During setup, the branches will be in one of the following *phases*: *initial* during the transaction, *committing* or *rolling back* during the first phase of 2PC, and *permanent* after commitment.

Initially, branches are coloured grey, since the outcome of the transaction is unknown. Committing branches will be coloured white, for "good", and branches rolling back will be coloured black, for "not good". When one or more of the branches cannot commit, all branches will be coloured black, and they will disappear (the transaction will have no effect). Otherwise, the branches will get a permanent colour.

All branches that are part of the same connection are given the same colour. This is accomplished with a *propagation* algorithm. The algorithm is called whenever a branch is given a permanent colour. This colour is then propagated to the other branches in the same connection.

The Visual object has a colour table that stores the colours to use for permanent colours, and the number of times that colour is in use. This colour table is read when the visualisation is initialised. When a permanent colour is needed, a colour that is not in use will be chosen. When such a colour is unavailable, the colour that is used less will be chosen.

When a branch is given a permanent colour, its "neighbours" are determined. The neighbours of branch **B** are

- in the same switch, the branches with the same input port, VPI and VCI as **B**;
- in the switch physically connected to the output port of **B**, all branches on the input port the physical connection is on, with input VCI and VPI equal to the output VCI and VPI of **B**;

Nr.	*Operation Name*	Functionality
1.	*VisAddVCBranch*	Add a VC branch to a switch, and to the display. The colour is chosen, depending on the *phase* (see Section 7.8).
2.	*VisAddVPBranch*	Add a VP branch to a switch, and to the display. The colour is chosen, depending on the *phase* (see Section 7.8).
3.	*VisDelVCBranch*	Delete a VC branch from a switch, and from the display.
4.	*VisDelVPBranch*	Delete a VC branch from a switch, and from the display.
5.	*VisMoveVCBranch*	Move a VC branch, that is, change its output parameters, both in a switch and on the display.
6.	*VisMoveVPBranch*	Move a VP branch, that is, change its output parameters, both in a switch and on the display.
7.	*VisChangeSwitchStatus*	Change the switch status, and the colour of the switch on the display.
8.	*VisChangePortStatus*	Change the status of a port, and its colour on the display.
9.	*VisResetSwitch*	Reset the switch: remove all VC and VP branches, reset the switch status to "down", reset all port statuses to "down", and change the colour of switch and ports on the display.
10.	*VisResetPort*	Reset a single port on the switch: remove all VC and VP branches originating in the port, reset the port status to "down", and change its colour on the display.

Figure 25 - Operations on the visualisation

- in the switch physically connected to the input port of **B**, the branch on the output port the physical connection is on, with output VCI and VPI equal to the input VCI and VPI of **B**.

The colour of the neighbours is changed, the administration in the colour table is updated, and the algorithm is called recursively on these branches. Note that with every branch added to the connection, the colour of the entire connection will change.

This algorithm is currently only implemented for *VisAddVCBranch*. The same algorithm can be used for VP branches. When a branch is deleted, the algorithm will not be called, leaving two disjunct trees of branches in the same colour.

The visualisation currently only shows the use of TP correctly when branches are added. So, when branches are deleted, they will not turn grey etc. Also, changing the status of switches or ports or resetting switches or ports in performed without TP consideration.

7.9 Possible Enhancements

Other possibilities to enhance the functionality of the visualisation include the following.

- Zooming in on switches. Normally, switches are shown as rectangles with smaller rectangles for the ports, and branches are set up as arrows in them. When many branches are added, single branches will no longer be visible; in a zoomed view this will be made clear. Furthermore, this allows the user to see and compare an overall

