Christophe Andrey

Agent-Based Network Management

Semester Project

June 1998

Abstract

This semester project aimed at investigating some possibilities and some strategies of network management by intelligent agents. Especially, we focused on connection admission control for multimedia sessions (like videoconferences), by collaborative agents coordinated by market-based mechanisms.

We first present an evaluation of platforms available today and supporting intelligent agents. Out of them, we selected Objectspace's Voyager, a distributed environment for Java applications.

Second, we developed a distributed, multi-threaded prototype that implements a simulated network of routers as well as a 400 Kbytes/second network. Using it, we finally experimented with price-based load balancing mechanisms, quality of service (Qos) auction for connection control and competition among different network providers trying to maximize their profit.

1   INTRODUCTION

1.1   Outline

This semester project meets the intersection of two domains: network management and artificial intelligence (AI) – distributed AI (DAI) in particular. We will therefore briefly introduce those two backgrounds. Then, a statement of the goals of our project will close the introduction. In a second chapter, we present all the platforms we have evaluated, going into more detail for the ones that pertained the most to our work. In the third chapter, we explain the actual development of a prototype application, phase after phase, with each time the ideas underlying its implementation, the object models – in OMT [8] notation – guiding it and an evaluation of its results.

1.2   Theoretical background

1.2.1   Network management paradigms

In the 1990s, network management has evolved from a centralized paradigm – all decision-making is concentrated in a single management station – to distributed paradigms, where decision-making is spread all over the nodes of the network. Martin-Flatin et al. [4] wrote a survey that presents several of these paradigms, along with their associated vocabulary and with technologies that implement them (like Mobile code, JMAPI, CORBA, WBEM or ODMA). Rose [7] explains classical, centralized network management, along with an extended presentation of SNMP, a standard network management protocol.

1.2.2   Intelligent agents

Wooldridge and Jennings [11] have written a complete reference about intelligent agents that presents various applications of them. For a more basic introduction to AI, the reader can refer to the book of Russel and Norvig [9] (pp. 31-50). There, an agent is basically described as:
- Receiving <u>percepts</u> from its <u>environment</u> through <u>sensors</u>.
- Executing <u>actions</u> in its environment through <u>effectors</u>.
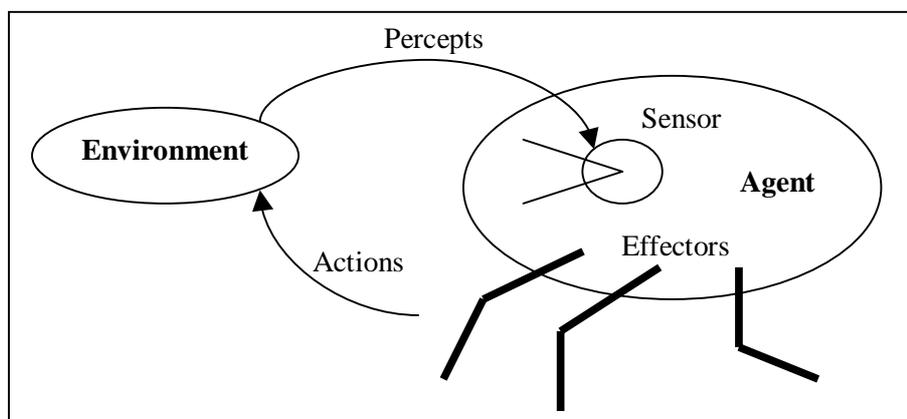This can be pictured as follows:

Figure 1

Then, an agent can be qualified as more or less intelligent, according to the category it falls in (listed from the least to the most complex):

- <u>Reflex agent</u>:   Agent that statically reacts to a percept according to some condition-action rule.
- <u>Reflex agent with an internal state</u>:   A state persists from one activation of the agent to the other and may be taken into account when firing rules.
- <u>Goal-based agent</u>:   An agent with an internal state, which does not only react, but also pursues an assigned goal.
- <u>Utility-based agent</u>:   A goal-based agent coping with several goals by associating a certain utility to each state of its environment.

1.2.3 Synthesis

Network management and intelligent agents can be combined as follows. Each node of the network is an intelligent agent whose environment is made of the node it is attached to and of neighbor agents. Agents can also be attached to a cluster of nodes instead of a single node.

Percepts are requests and notifications from other agents and node's properties extracted from its Management Information Base (MIB) [7], for instance. Actions are interactions with other agents and tuning of nodes.

Figure 2

Distributed network management consists in having the agents take individual decisions, communicate one with each other and pilot their node, in such a way that the network, globally, is managed.

1.2.4 Focus on connection admission control

The above architecture can encompass a wide range of management techniques, network topologies and applications. As stated in the abstract, we focused on connection admission control for multimedia sessions. A typical scenario is the following.

A user, at an end host, asks the network for a certain bandwidth in order to open a multimedia session with another user. The network has to respond to her whether it can ensure the QoS she asks for. Then, once the session is allocated, it has to manage, locally and globally, the appearances of new sessions, the variations in the actual traffic and the possible changes in the network properties. Furthermore, it has to bill the user, at a fixed price or in interactive auction.

Such issues were addressed by the prototype application we developed (see chapter 3). But, before, we needed to choose a platform that would support agents.

1.3 Goals of the project

A first goal was to evaluate platforms that support distributed, collaborative and intelligent agents, and that were available in spring 1998. This evaluation was made based on availability, compatibility and suitability criteria. Its expected outcome was the reasoned choice of one of those platforms as a tool for the second part of the project.

A second goal was to develop an application that would use and exhibit some strategies based on cooperative and intelligent agents, and addressing the problem of connection admission control. As such, a proof of concept for agent-based network management had to be brought.

2 RESEARCH OF A DEVELOPMENT PLATFORM FOR AGENTS

This phase of our work was conducted in two steps. First, we established a list of criteria that the platform we were looking for had to satisfy. But we found no such platform. As a consequence, we broadened our criteria, found a list of candidates and eventually chose one of them.

2.1 Original goal of the search

We sought first a single platform encompassing several properties. By "platform", we mean software components that can range from a collection of classes compatible among them to a high-level programming environment equipped with several tools helping programming agents.
Of course, it would have been possible to program agents from scratch, but this would certainly have filled the entire time dedicated to our semester project. Thus, the purpose of a platform was to ease our task as much as possible.

The properties of such a platform revolve around three issues:
First, the platform had to allow easy implementation of <u>intelligent</u> agents. With this expression, we mean agents that enjoy:
- <u>Autonomy</u>: Not each of their actions has to be specified, but they can be given goals they will seek to fulfill by themselves, given their own knowledge of their environment and ways to acquire knowledge.
- <u>Reactivity</u>: Agents have a perception of their environment and an external event may trigger a certain behavior of theirs.
- The <u>capability of inferring</u> actions from given goals or the capability of planning their actions. This is actually a consequence of the autonomy property.

Second, the platform had to provide an infrastructure for agents, including:
- High-level communication between agents, as message passing or as reactivity to events generated by other agents.
- Easy creation and management of a distributed population of agents.

Third, since our budget was null, the platform had to be freely available. Furthermore, in order to better understand its functioning (and perhaps to modify it), the availability of its source code was preferable.
It has to be noted that we did not require the searched platform to offer mobility to its agents, although many platforms we present next have this property.

2.2 Means of research

The search for such a platform was performed mainly on the World Wide Web. Our starting points were a collection of papers [1, 2] and a list of URLs:
- http://www.cs.umbc.edu/agents
- http://www.cl.cam.ca.uk/users/rwab1/ag-pages.html
- http://sics.se/ps/abc/survey.html

2.3   Results (1ˢᵗ part)

We found two platforms that satisfied the above goal : LALO and Plangent.

2.3.1 LALO

URL : http://www.crim.ca/sbc/lalo

LALO (Langage d'Agents Logiciel Objet) has been developed by the CRIM (Centre de Recherche en Informatique de Montreal). It is an agent-based programming language and an environment allowing the development of multi-agent systems.

The structure of LALO is presented by its authors themselves in a document in appendix A, where Agent Oriented Programming (AOP) is explained and the different types of agents supported by LALO dissected and illustrated. Therefore, we will not further detail LALO here, but just summarize its main features and bring our evaluation of it.

On one hand the agent infrastructure is made of a set of C++ classes one can customize through inheritance. The main provided classes are (from the least to the most sophisticated) :

- `BasicAgent:`   It just communicates with peers and takes actions according to received messages. Thus, it is purely reactive.
- `LaloAgent:`   A `BasicAgent` equipped with a mental state : beliefs about its environment and about the beliefs and capabilities of its peers. Its mental state is updated through its own actions and through received messages.
- `RbasedAgents:` A `LaloAgent` whose knowledge is updated and actions decided according to rules of inference. Therefore, this type of agent is the only truly intelligent one among the three.

On the other hand, the intelligence part is supported by the LALO language itself: a LALO program is translated into a C++ one and then compiled. For communications, agents use KQML (Knowledge Query Language).

Finally, LALO satisfies our criteria for a platform: `RbasedAgents` are autonomous, reactive and capable of inference, and KQML provides a high-level communication language. However, it has the major drawback of relying on C++. Not only C++ is very tedious to work with – compared to other languages like Java -, but is also little portable. And portability is critical for network management tasks, certainly involving different types of machines. That is why we did not adopt LALO.

2.3.2 Plangent

URL: http://www2.toshiba.co.jp/plangent/index.htm

Plangent was developed by the Systems and Software Research Laboratories of Toshiba Corporation. It enables Java-based agents that have the following features (quoted from above URL):

- *Autonomy:*   *The agents generate their own plans by themselves (planning capability), and perform their tasks according to plans (plan execution capability).*
- *Mobility:*   *The agents can perform their tasks as they move around networks.*
- *Adaptability:*   *If execution of the agents fails, they can redo planning after updating the information responsible for the failure.*
- *Locality:*   *The agents plan and perform their tasks according to the information at existing local nodes.*

However, an agent enjoys no mechanism for cooperating with its peers.

Planning is used by an agent to determine its own behavior in order to reach a certain goal, given by the user. The planning activity is guided by the following state diagram.

Figure 3 [above URL]

Also, actual plans are generated according to knowledge bases. The latter are attached either to a node or to an agent itself. They contain actions, which are *declarative descriptions of fragments of the plan to achieve the specified goal. Plangent agents generate plans by selecting appropriate actions corresponding to the goal and the situation.* [above URL]

Figure 4 [above URL]

Moreover, a user sets goals to agents through a GUI.

Finally, Plangent satisfies all our criteria for an agents platform, cooperation apart. It is currently available as a beta version.
However, the support its creators claim to offer seems to have faded away. Nevertheless, it would have been very helpful, since Plangent suffers from many bugs, preventing its use. Therefore, we could not use this platform either.

2.4   Revised goal of the search

Both above platforms being undesirable, we concluded that no single platform existed, that would satisfy all our criteria at once. Therefore, we tried to find the required properties in different but compatible products.

Typically, the platform we were looking for was divided into three parts:

Intelligence	Communication
Infrastructure	

Figure 5

- Infrastructure: Provides means to create, distribute and execute agents. The found contenders are listed in section 2.5.
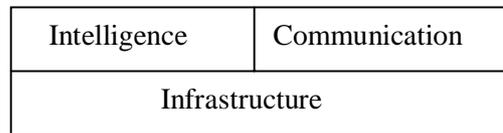- Intelligence: An inference engine or planner.
- Communication: A module providing high-level communication language, like KQML classes, for instance.

In this perspective, Java-based platforms became the most serious contenders, since they hold two decisive advantages.

First, thanks to the Java Virtual Machine that interprets Java byte code, Java applications are greatly portable and their behavior varies little from one type of platform to the other. As stated before, portability is a need when working in heterogeneous environments like the ones involved in network management : nodes of a network are actually likely to be running different operating systems.

Second, since we looked for separate products we would then integrate, those products needed to be highly and easily inter-compatible. Thus, the language they used had to be an object-oriented one (to facilitate integration) and a widespread one (to make an integration possible at all). Java is such a language: it is strongly object-oriented and it enjoys many publicly available libraries and applications.

2.5   Results (2nd part) : Candidate infrastructures

The first subgoal of our research was to find a platform as an infrastructure for agents. The second one was to find some planner software or some inference engine to be mounted on top of the agent infrastructure, in order to carry out the intelligence role. The second subgoal consisted in finding communication classes.

First, some impracticable platforms.

2.5.1   Agent Building Environment (ABE)

URL : http://www.networking.ibm.com/iag/iagsoft.htm

ABE is a set of C++ classes made by IBM and mainly providing intelligence as an inference engine. It supports Knowledge Interchange Format (KIF), an emerging standard in the area of inter-agent knowledge-level communication and an alternative to KQML.

Moreover, rules for the inference engine can be set by the user. An interesting aspect of ABE is the set of adapters it provides. An adapter is an interface that allows an agent to dialog with an external source of information. That way, adapters can act as sensors, effectors or adapters. Pre-built adapters include a time one, a NNTP one (for newsgroups), a HTTP one, a file system one and an email one.

ABE's drawbacks are its lack of explicit coordination mechanism between agents and the absence of a Unix version of the software.

In conclusion, ABE is certainly adapted to developing a standalone intelligent agent running under an IBM's OS, but not to building a distributed, multi-platform agent system.

2.5.2 Agent Building Shell (ABS)

URL : http://www.ie.utoronto.ca/EIL/ABS-page/ABS-overview.html

ABS is a research project of the Enterprise Integration Laboratory in the University of Toronto. Its aim is to offer *several layers of languages and services for building agent systems* [above URL].
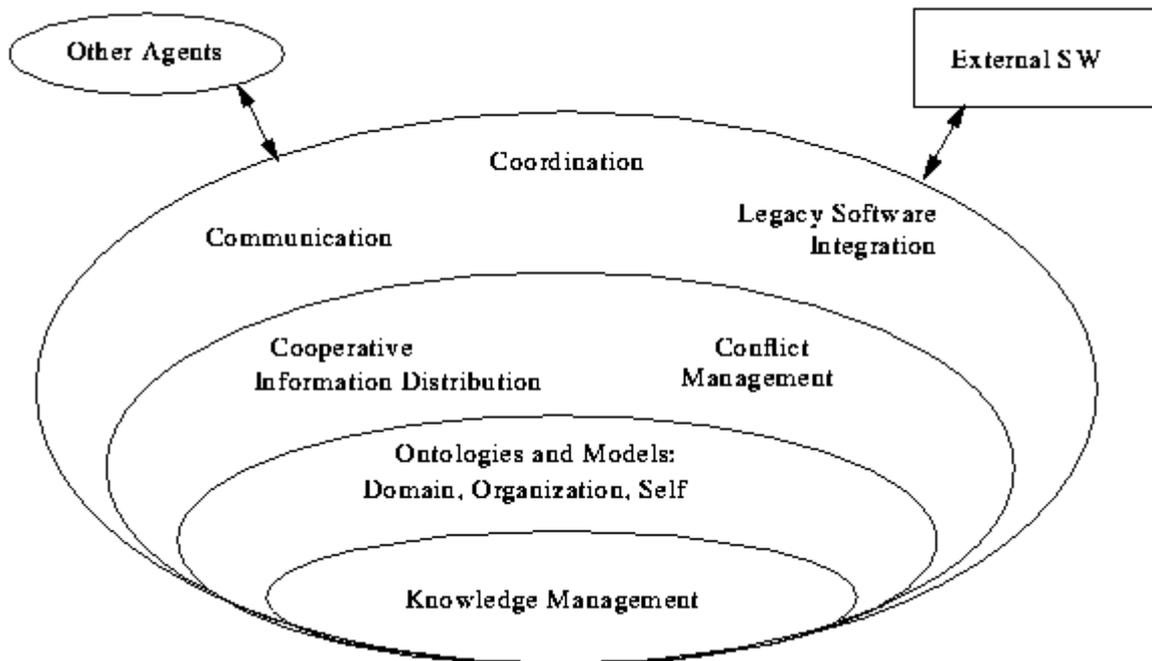

Figure 6 [above URL]

The <u>ontology layer</u> *consists of the actual conceptualizations agents maintain about their domain, environment and self.*
The <u>cooperative information distribution</u> services provide permanently active information distribution services allowing agents to stay informed about significant events without having to explicitly demand other agents to provide this information each and every time they need it. Agents advertise their long-term topics of interest to the community.
The <u>cooperative conflict management</u> service provides a general model for reasoning about retraction in a multi-agent setting. If an agent receives contradictory information from other agents, it applies this model to retract some beliefs and reinstall consistency both locally and with its neighbors. [above URL]
The last level - <u>Coordination, Communication and Legacy software implementation</u> - offers high reusability of existing software, because using a domain independent COOordination Language (COOL) in order to enable communication with other agents or pieces of software.
Much care is given to the organizational behavior of agents. Therefore, communication mechanisms are not buried into the agent's code, but rather encoded in KQML, as a component of COOL, and thus directly observable.
As a research project, ABS has given birth to a series of papers (see URL for reference) but not to any available software.

2.5.3 Java-to-Go

URL : http://ptolemy.eecs.berkeley.edu/dgm/javatools/java-to-go

Developed at UC Berkeley, Java-to-Go *assists in the development and experimentation of mobile agents and agent-based applications for* site-to-site computing. (The quotation is drawn from the above URL.)
Java-to-Go is a Java-based experimental toolkit, explicitly focuses on mobility rather than cooperation or autonomy, and is likely to be outdated: β version was released in 1996 and unchanged since. That is why we did not adopt it.

2.5.4 Kafka

URL : http://balcky.fujitsu.co.jp/hypertext/free/kafka

Kafka was developed by Fujitsu. It is a library of Java *classes designed for constructing multi-agent based distributed applications.* [above URL]
It relies on Java's RMI mechanism, which it adds the following services to:
- Runtime reflection: *Agents can modify their behavior (program codes) at runtime. The behavior of the agent is represented by an abstract class Action. It is useful for remote maintenance or installation service.*
- Distributed name service: *Agents have any number of logical names that don't contain the host name. These names can be managed by the distributed directories.*
Thus, Kafka only provides a thin addition to RMI. As such, it is not of great help in developing distributed, intelligent multi-agent systems.

2.5.5 Odyssey

URL : http://www.genmagic.com/agents

Made by General Magic, Odyssey is a Java-based agents platform, allowing inter-agent collaboration and agent mobility. Agents - subclassing the `Agent` class - have their own thread of execution and are by default mobile. A particular subclass of `Agent`, `Worker`, contains *a set of tasks and a set of destinations.* At each destination, the worker executes to completion the next task on its task list. Moreover, it can *manipulate its task list at any point during its travels.*
An interesting feature of Odyssey is its audit trail mechanism: an application can direct the status and debugging output of an Odyssey class it is using either to a window or to a particular file. That way, traces let by a whole population of agents are clearly separated.
However, Odyssey presents three drawbacks:
First, the mechanism it provides for agent collaboration is heavy: if they want to collaborate, agents set up and disband "meetings" on a particular host. This technique is fine for very mobile agents but inadequate for agents steadily setting on separate hosts and that want to exchange messages. And the latter case is ours.
Second, one single Odyssey server can run per machine. However, while developing a distributed system, it is convenient to have several virtual machines running on the same host.
Third, learning Odyssey is lengthened by its lack of documentation.
Among the above drawbacks, the first one motivated our decision of not choosing Odyssey.

2.5.6 Sodabot

URL : http://www.ai.mit.edu/people/sodabot

This environment, developed by the MIT AI Lab, claims to be able to generate software agents solely based on a specification of their behavior by the user. The latter would use the provided GUI and the high-level language SodaBotL to program agents. In fact, not only no download is available and documentation is scarce, but also the project looks dead, seen from its Web page.

2.5.7 Java Agent Template (JAT)

URL : http://cdr.stanford.edu

Developed at the University of Stanford, JAT provides a set of Java classes that enable cooperation between agents, using KQML messages.

However, coordination is insured by an Agent Name Server (ANS), which acts as a registry for all agents. When an agent is created, it has to first register with the ANS, informing it of its name and location. When it terminates, it has to notify the ANS as well. Also, all inter-agent communications go through so-called Agent Routers, which manage to find the recipient's address through the ANS and to forward the message to this recipient, potentially anywhere on the Internet.

Thus, communications are centralized, either through the unique ANS, or through routers. Therefore, JAT is not ideal for systems where agents must be distributed and is especially not scalable. That is why we discarded it.

Despite this abandon, we retained the KQML-related classes in JAT in order to use them later as a complement to a Java platform that would not provide any high-level coordination mean between agents. However, the course of the development of our application (see chapter 3) showed no need for a language such as KQML.

Now, let us consider two candidates that satisfied our criteria.

2.5.8 Aglets

URL : http://www.ibm.co.jp/aglets

IBM's Aglets are one of the most popular agents platform at the moment. It is made of Java classes, where agents are built by subclassing the `Aglet` base class. On each host where agents will be running, a server has to be started. Later, all agents can freely move between servers and communicate one with each other. They can also query each other, thanks to a set of key-value pairs attached to every agent and modifiable by itself. This mean of communication is much more flexible than a fixed set of methods.

From a GUI called Tahiti and attached to a certain aglet server, the user can create, deploy, retract and kill aglets. Additionally, she can ask the server to display information on memory usage, thread state, and log messages. Tahiti also provides a configurable security manager that enforces options - about incoming agents - set by the user.