view of the entire network, with the current state of a single switch in that network. Note that zooming in on parts of the network requires both the normal and zoomed view to be updated when operations are performed on the network.
- In a zoomed view, multiple branches running between the same two ports, could be shown next to each other, instead of "on top of" each other. When a branch is added or removed, this would mean that all branches between the same two ports have to be redrawn.
- The Quality of Service of the branches is not shown. This could be done by changing the width of the arrows to represent the bandwidth. Note that this poses new problems, like normalisation (how wide should an arrow of a given bandwidth be?).
- The colour of the switch is used for its status. The colour could also show the load of the switch, for example by using the total bandwidth of all branches as a measure.
- Performing operations on the simulation from the visualisation. Possible operations include resetting ports or switches, and changing their status. This means a link in the other direction, from visualisation to simulation must be laid.
- All branches of an end-to-end connection are shown in the same colour. The physical lines between the switches could be coloured as well, to form an uninterrupted line from the source to the destination.
- Static information could be added by using the mouse. Switches could be added by selecting their location; physical lines between the switches could be added by clicking on the first port, dragging to the second port, and releasing the mouse button there.

7.10 Filtering

7.10.1 Overview

For the results of operations on the simulation to be shown in the visualisation, the two must be linked. To do this, the following solutions exist.

- The simulation can send commands to the visualisation.
- The visualisation can frequently retrieve the state of the simulation.
- The operations on the simulation can be intercepted and sent to both the simulation and the visualisation.

The first solution is efficient, but not very flexible: this cannot be used when the simulation is replaced by a hardware ATM network. The second solution is much less efficient, since for a reasonably quick response to changes in the simulation, the visualisation must retrieve the state of the simulation quite often, which involves CORBA requests. However, this solution can still be used after transition to hardware, since the visualisation can still retrieve the state of the switches then.

The last solution is both efficient and flexible. Operations on the visualisation will only be sent when necessary (only when operations are sent to the simulation), and the solution can still be used after transition to hardware. To intercept operations on the simulation, a filtering mechanism must be used.

7.10.2 Filtering

Such a filter is able to monitor all calls to and from clients and servers, allowing actions to be taken when certain calls are performed; this can be used to send commands to the visualisation. For example, if a branch is added on a switch using an *AddVCBranch* call, the corresponding *VisAddVCBranch* call must be performed on the appropriate switch in the visualisation, to add an arrow on the display.

With the interaction between a client and a server, there are four points where actions can be taken, as shown in Figure 26.

- the *request*, the sending of a call from the client to the server;

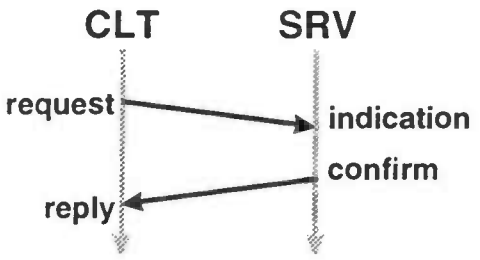

Figure 26 - Terminology in interaction between client and server

- the *indication*, the arrival of that call at the server;
- the *confirm*, the sending of the answer from the server to the client;
- the *reply*, the arrival of the answer at the client.

A *smart proxy* is a feature of *Orbix*, the CORBA-compliant ORB used in the ACTranS project. It can intercept all calls at the four points listed here, along with their parameters. So, not only calls from connection management are monitored; calls from the Transaction Manager like *prepare* and *commit* can be intercepted too. This allows the visualisation to display the steps taken during 2PC, as described in Section 7.8.

Not every call to the simulation has its counterpart in the visualisation. For example, *ChangeVCQoS* and *ChangeVPQoS* only change the Quality of Service, and the QoS is not visualised. *GetSwitchInfo* and *GetPortInfo* only read information, and do not change the state of a switch. This means a mapping must be made between "all" calls, and "useful" calls.

The calls to the visualisation are made by sending a script to a named pipe. This pipe will be read from by the *sender* of the visualisation. This is a small program that reads the pipe and sends the commands to the visualisation. This approach was taken, since the visualisation could not be a CORBA server; that would have been much simpler since the simulation is a server and the filter can perform CORBA calls. Unfortunately, the combination of being a CORBA server and using X Windows proved to be unstable.

7.10.3 Mapping

Most operations must be checked at the *confirm* phase of the call. Here, the result can be monitored, stating whether the operation was executed successfully or not. When an operation was completed successfully, the visualisation must be notified, when not successful, the visualisation does not need to know a command was sent.

Operations like *AddVCBranch*, *DelVCBranch*, and *MoveVCBranch* will be reacted on at the *confirm* phase of the call. The visualisation will be sent a *VisAddVCBranch* call in the "initial" state. The call sent to the visualisation will be queued fo further handling.

Since the visualisation currently only reacts correctly to TP with the *AddVCBranch* operations, the other calls will only be sent in the "permanent" phase, that is, after commitment.

The other operations, both *ChangeQoS* operations and both *GetInfo* operations, will not induce calls to the visualisation, since no data is changed.

When a *prepare* call is done on a switch in the simulation, all queued calls will be checked, and the operations on that particular switch will be sent again. This time the call will be sent in the "committing" or "rollingback" state, depending on the result of the prepare call (*VoteCommit* or *VoteRollback*).

When a *commit* call is done on a switch in the simulation, all queued calls will be checked again, and the operations on that particular switch will be sent again, in the "permanent" state.

When a *rollback* call is done on a switch in the simulation, all queued calls will be checked, and they will be sent again, in the "rollingback" state.

When a *commit_one_phase* call is done on a switch in the simulation, and the switch can commit, calls in the "committing" phase will be sent, and calls in the "permanent" phase some time after that. If the switch cannot commit, the calls will be sent in the "rollingback" state.

The *forget* call will not induce calls on the visualisation. The visualisation does not react correctly to heuristic decisions.

The queue will be emptied at the first *register_resource* call. This is an outgoing call from the simulation to the OTS. The first *register_resource* call means that a new transaction has begun, so the old information in the queue is no longer needed.

7.10.4 Realisation

Due to problems with the software, the filtering mechanism has not been implemented. At first, another feature of Orbix, called a *filter*, was tried. This filter should be able to retrieve the following information about an operation: the operation name, the object on which it is performed (the switch name), the input parameters, the output parameters, and the return value.

However, the operation name and the object name could be retrieved, but not the arguments or return value. These values are necessary for the visualisation. It appears that the smart proxy *can* be used. This solution could unfortunately not be implemented in time.

7.11 Summary

This chapter described the visualisation of an ATM network that was built during this graduate project. First, the techniques to be used were discussed. Next, the design and realisation on the actual visualisation were presented, as well as details concerning the implementation of transaction processing, and possible enhancements to the visualisation.

Finally, the filtering mechanism that can link the simulation to the visualisation was described. Unfortunately, during this graduate project the filtering mechanism could not be implemented in time. During use, many more ideas to improve the software will be found, especially ideas concerning the visualisation and its user interface.

8 Conclusions & Recommendations

8.1 Evaluation of the Simulation

The simulation, as described in Chapter 6, offers the functionality of an ATM network, that can be addressed within a transaction. The simulation will be used in the ACTranS project for testing a connection management application.

During this graduate project, no real testing has been done to see if the simulation offers the functionality needed. Since some aspects of its functionality have not been defined clearly, choices were made that can turn out to be incorrect, but in general those choices can be altered without much problems.

8.2 Evaluation of the Visualisation

The visualisation offers an overview of the simulated network, and shows the result of operations performed on it, as described in Chapter 7. The switches are shown with the ports on them, and when branches are created they will appear as arrows on the switches. The visualisation will be used in the ACTranS project in cooperation with the simulation, to test the connection management application and see what effects the operations performed by that application.

Unfortunately, during this graduate project the link between the simulation could not be laid, due to problems with the DPE software. A possible solution is given, but that was not realised.

8.3 Conclusions

Building a simulation of an ATM network is a good alternative to using a real ATM network. A realisation in software can easily be tuned, and is much less expensive.

The use of a visualisation greatly enhances the understanding of algorithms. The use of transaction processing, and the setup of connections in ATM are visualised. The visualisation can be used to get an overview of the simulated network.

8.4 Recommendations

The visualisation can be enhanced on many aspects, as described in Section 7.9. The current version is usable, but during use many more ideas will present themselves for improvement.

The simulation and visualisation are linked together in an ad-hoc way, and can be made to work. However, it would be better if the visualisation would be addressable using CORBA, too. Perhaps methods exist to make the combination of X Windows graphics with CORBA stable.