2.5.9 Voyager

URL : http://www.objectspace.com

Voyager, developed by ObjectSpace, is a tool for building Java-based distributed systems.

Virtual Object
Voyager revolves around the notion of <u>virtual object</u>. It is an instance of a virtual class - a Java class inherited from the predefined `Vobject` one - located on a certain host. As such, a virtual object is a distributed object.

The code of a virtual class is automatically generated from a classical Java class by passing it through a generator called `vcc`. It maps all methods of the original class in order to accommodate remote method invocation (RMI). It functions like an IDL compiler for RPC, except that it does not generate any separate stub and skeleton, but just a single Java class that can then be compiled as any other class, using `javac` for example. In a same program, classical classes can cohabit with virtual ones; the former just cannot be accessed remotely.

Communication
Like on any reference, method calls can be executed on a virtual reference, thus accessing and communicating with the remote object.
But virtual references provide three modes of invocation:
- synchronous: This method invocation is a classical, local one.
- asynchronous: *Future Messenger*

> Instead of directly calling the remote object, the caller object crates a new thread - a <u>Messenger</u>, in Voyager's words - that take charge of calling the remote object and return the result to the caller afterwards. That way, a caller is not blocked waiting in case communications are slow or the called host crashed. This mechanism can also be used for activating dynamic objects, that is, objects that keep their own flow of control and behave like distributed programs.

- asynchronous, but the result is never returned : *One-Way Messenger*.

A timeout can always be assigned to a remote method call, whichever its mode is. Also, remote method calls can occur between objects on the same host.

Voyager also supports multicasting. Each virtual object can subscribe to a group, called a <u>Subspace</u>. Then, methods can be invoked on a whole group instead of on a single object; the method call is meant to be transparently applied to each member of the group. Actually, our experience shows that, instead of real multicast, a new thread is created that calls every member object one after the other. Thus, if the method call crashes on one object, it is not delivered to the other ones.

Infrastructure

On each host where virtual objects are residing, a Voyager server is running, attached to a certain port. This server can be started either from within or as a Java application. In the latter case, it just passively waits for objects to come. In fact a virtual object can be transparently moved from one server to the other.

We said that Voyager does not provide any registry and that a remote object can be accessed only via its virtual reference. To get to know the virtual reference of a certain object, Voyager provides aliases.

To each virtual object can be associated an <u>alias</u> (a string). Then, an object, knowing the location (host name and port number) and the alias of another object, can call the corresponding Voyager server to get the virtual reference to that object.

Remarks

First, Voyager servers can render their objects persistent.

Second, virtual objects can also reside in applets, thus easing agent management and user interaction with the system.

Third, what Voyager calls an <u>Agent</u> has little to do with the AI meaning of the word, but simply denotes a virtual object that can be moved while seamlessly executing.

Finally, Voyager comes with an extended documentation [6], including examples of use.

Object Model

Appendix B is an object model - in OMT notation - we drew, based on the explanations given by the Voyager User Guide [6]. This model does by far not represent all Voyager classes, but shows the most significant ones.

2.6   Choice

The only two remaining platforms that were practicable were Aglets and Voyager. Both are very similar in their capabilities and limitations.

They both enable distributed agents that communicate one with each other in various messaging modes, move from host to host, have GUIs and query each other about key-value properties.

But none of them provides intelligent agents that would be equipped with an inference engine, a system of beliefs or a planner. Nor do they offer a high-level communication language like KQML or KIF; rather, all communications go through query or RMI.

Therefore, choosing one platform rather than the other to develop our application would not have any critical incidence on that development. But a choice needing to be done, we looked at the details.

Aglets benefit from their user-friendliness (they can be piloted through a GUI) and their simplicity of use. Voyager benefits from its simplicity of reusing existing Java classes (that would just need to be

passed through `vcc`), from its extensive documentation and from its more sophisticated messaging (multicasting, asynchronous calls).

Finally, we opted for Voyager based on two motivations. First, we had enough time, during our semester project, to overcome the relatively greater complexity of Voyager. Second, we desired the greatest flexibility in inter-agents communication.

This choice done, we had at our disposal a basic infrastructure for distributed agents, but still no tool to generate intelligent agents, like we initially wanted (see part 2.1). Actually, the direction subsequently taken by our work - we focused on agents' collaboration instead of intelligence - showed no more need for such a component.

3   REALIZATION

Our work of design and implementation of an agent system ranged over several, incremental phases. The first one was concerned with setting up a simulated and distributed network, and an agent infrastructure. The second one saw the simulated network have a second and equivalent implementation that allowed real data transport. In phase 3, the application became distributed on various machines, although its functioning did not change. Phase 4 and subsequent ones were the development of cooperative agents, used for load balancing and for connection admission control. The cooperation framework is a market, where agents are exchanging bandwidth with some commodity, which we will call "money" in the remaining part of this report. Thus, cooperation mechanisms include price publishing, bandwidth auction and sale of guaranteed QoS. For instance, phase 4 was concerned with load balancing, and phase 5 with billing the user for its use of the network. Phase 6 was the development of a price-based connection control mechanism.

Subsequent phases were no implementation but solely specification work. Phase 7 laid the base for multiple providers sharing the network among themselves and competing for services. Afterwards, we modified our agent system in order to increase the realism of the simulation. Thus, phase 8 was the design of a new connection admission mechanism based on bandwidth reservation. Phase 9 sophisticated it by linking it with pricing.

3.1   Phase 1 : Simulated network faking data transport

In the rest of this report, we will abbreviate this network implementation by "simulated network".

3.1.1   Network settings

A <u>network</u> is a graph of routers. In this phase, we used a very simple network topology, as pictured in figure 7.

The number of layers and the number of nodes per layer can be
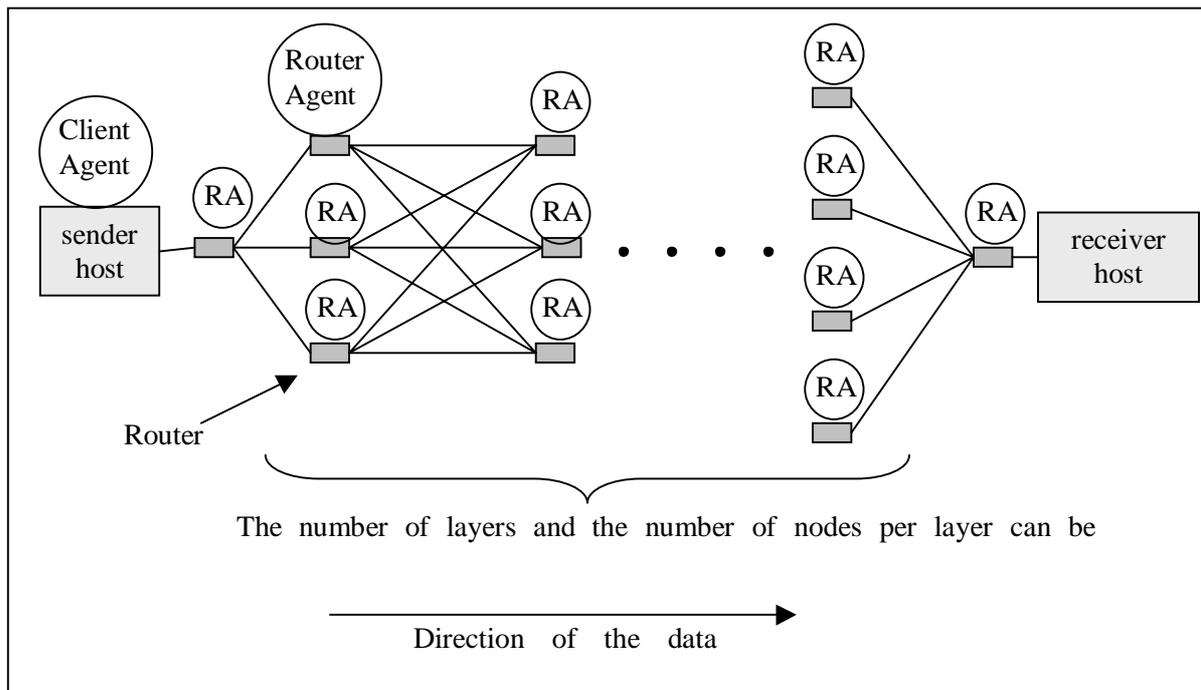
Direction of the data

Figure 7

At each end of the network is a host. One host - the "sender" host - at one end will request a certain service with a host at the other end - the "receiver" host.

Notice that data flow is unidirectional (from sender to receiver). We chose this setting in order to simplify the simulation. Bidirectionality can be recreated by putting unidirectional networks one on top of each other, and would therefore complicate the simulation without bringing any fundamentally new network management issue.

Amounts of data are measured in data units. A data unit can be just a number used for simulation purposes - like in this phase - or it can represent a packet of a certain size - like in phase 2.

In this chapter, by "node", we will indifferently mean a router or a host.

3.1.2 Agent settings

On each router sits a router agent. It knows all agents attached to adjacent routers - upstream and downstream.

With each host is a client agent. It knows the agent of the unique router the end host is linked to. A client is an application who sends data through the network, on behalf of a user.

3.1.3 User tools

The parameters of the network settings are asked to the experimenter (the one who is running the network simulation) by an `Initializer` class. In this phase, they are simply extracted from an editable file.

A `Supervisor` GUI enables viewing what is happening in the network during the simulation.
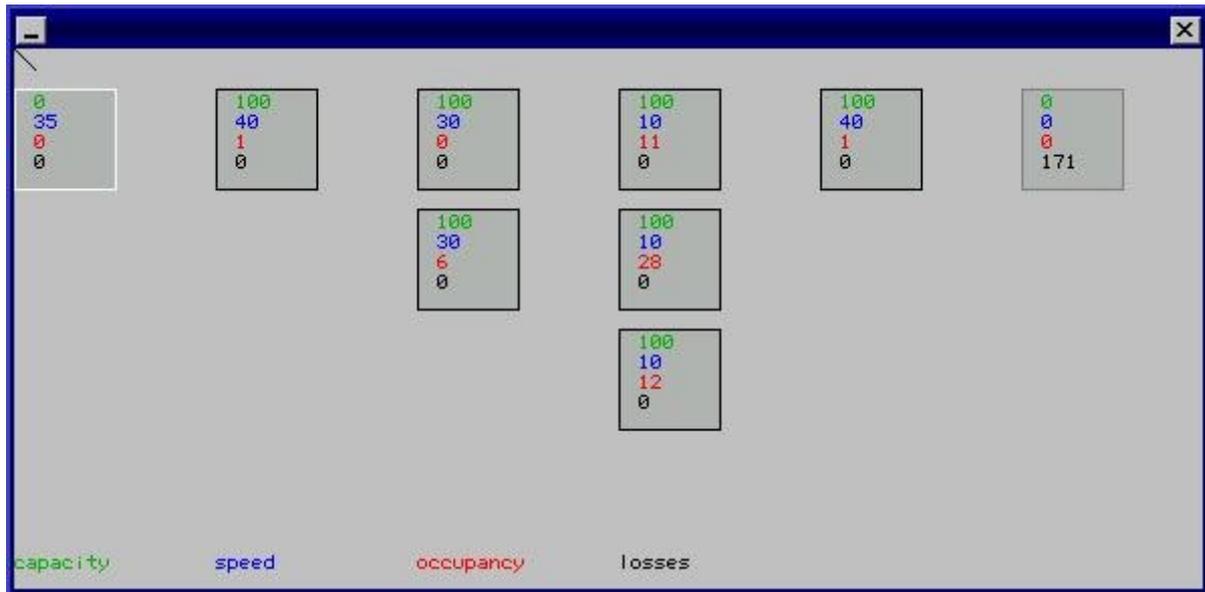
Figure 8

3.1.4 Router structure

A router is characterized by 4 attributes:
- 2 constants :
    - a buffer <u>capacity</u>
    - an <u>output rate</u> : the number of data units it can send per second
- 2 variables :
    - a current buffer <u>occupancy</u>
    - the <u>loss</u> : the number of data units it has lost so far.

Moreover, to each output channel of a host is associated an enabler that allows or forbids data to be sent through it.

We did not care about routing, since it is trivial in the chosen network structure. Data to be sent is dispatched to an arbitrarily chosen output channel.

Data flow is just represented by a quantity (data units).

The above attributes can be chosen at random at router creation, except for both routers close to an end host, which receive higher values since they are bottlenecks in this topology.

A router has no limit over the amount of data it receives from other routers. However, if its buffer is full, incoming data is still accepted but discarded.

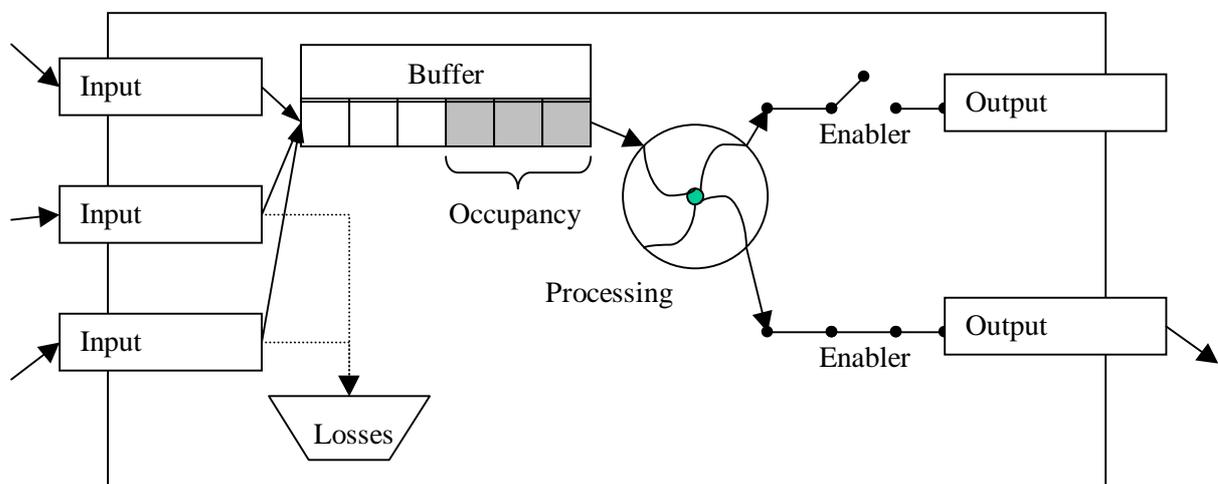To summarize, a router can be represented in the following way:

Figure 9

3.1.5 Host structure

In our network simulation, the `SenderHost` is simply a source of data flow while the `ReceiverHost` is a pit.

Both of them do not purely represent one computer (end host) each. Rather, they encapsulate numerous network components.

In fact, videoconferences are most of the time established across WANs or across the Internet. For shorter distances (LANs), participants can physically meet each other. Thus, the network we were working with (see figure 7) is a WAN or the Internet. But end hosts are seldom directly connected to such networks. Often, they are part of a LAN, which, in turn, has access to a WAN. Such a situation is depicted by figure 10 for a sender host and by figure 11 for a receiver host.
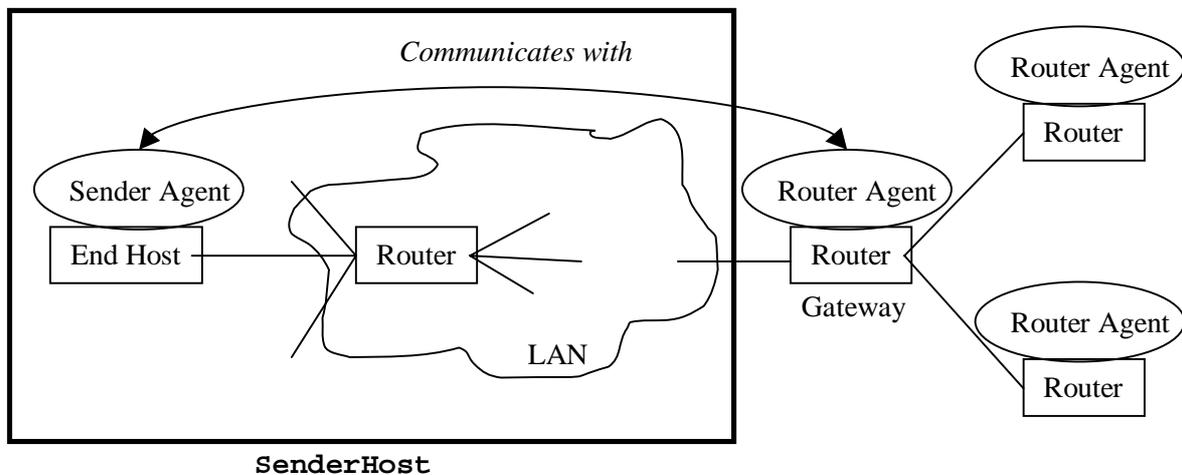

SenderHost

Figure 10

A sender host is a part of a LAN. In particular, it is connected to a unique router. The LAN itself is connected through one router - a gateway - to the WAN. This router has access to several other ones. The agent system is distributed as follows.

Since the gateway belongs to the WAN, a router agent stands on it. Router agents also sit on top of the routers next to the gateway, like on any other router in the WAN. The gateway itself is not differently modeled from other routers. Routers in the LAN have no agent, because we did not manage the LAN, but the WAN. The end host itself interacts with a sender agent - for instance, to dialog with the user or to set the application's output rate - attached to it and communicating with the gateway's agent.

Since there is no agent system in the LAN, we could abstract the end-host and its LAN into a single `SenderHost` object, equipped with a sender agent. Moreover, we could have included the gateway into this abstraction as well, and have a `SenderHost` directly sending data to various routers. We chose not to do so, based on distinctions we will be making in phase 7.

A similar situation prevails for the `ReceiverHost` - see figure 11. To be noticed is the `ReceiverAgent` on the receiver host. Its purpose is to provide feedback to the next router agent about the data inflow rate and to agree with the initiator of a videoconference, for instance, upon the connection set up. Until further notice, we will assume it always accepts any incoming data flow, however low or high its rate is.
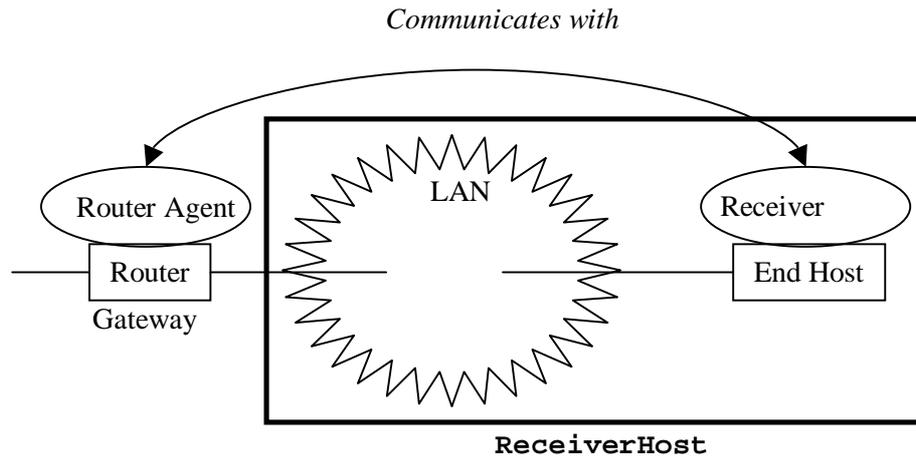
Communicates with

Figure 11

3.1.6 Client structure

The client host is instructed by its agent to send data at a certain rate. This rate can account for several opened multimedia connections, for example. Then, it produces data at the specified rate and sends it to the next router.

3.1.7 Agent-router communication

An agent communicates with its host through a set of attributes. Each host has such a set : basically, the ones mentioned under "router structure". An agent can query its host about those attributes, and it is all it can do to it. Some of those attributes are in read-only mode - like the capacity, the losses or the occupancy of the hosts -, while other ones are modifiable - like the output rate of the sender client or the enablers of hosts' output channels. Therefore, a node never executes complete operations on behalf of its agent, like method invocations would however allow it to. Rather, the agent can guide the node's behavior by changing its attributes. That way, the agent's presence does not disrupt the node's normal execution. Furthermore, the node has no knowledge of the agent that is monitoring it.

This weak interconnectivity enables the following architecture, where the implementation of the network layer can be changed without affecting the agent system.
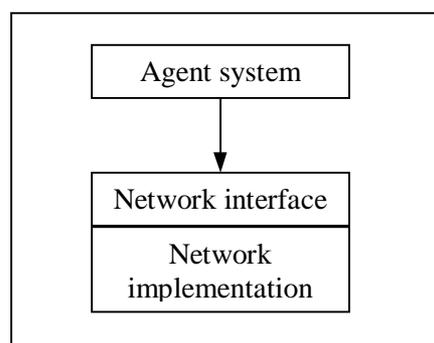

Figure 12

We realized the network interface using Java interfaces, for routers, sender hosts and receiver hosts respectively, as pictured in appendix D1.

In this phase, agents were idle. We developed them from phase 4 on.

20

3.1.8 Implementation remarks

Ultimately, routers and their agents must be distributed on several machines. In this phase however, they are all executing on the same computer. The creation of routers, clients and agents, as well as their eventual distribution, is ensured by a `Distributor` class.