The system was built using CORBA, and X Windows on Unix. However, other middleware software, graphics software and operating systems are in use. The system could be ported to other environments; multiple ways for accessing it could be made; or other methods of creating graphical output can be created.

The visualisation uses an initialisation file for the network topology it presents. It would be more flexible to be able to add switches, lines between the switches, etc. during run-time instead.

Stable storage for transaction processing is currently provided by files on the Unix file system. Using the XA interface, a database like Oracle can be used as stable storage. Oracle has proven to be stable and robust.

9 References

[1] "ATM Switching Systems", Thomas M. Chen & Stephen S. Liu, Artech House 1995.

[2] "Definitions of Managed Objects for ATM Management Version 8.0 using SMIv2", RFC 1695, http://www.cis.ohio-state.edu/htbin/rfc/rfc1695.html.

[3] "General Switch Management Protocol Specification Version 1.1", RFC ????.

[4] "ACTranS deliverable D1d (update)", ACTranS project, 1996.

[5] "Visualization of Scientific Parallel Programs", Gerald Tomas & Christoph W. Ueberhuber, Lecture Notes in Computer Science 771, Springer-Verlag 1994.

[6] "Visualization in Human-Computer Interaction", P. Gorny & M.J. Tauber (eds.), Lecture Notes in Computer Science 439, Springer-Verlag 1990.

[7] "Nice Drawings of Graphs are Computationally Hard", Franz J. Brandenburg, in [6].

[8] "A Graphical Representation of the Prolog Programmer's Knowledge", J. Polak & S.P. Guest, in [6].

[9] "Envisioning Information", Edward R. Tufte, Graphics Press, 1990.

[10] "Color and Sound in Algorithm Animation", Marc H. Brown and John Hershberger, DEC SRC Report 76a, 1991, ftp://gatekeeper.dec.com/pub/DEC/SRC/research_reports/SRC-076a.ps.Z

[11] "The 1993 SRC Algorithm Animation Festival", Marc H. Brown, DEC SRC Report 126, 1994, ftp://gatekeeper.dec.com/pub/DEC/SRC/research_reports/SRC-126.ps.Z

[12] "CORBA Services: Common Object Services Specification", OMG, 1995-1996, http://www.omg.org/library/corbserv.htm.

[13] "Computernetwerken", Open Universiteit, 1988.

[14] "Object Transaction Service", OMG document 94.8.4, 1994, ftp://www.omg.org/pub/archives/docs/1994/94-08-04.ps (latest version of the OTS is in ftp://www.omg.org/pub/docs/formal/97-02-15.ps).

[15] "A Principled Taxonomy of Software Visualization", B.A. Price, R.M. Baecker, I.S. Small, Journal of Visual Languages and Computing 4(3):211-266, http://www-cs.open.ac.uk/~doc/jvlc/JVLC-Body.html.

[16] "Audiovisuelle Animation von verteilten Algorithmen und Kommunikationsprotokollen", Arnulf Mester und Peter Herrmann (Hrsg.), Endbericht der Projektgruppe 225, Universität Dortmund Fachbereich Informatik, 1994, http://ls4-www.informatik.uni-dortmund.de/MA/am/Pub/index.html.

[17] "Asynchronous Transfer Mode: ATM Architecture and Implementation", James Martin, Kathleen Kavanagh Chapman, Joe Leben, Prentice Hall PTR, 1997.

[18] "Embedded Tk", D. Richard Hipp, http://users.vnet.net/drh/ET.html.

[19] "Multimedia Services in Open Distributed Telecommunications Environments", Peter Leydekkers, CTIT Ph.D-thesis series No. 97-12, 1997.

[20] "The Common Object Request Broker: Architecture and Specification, Revision 2.0", OMG document 97.2.25, 1997.

[21] "The ATM Forum", http://www.atmforum.com.