Network traffic is simulated as follows. A router is a thread that sends one data unit every 1/m second to a certain router, unless its buffer is empty. Thus, m is its output rate. Data unit sending is realized by a remote method invocation (RMI).

M values need to be small. Otherwise, the router's speed depends more on the operating system's and Java Virtual Machine's performances and on the switching mechanism between threads, than on the actual simulation settings. As long as m remains small (~10) and the number of nodes too (~10), the simulation works fine on a single computer.

3.1.9 Conclusion

Phase 2 ended with a network simulator running on a single machine and whose activity can be viewed through a GUI.

3.2 Phase 2 : simulated network transporting real data

In the rest of this report, we will abbreviate this network implementation by "real network".

3.2.1 Framework

Phase 2 responded to an additional objective of our work: have an agent system running on a network that transports real data, and that does not only simulate this transport.

Thanks to our previously mentioned two-layer architecture, we satisfied this requirement by just providing a new implementation of the network layer that matches the network interface.

3.2.2 Router's implementation

Because real data needed to be transported at a steady rate, the router's implementation is more sophisticated than previously an is schematized as follows:

Figure 13

A first choice needed to be made between two ways of transporting data from one node (S) to the other (D).

The first way was for S to make a RMI on B with the data packet to be transmitted as an argument. It had the advantage of being easy to use but the inconvenient of the overhead of computation and communication associated with RMI. The second way was to use sockets between routers. It was more realistic - a socket is a real channel between two objects - but trickier to set up initially.

For realism and performance reasons, we chose the second alternative. Actually, since we did not implement the first one, we never assessed the possible performance gain.

Let's have a look at the inside of a router (see figure 13). It is made of one array of input sockets and one of output sockets. In the middle is a `SharedBuffer` for packets. Packets are simply fixed-sized arrays of bytes. A unique `SenderThread` is taking packets out of the buffer and sending them through one of the output sockets. In order to satisfy the given output rate `r` of the router, it wakes up every second and attempts sending as much as `r` packets. That way, the router's performance can be totally controlled by the experimenter's setting of `r`. For a best-effort performance, one will simply set a very high `r`.

To each input socket is attached a `ReceiverThread` that waits for packets on the given socket and puts them into the buffer (or discards them if the buffer is full). We used a `ReceiverThread` socket instead of a unique thread because it was easier to have a thread waiting on one socket rather than on several.

3.2.3 Implementation remarks

Sender and receiver hosts are similar in structure to routers, except that a `ReceiverHost` has neither `SenderThread` nor buffer and that a `SenderHost` has neither `ReceiverThread` nor buffer, but is producing its data from scratch.

The overall inheritance and interface structure of all node objects involved in phases 1 and 2 is shown in appendices C2, D1 and D2.

3.2.4 Priorities

For a good functioning of the system, its various threads had to have different priorities, especially because the Java Virtual Machine (JVM) under Solaris or Linux (operating systems we worked with for this project) did not provide time slicing among threads.

We used 3 levels of priority: (1 is the lowest)

Level 3 :   Agents, infrastructure threads
Level 2 :   `ReceiverThread`
Level 1 :   `SenderThread`, network simulation threads (phase 1)

`ReceiverThread`, `SenderThread`, and network simulation threads are part of the network layer. This layer has lower priority than the agent system layer, because agents need to be active even if the network is overloaded and has its threads continuously executing. Also, infrastructure threads - that are initializing or closing the system, like distributing objects - need to be of highest priority to ensure the system's consistency.

In a single node, `ReceiverThreads` have priority over `SenderThreads`. Actually, sockets are handled at their both ends as input/output streams. Thus, they can act as buffers, which would falsify the simulation. Then, it has to be absolutely avoided that packets are sent into a socket an not immediately received by the router at the other end. Giving higher priority to `ReceiverThreads` guarantees that channels are always empty before a packet is sent.

3.2.5 Performance of the network

We measured the maximum throughput of our network on a single 200 MHz computer running Linux and in the following configuration:
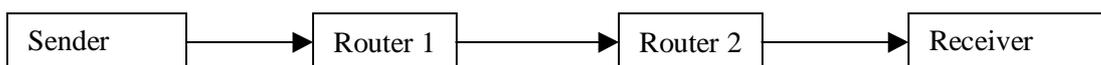

Figure 14

The router's buffer capacity was 20'000 data units (packets).

The maximum throughput was around 1,5 Mbytes/second. The limit did not stem from the buffer capacity or losses, but from the activity of the sender host that could not send packets faster. The

corresponding throughput on a single Sun Sparc 20 station running Solaris was around 150 Kbytes/second.

3.3   Phase 3 : Distributed network

In previous phases, all objects were running on the same computer. To be more realistic, we distributed each node together with its agent on a different computer. That way, each node had its own CPU and communications were really remote.

Since most of the objects we had developed so far were using Voyager's `Vobject`'s capabilities, having them distributed left their implementation untouched, but modified the `Distributor` class.

We tested again the real network, but this time with one `RealRouter` distributed on each host machine. Hosts were Sun Sparc 20 running Solaris. The obtained throughput was around 400 Kbytes/seconds. This corresponds to a lower limit bandwidth for multimedia sessions. However, much higher throughput would be expected on faster computers. As a remainder, we showed that, when running on a single host, the real network was performing 10 times better on a 200 MHz PC than on a Sun Sparc 20 station.

3.4   Phase 4 : Bidding-based load balancing

Previous phases were dealing solely with the network infrastructure. In this phase, agents came into play.

3.4.1   Problem

Previously, a router receiving data was forwarding it towards an arbitrary chosen channel. Given our network topology, data was guaranteed to reach its destination anyway. However, such a blind routing among equivalent paths from source to destination could overload some routers - typically when several routers were sending all their data to a same router - while letting other ones idle. Consequences of overload include longer delays of transmission and possible losses of data.

3.4.2   Solution

Agents could collaborate to balance load among routers. Since we were applying market-based mechanisms, we retained the following solution.

To each router agent is associated a price. This price is chosen and published by the agent, and corresponds to the amount of money per data unit, billed to every router that is sending data to the one where the agent stands.

Given this basic economic mechanism, an agent has two decisions to make:
- Which router to send data to?
- Which price to ask?

In the next two points, we will provide answers to these questions.

3.4.3   Routing

In a first stage, an agent simply ordered its router to send all his data to the cheapest next router. The drawback of this strategy was the fact that all routers of a same layer would send their data to the same router, thus completely overwhelm it.

Continuously running a network agent would be a waste of processing time, especially since the router and its agent are running on the same machine. Rather, an agent is waking up at regular intervals of time. For our experiments, we set this duration to one second. When it wakes up, it polls its router for its state (buffer occupancy, losses, …), takes prescribed decisions (like asking other agents about their price, or deciding of its own price), executes consequent actions (like re-routing its router's data flow) and finally gets to sleep again.

3.4.4 Price calculation

At first, one variable only was considered: the router's buffer occupancy rate. Then, the idea was to have a price that increased with the buffer occupancy. The simplest formula was a linear one :

$$P(\alpha) = 100 \cdot \alpha$$

where   P = price
                α = occupancy rate = occupancy / buffer size

3.4.5 Object Model

To account for heterogenous agent in a same agent system, we used Java interfaces, which allow different implementations (for instance, various price calculations or data dispatching strategies) and which are pictured in appendix D3.

3.4.6 Client Agent

On the sender host's side, the client agent was displaying a GUI that enabled the user to specify the data output rate it was sending through the network. This interface eased the experimentation's control. For the same purpose, we used the simulation network rather than the real one, over phase 4 and subsequent ones.
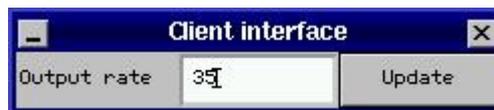

Figure 15

3.4.7 Evaluation

Although simple, the above formula was sufficient to balance network load in all experiments we ran. The good functioning of the load-balancing scheme we used is however limited by the size of the buffer. If this size is small, then a router's state can quickly jump from quite idle to saturated. Then, losses can occur if upstream agents do not react soon enough by redirecting their data flow toward another router.

This phenomenon does however not question the mechanism we used, but rather points out that the length of the sleeping period of the agents is a parameter that has to be carefully tuned, according to the routers' buffer capacity.

3.5 Phase 5 : Billing

3.5.1 Goal

In phase 4, the price displayed by each agent was actually just a translation of the occupancy level of its router. In this phase, we wanted the price to be a cost for the client, just like a phone company is billing its users for the use of their phone.

3.5.2 Billing mechanism

The cost of the transport of a data unit along a certain path P in the network is the sum of the prices asked by all the nodes on this path. That is :

$$c = \sum_{n \in path} P_n$$

where   C = cost
            P_n = price at router `n` when the data unit is passing there.

We imagined two way of summing all costs back to the client host.

In the first one, each router agent was distinguishing how much data had come from each channel, and was then billing the corresponding amount to each upstream agent. The latter was then taking this bill into account to bill its own upstream agents (in such a way that it was keeping its own money balance), and so on back to the client. This mechanism provides an accurate accounting. However, it involves a lot of inter-agent communication, especially if it works in real-time (remote calls for bills every second).

Agent 1   *Price publication*   Agent 2

$Price_1$ = own cost
$Bill_2 = Bill_1 + Price_1$
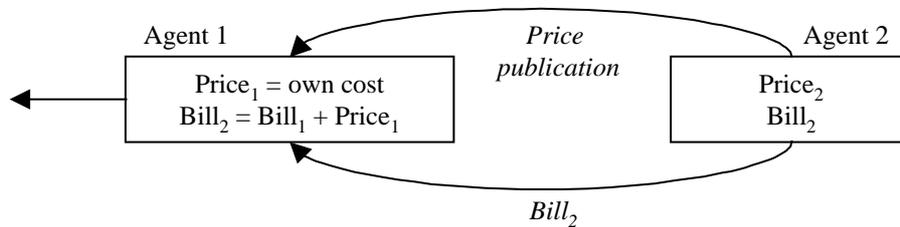
$Price_2$
$Bill_2$

Bill$_2$

Figure 16

The second mechanism, on the other hand, does not involve more communication than in the previous phase. An agent does not explicitly bill the routing cost to its upstream agents. Rather, it records the price displayed by the agents it is sending data to and adds the corresponding cost to its own published price. Thus, billing information transmission is embedded into price communication. Actually, this scheme does not lead to an exact accounting, since the price asked by a downstream agent may change while the agent is sleeping and still believing in the old price. However, those inaccuracies tend to compensate each other over price raises and falls.

Price$_1$
publication   Agent 1   *Price$_2$* *publication*   Agent 2

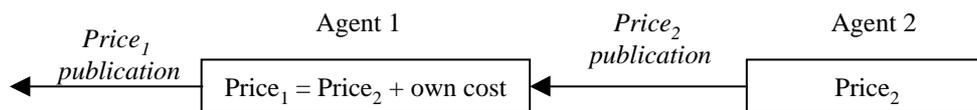$Price_1 = Price_2$ + own cost $Price_2$

Figure 17

Finally, the second scheme is closer to the notion of price. Over a path taken by some data, the price is increasing as the distance to the client decreases, just like in the production chain of a service or a good.

3.5.3 Client GUI

The existent client GUI is enhanced with a field displaying the current price for using the network. The user can update its data output consequently.

Client interface		
Output rate	35	Update
Price per data unit	48.0	
Total charge	24057.0	

Figure 18

3.6 Phase 6 : Connections

3.6.1 Goal

In phase 5, the client host was just sending data at a fixed rate, however low or high the price was, and then passively receiving the bill.

In practice, users have budget constraints and to not hold to a certain QoS without caring about its price.

3.6.2 Solution

In order to accommodate this requirement, we defined the notion of connection, which is the association of:

a)   A maximum bandwidth
      The current bandwidth does not grow higher than this limit.

b)   A minimum bandwidth
      If the current bandwidth falls under this limit, the connection is canceled.

c)   A ceiling price per data unit
      If the price goes above this limit, the current bandwidth is decreased.

d)   A current bandwidth
      Which is kept as high as possible.

In this framework, we defined several kinds of connections, which are products sold by the network to its client.

Product number	*1*	*2*	*3*	*4*
Name	Fixed Bandwidth (R)	Bounded price (P)	High-class multimedia (R, S)	Realistic multimedia (R, S, C)
Max. bandwidth	R	∞	R	R
Min. bandwidth	0	0	S	S
Ceiling price	∞	P	∞	C
Comments	It actually corresponds to the product offered under phases 4 and 5.	The client sends as much data as its can, as long as it does not exceed a certain price.		

Since product # 4 is the most realistic and general one out of the four, the three other products will not be considered in the remainder of this phase.

3.6.3 Client View

A user can create connections via GUIs. We added a button to the `ClientGUI` inherited from the previous phase.
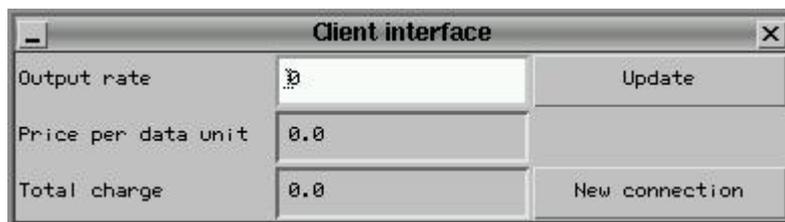

Figure 19

Clicking "New Connection" pops up a `ConnectionGUI`, enabling a user to enter the characteristics of the connection she wants to open.
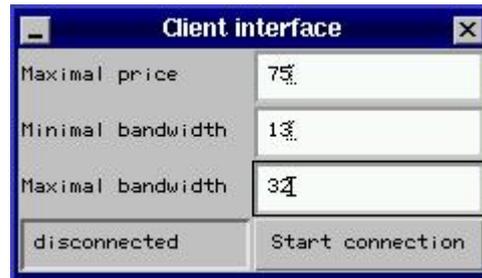
Figure 20

Upon a click on "start" button, the connection is created, its attributes cannot be changed any more and a "stop" button replaces the "start" one. Clicking on "stop" destroys the connection and enables updating the attributes. When a connection is created, a field (in the lower-left corner of the window) displays the current connection status: disconnected or connected. In the latter case, the exact amount of bandwidth it is using at the moment is displayed.

Several connections can exist at the same time, thus competing for bandwidth by means of the price they are ready to pay for it.

3.6.4 Client Agent

The client agent is registering all created connections. Then, its job consists in realizing them as much as possible while respecting their price constraints.

We formalized this allocation problem as follows:

$$\text{Max} \sum_i b_i$$

under constraints :

$$\forall i \text{ such that } b_i > 0, \qquad p_i > P$$
$$b_i \le M_i$$
$$b_i \ge m_i$$

Where: The connections are numbered 1, .., i, .. n
p_i = maximal price of connection i
m_i = minimal bandwidth of connection i
M_i = maximal bandwidth of connection i
b_i = current bandwidth used by connection i
P = current price asked by the gateway.

The above looks like a linear program, but it is not. Or at least it is not a simple one, because either b_i = 0 or $b_i > m_i$. As a consequence, we did not solve this problem using a simple operations research algorithm, but rather used methods of our own, such as the one introduced below.

3.6.5 Connection allocation scheme 1

The principle of this first connection allocation scheme is:

For each connection: if $p_i < P$
Then $b_i := 0$
Else $b_i := M_i$

In other words, the minimum bandwidth is ignored: a connection is either disconnected or using its full bandwidth. Thus, we hoped that, when several connections are competing, the price reaches an equilibrium that fully enables some connections while excluding the other ones from being connected.

However, this scheme could not work with the price function we used so far. Because of $p(x) = 100 \cdot x$ (where x is the buffer occupancy rate), the overall price is bounded by $100 \cdot$ (# layers).

Let B be the maximum bandwidth the network can provide. If a connection has $M_i > B$ and $p_i > 100 \cdot$ (# layers), it will never be prevented from using a bandwidth of M, although this bandwidth cannot be

supported and losses will occur as a consequence. Therefore, a new price function had to be found in order to avoid those losses or to inform the client agent of their occurrence.

Losses occur when buffers are full. Then, we needed a price function that makes buffer occupancy rates close to 1 unaffordable, however high the maximal price is.

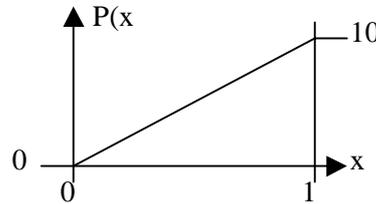So far, for load-balancing purposes, p(x) just needed to be monotonously growing. We chose a simply linear function:

Figure 21

The new price function we proposed was:

$P(x) = $ $100 \cdot x$ if $x \leq 1/2$

          $25 / (1 - x)$ if $x \geq 1/2$

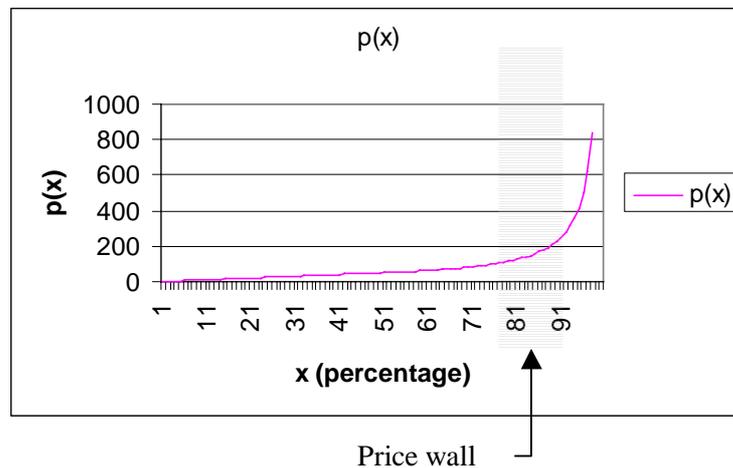Thus defined, p, as well as its derivative, is continuous.

Figure 22

Between $x = 1/2$ and $x = 1$ is a zone we called "price wall" (see figure 22). Beneath this zone, prices are moderate and act as selectors among competing connections. Beyond it, prices become so high that they prevent all connections from filling a router's buffer.

Tests we led showed that this first scheme was working well. However, its major drawback is to be binary: a connection is either disconnected or fully enabled.

For instance, when a connection has $M_i > B$, it oscillates between disconnected and connected states. But a user never wants a connection that is interrupted every couple of seconds. Then, a new control scheme needed to address this issue.

3.6.6 Connection allocation scheme 2

Figure 23 shows a sample execution of the previously proposed control scheme, with one connection. The bandwidth actually allocated to the connection is pictured in pink, while the price asked by the network is traced in blue. The time on the X-axis is measured in seconds, because agents react every second. The experimental settings were:

          $B = 30$ $m = 40$

          $M = 49$ $P = 50$

Since $M > B$, the network could not continuously ensure the requested connection.
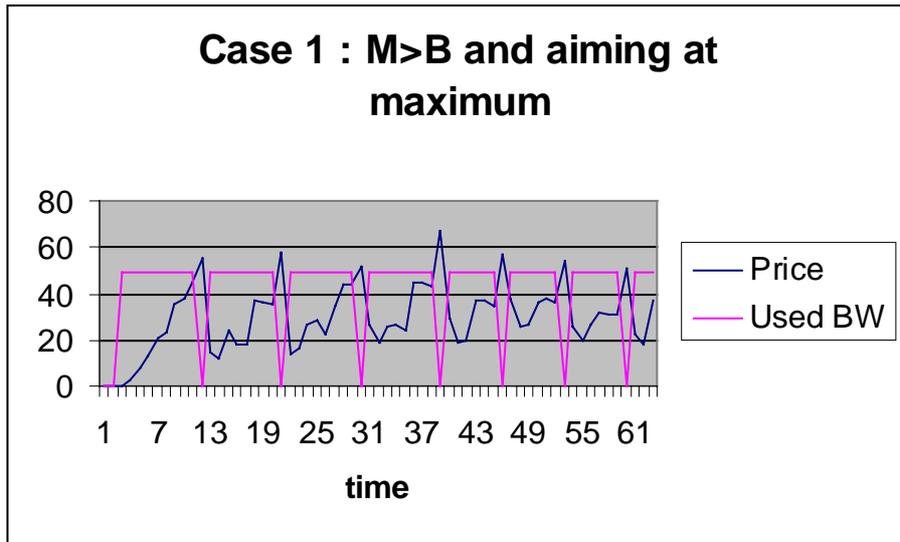
Figure 23

We observed that the connection is interrupted about every 8 seconds. These very short interruptions of the data flow let the routers' buffers flush themselves, thus lowering the price and allowing the connection to be established again.