Appendix A IDL Interface to an ATM Switch

```
// Switch.idl
//
// Authors : David MACIA, Pierre HANOUNE
//
// VERSION 0.2beta
//
// Date : 1 august 1996
//
// Based on KPN proposition, GSMP, MIB-II (RFC 1213) and
// ATM MIB-II extension (RFC 1695)
//
// Our switch is a crossconnect (no signalisation) so we can have :
//    - VP crossconnect
//    - VC crossconnect
//
// QoS : we propose (for the moment) the following QoS parameters
//    - averageRate   |
//    - peakRate      | because they are supported by the MIB-II (and all
//    - burstSize     | the MIBs)
//
// and some optional parameters: errorRatio, lossRatio, delay,
// delayVariation
//
// ----------------------------------------------------------------

#include "Transactions.idl"

// Structures and constants
// ----------------------------------------------------------------

#define MAXVCCC 100
#define MAXVPCC 100

typedef long index ;

enum SwStatus {
  up,       // device is operationnal
  down,     // device is NOT operationnal
  locked,   // device is locked for configuration purpose
  buggy     // device is in the 'intermediate' buggy state
};

enum Answer {
  ok,
  failed
};

struct Result {
  Answer ack ;
  long errno ;      // error list to be submited
  string comment ;
};

struct QoS {
// mandatory
  long averageRate ;
  long peakRate ;
  long burstSize ;
// optionnal
  long errorRatio ;
  long lossRatio ;
```

```
    long delay ;
    long delayVariation ;
};

struct VC {
    index vci ;
    index vpi ;
    index portnum ;
};

struct VP{
    index vpi ;
    index portnum ;
};

struct VCBranch{
    VC inVC ;
    VC outVC ;
};

struct VPBranch{
    VP inVP ;
    VP outVP ;
};

// for the moment info is based on the MIB-II specification
struct info{
    long inOctets ;
    long inDiscards ;
    long inErrors ;
    long inUnknownProtos ;
    long outOctets ;
    long outDiscards ;
    long outErrors ;
    long outUnknownProtos ;
};

struct VCConnectTableEntry {
    SwStatus VCCCStatus ;
    VCBranch VCB ;
};

struct VPConnectTableEntry {
    SwStatus VPCCStatus ;
    VPBranch VPB ;
};

typedef VCConnectTableEntry VCConnectTable[MAXVCCC];

typedef VPConnectTableEntry VPConnectTable[MAXVPCC];


// Interfaces :
// ------------

interface Switch : CosTransactions::TransactionalObject
{
    Result ResetSwitch();

    Result ChangeSwitchStatus(in SwStatus S);
    Result GetSwitchInfo(out info switchinfo);

    Result AddVCBranch(in VCBranch VCB, in QoS Qi, out QoS Qo);
    Result AddVPBranch(in VPBranch VPB, in QoS Qi, out QoS Qo);

    Result DelVCBranch(in VCBranch VCB);
    Result DelVPBranch(in VPBranch VPB);

    Result MoveVCBranch(in VCBranch VCBi, in VCBranch VCBo, out QoS Q);
    Result MoveVPBranch(in VPBranch VPBi, in VPBranch VPBo, out QoS Q);

    Result ChangeVCQoS(in VCBranch VCB, in QoS Qi, out QoS Qo);
```

```
    Result ChangeVPQoS(in VPBranch VPB, in QoS Qi, out QoS Qo);

    Result GetAllVCBranches(out VCConnectTable VCCC);
    Result GetAllVPBranches(out VPConnectTable VPCC);

    Result ResetPort(in long PortNumber);

    Result ChangePortStatus(in long PortNumber, in SwStatus S);
    Result GetPortInfo(in long PortNumber, out info portinfo);
};
```

Appendix B Technical Specifications

B.1 Simulation

Operating System	Solaris v2.5.1
Language	C++ (Sun C++ compiler v4.0)
Middleware	Orbix v2.1
Tools	Bull OTS v1.1.1.4
LOC	about 2000 (newly written code)

Figure 27 - Specifications of the simulation

B.2 Visualisation

Operating System	Solaris v2.5.1
Language	C++ (Sun C++ compiler v4.0), Tcl v7.6/Tk v4.2
Tools	Embeddable Tk v1.7, Visual Tcl v1.07
LOC	2568

Figure 28 - Specifications of the visualisation