Another naive connection control scheme is the following (scheme 2):

For each connection: if $p_i < P$
Then $b_i := 0$
Else $b_i := m_i$

It is actually the same as the previous one, except that the minimum bandwidth is aimed at, instead of the maximum one. With the same settings as before - thus with m>B -, the following occurred:
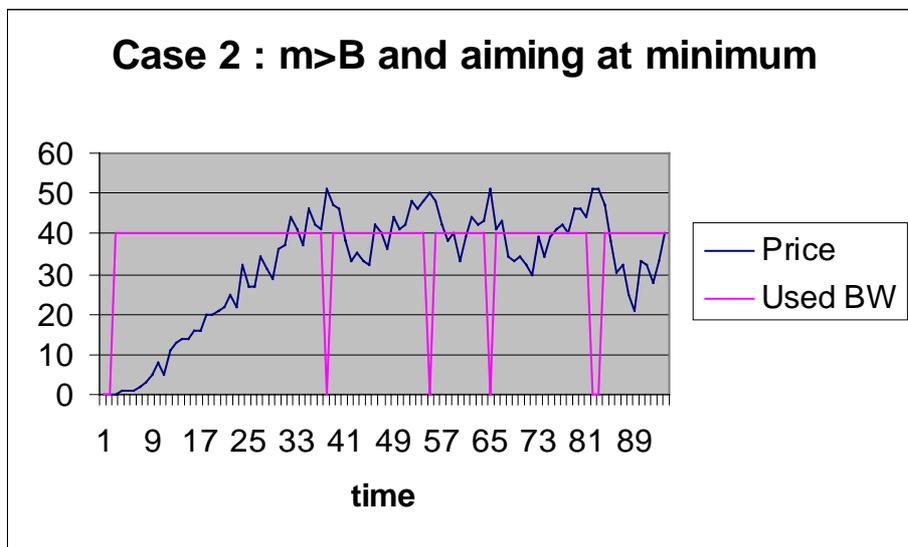

Figure 24

The shape of both curves is similar to the ones observed in case one. The interval between interruptions is longer, but it is mainly because (m-B) < (M-B). Not surprisingly, this scheme is sufficient when 25 = m < B, as shown in figure 25 But such a situation is not realistic.
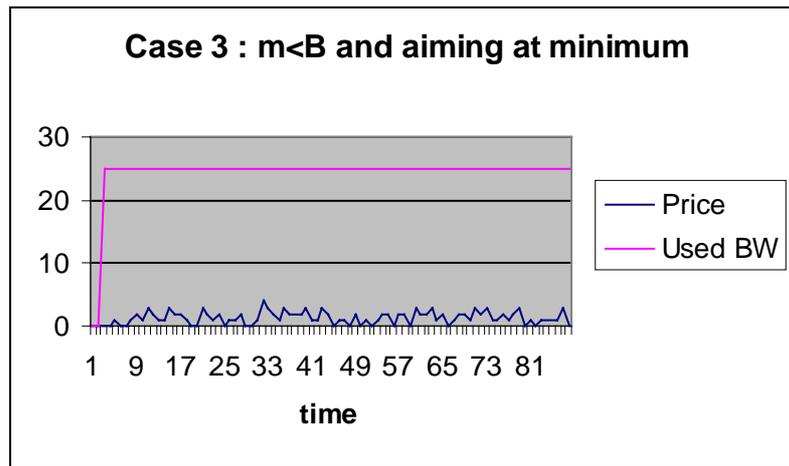
Figure 25

In conclusion, scheme 2 does not outperform scheme 1 but presents the same problems.

3.6.7 Connection allocation scheme 3

A connection control scheme consists in updating b according to the values of b and P. Then, it can be represented in a table, such as the following one, which defines scheme 3:

b P	≤ p	∈ [p, 110% p]	> 110% p
0	M	0	0
M	Incremented	0	0
∈ [m, M]	incremented	Decremented	m

$$
\begin{array}{lll}
\text{Incremented} & ::= & b := (M\text{-}b)/2 + b \\
\text{Decremented} & ::= & b := (b\text{-}m)/2 + b
\end{array}
$$

Comments:
The cells represent future values of b. Price values are separated in 3 zones:
- P ≤ p   satisfactory zone: bandwidth can be incremented
- P ∈ [p, 110% p] :   Price is beyond limit, but still in a reasonable margin (10% has arbitrarily chosen; the margin width is a parameter to be tuned). Bandwidth has to be slightly decreased, but the QoS does not need to be disrupted.
- P > 110% p :   Price is widely beyond limit : bandwidth has to be dramatically decreased.

Incrementing b is done by halving the distance from b to M. Similarly, decrementing b halves its distance to m.
That way, b is greatly reactive when in the middle of its margin (around (m+M)/2) but slow to reach boundary conditions.
Figure 26 shows a sample execution of this scheme, under the following settings :

      B = 30   m = 20
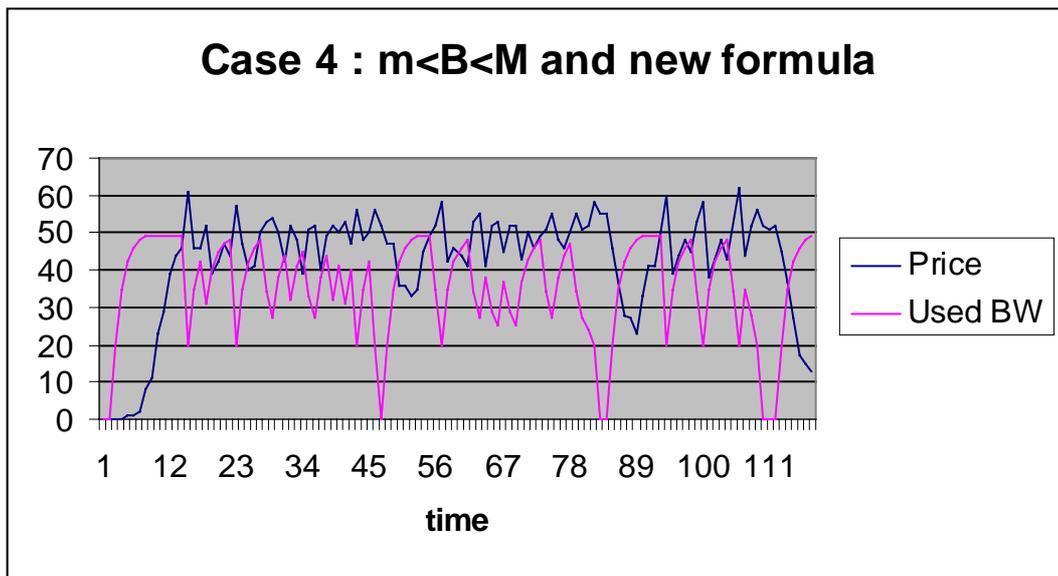   M = 50   P = 50

Figure 26

Surprisingly, connection interruptions still occurred, but much less often than with schemes 1 and 2. Actually this problem stemmed from the fact that agents and network simulation were running at the same "frequency". Thus, a router needed several seconds to empty its buffer, while a router agent had updated its price/output rate at each of these seconds. This led the client agent to overreact and provoked instability. In reality, the time elapsed between 2 agents polling was expected to be much longer than the time needed by a router to empty its buffer.

To test the veracity of our explanation, we reduced our agent polling frequency from once a second to once every 10 seconds, and led the same experiment as in case 4.
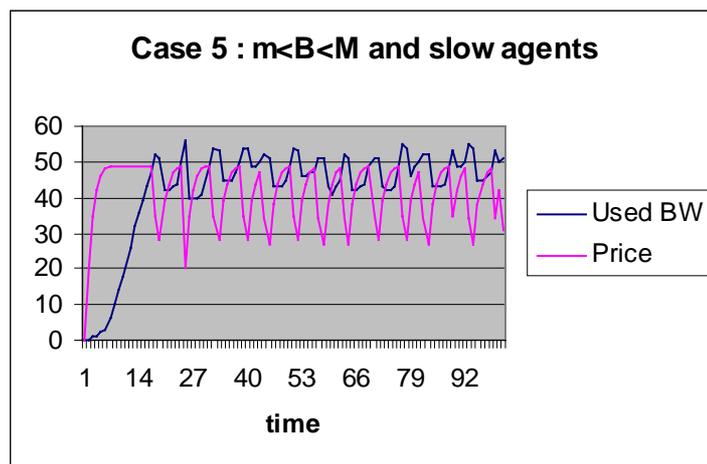

Figure 27

It resulted a greater stability, in the sense that no disruption occurred, as shown by figure 27. The cause is that, after a too high price has appeared and the sender decreased its data flow accordingly, price has time to go under the limit - specified by the client - before the client agent wakes up again and takes corrective action. However, we also observed many losses with the new agents' frequency of action, the reason being that agents are not reacting in real-time any more and network goes thus out of control.

This issue of agent having to react in real-time was a critical one and was addressed again in subsequent phases of our work.

Certainly many other - better - control schemes could be invented to address the problem we faced. But they would lead us to issues regarding controls, while our focus is on network management.

3.7   Phase 7 : Network providers

This phase was a further step into the market paradigm for network management that our project was aiming at.

3.7.1 Routers distribution

Routers - except both gateways - are distributed among <u>providers</u>. A provider is a company or team, in other words, an organization, that is providing network services to clients.
Our idea was to have those providers compete among themselves for having the - in our case, unique - client buying services to them. We compare our providers to local phone companies as if a customer would have at home several sockets - one for each provider - among which she could choose one and plug her phone into it.
Routers are randomly distributed to providers at simulation startup. Via an options file, the experimenter chooses the number of providers. The distribution process is constrained by the fact that a provider must own at least one router on each network layer. Otherwise, it cannot enjoy a continuous path between both clients. To satisfy this constraint, routers must be numerous enough on each layer. From a provider point of view, the NW looks as follows:
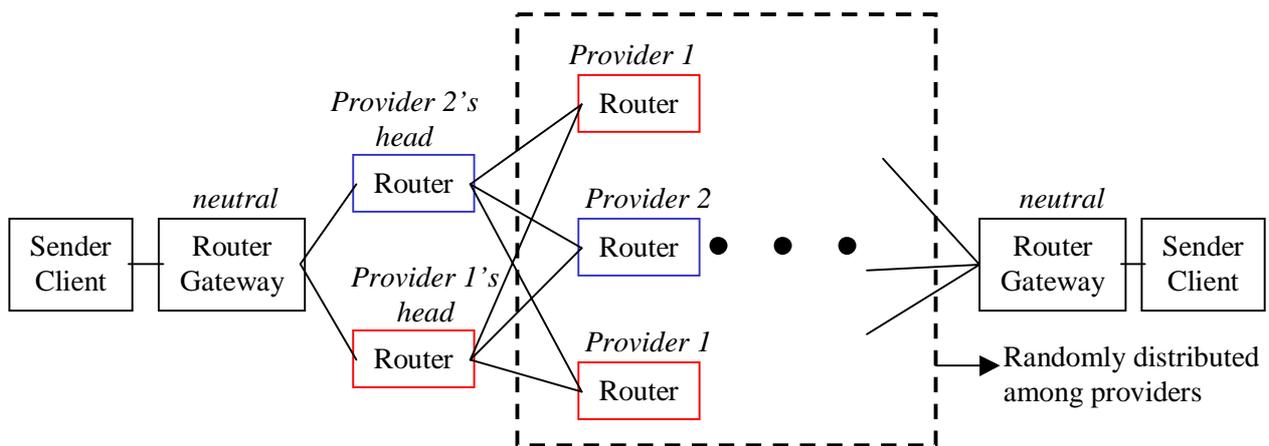

Figure 28

3.7.2 Policy

In this phase, providers are strictly competing and do not collaborate one with each other. Therefore, no data is sent from a router of one provider to a router of another one. Thus, each provider owns a proprietary network.

3.7.3 Implementation

Providers actually just own agents. Those are, in turn, restricting the output of their routers to routers of the same provider. The only interface to be modified was `RouterAgentProfile`, which is updated with two additional methods and two attributes:

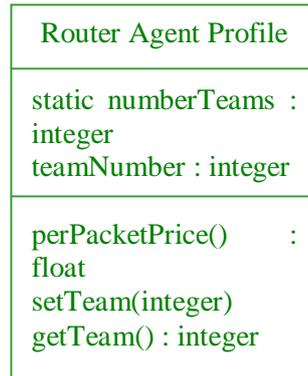| Router Agent Profile |
|---|
| static numberTeams :
integer
teamNumber : integer |
| perPacketPrice()   :
float
setTeam(integer)
getTeam() : integer |

Figure 29

One particular agent is the one attached to the head router, which is the router next to the sender's gateway. There is one head router per provider. Its agent not only fulfills the tasks of another router agent, but also "represents" the provider towards the sender user and its gateway in particular. For instance, it dialogs with the user for connection admission or for billing.

Another particular agent is attached to the sender's gateway. It does not belong to any provider but makes choices of providers on behalf of the user.

In order to implement both of these nuances, we subclassed `RouterAgent`:

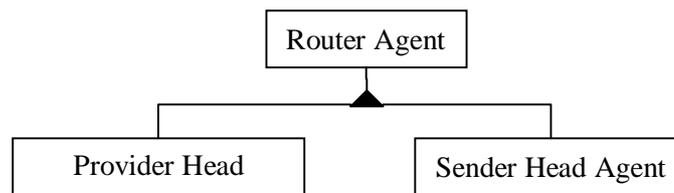Router Agent

Provider Head   Sender Head Agent

Figure 30

The analogy behind the names of the two new classes in that they enable one sender and one provider to face each other to dialog about connections' admission and price.

As a final remark, we did not implement this phase and the subsequent ones, due to lack of time. They merely remain as specifications for future work.

3.8 New price calculation

In this part, we made our simulation more realistic by addressing the issue of the routers' buffer size.

In the model we used so far, a router was massively sending all the data it could – at a given rate – to another router. The latter was either loosing it or storing it in its buffer, in order to forward it later. This scheme worked because, even, when a router was saturated in terms of bandwidth, it still had a large enough buffer to soften the saturation and let enough time for the agent system to react.

But this model raises two problems.

3.8.1 Problems

First, it may cause long delays between sending and receiving the data, because packets may have been stored a certain time in buffers on their way. Furthermore, packets are not guaranteed to arrive at a fixed rate. As a consequence, a high jitter may appear. This would be fine for pure data transmission – like in FTP –, where all data is needed, sooner or later, by the receiver. But, in multimedia applications, jitter has to be avoided: if, when viewing a movie, no data is arriving during several seconds, then there is no use displaying the corresponding images later, in faster motion. Rather, the

screen will remain blank a certain time and the delayed packets can therefore be dropped. In other words, the time to live (TTL) of multimedia packets is short.

Second, an agent is activated every second, while its takes about 1 to 10 seconds for a router to send the full content of its buffer. Thus, the period ratio (agent's period / router's period) $\cong 0.1$, as an order of magnitude. The value of this ratio means that our agent system was so far a real-time system. But most routers, in today's technology, cannot support such active agents. Agents are meant to supervise routers but not to continuously control their behavior.

3.8.2 Solution

We solved the above problems by setting the period ratio to at least 10. Thus, agent's period $> 10 \cdot$ agent's router's period. This can be done by shortening the buffer size to (output rate / 10).

That way enforces a short TTL for packets. The packets cannot remain a long time in buffers, since these have small capacity. Also, the agent system cannot be a real-time system any more, because buffers can become full before an agent had time to react.

3.8.3 Simulation

We ran a simulation with the same agent system as in phase 7 but with a period ratio of 10. As a result, many packets were lost, because agents were too slow to ensure load balancing and because buffers were quickly full. Therefore, a new connection control mechanism was needed.

3.9 Phase 8: Connection control by reservation of bandwidth

This phase addressed the problems raised in section 3.8. In particular, it defined a new connection control mechanism, not based, as before, on occupancy of buffers but on reservation of bandwidth.

3.9.1 Requirements

Previously, a sender was sending as much data as it wanted. The network was then trying to cope with it and, if it could not, to quickly enough send to the sender a negative feedback. We saw (section 3.8) that this mechanism only works when buffers are large enough or when the agent system is a real-time one. But none of those conditions is realistic. Therefore, the sender cannot send more data than the network is ready to accept.

We satisfied this requirement through a bandwidth reservation mechanism. Before a connection of bandwidth b can be established, it has to be checked that it exists a path from sender to receiver along which each node has an available bandwidth of at least b. Then, this path of bandwidth has to be reserved, before the connection is established.

3.9.2 Reservation

The above principle is detailed as follows. For the sake of simplicity, we consider that we have, in this phase, a single provider.

Initially, a user at the sender host asks its gateway's agent for establishing a connection of bandwidth b. the agent responds either negatively or positively by establishing the connection. To do so, it forwards the same request to the provider's head agent. The latter knows the capacity B of its network (see calculus below) and the amount R of bandwidth reserved so far. The connection can be established if b ≤ B-R. If so, it proceeds to the reservation of bandwidth.

Each router keeps track of the amount of its bandwidth that has been reserved so far. Thus, reserving bandwidth along a path simply consists in asking a router whether it has enough bandwidth and, if so, in having it reserve the needed amount. And so on from one layer of the network to the next, until the end host is reached.

Sometimes, enough bandwidth may be available in the network, but not on a single path. In this case, the data flow of a same connection needs to be split on several paths. In our experimental setting, this did not matter, since all paths have same length. In reality, however, this condition does not hold and it

may not take the same time to reach destination for packets going through different routes. As a consequence, losses or jitter may occur.

3.9.3 Network capacity calculation

We saw that a provider's head agent needed to know the capacity of its network. Since all routers' input rates were constant, this capacity could be calculated once forever at initialization. It actually corresponds to the sum of the rates of all routers in the layer that is the bottleneck of the network. We designed a distributed algorithm that makes this calculation.

Let `1, .., n` be the layers of the network.

Let r_{ij} be the `j`-th router agent on layer i.

Let r_{11} be the provider's head agent.

Let r_{n1} be the receiver's gateway's agent.

Let b_{ij} be the maximal bandwidth (output rate) of r_{ij}.

Let m_i be the maximal bandwidth of the network from layer i downstream.

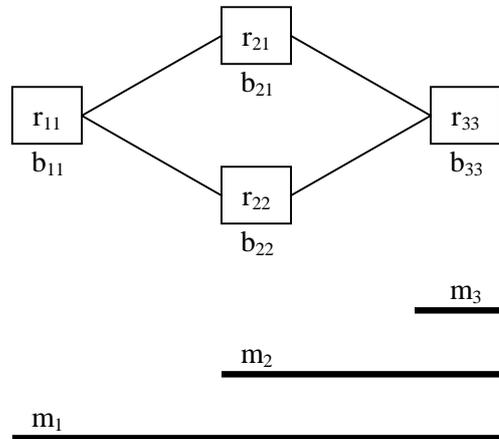⇨ m_1 is the maximal bandwidth of the whole network.

Figure 31

$m_i = \min (m_{i+1}, \Sigma_j b_{ij})$

m_i represents the minimum of the bandwidth of layer i and the one of the network downstream from i.

$m_n = b_{nn}$

m_i is calculated by all agents on layer `(i-1)`.

Finally, $m_1 = \min (b_{11}, m_2)$

The above algorithm is executed for each provider for its own network.

3.9.4 Implementation

Establishing a connection along a path may take some time, since agents need to query each other in cascade. Therefore, it was adequate to use here Voyager's asynchronous method calls. For instance, the `ClientAgent` requests its gateway's agent for a connection. As a return value, it immediately receives a `Result` object, which is just a placeholder for the future arrival of the effective return value. The `ClientAgent` stores this `Result`, attaches a `ResultListener` to it and continues its execution. Later, when the gateway's agent acknowledges the request, the `ResultListener` is triggered and establishes the connection.

3.10 Phase 9: Price-based connection control

This phase introduced price as a connection control mechanism for the connection management scheme defined in the previous phase.
There, when the network was saturated, requests for new connections were simply refused. However, it may be desirable to interrupt existing connections in order to open new, more profitable ones.

3.10.1 Principle

A connection request is written as `c = (b, p)`, where
> `b` is the requested bandwidth
> `p` is the maximal accepted price per data unit

The provider's head maintains 2 lists of connection requests: pending ones and allocated ones. The latter are established connections. An "allocator" moves requests between both lists, that is, suspends or (re)opens connections.
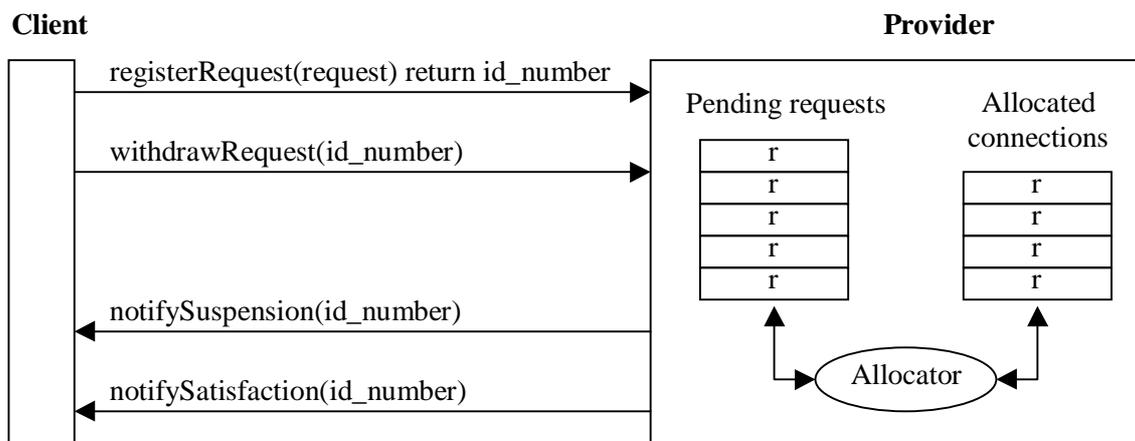

Figure 32

The client - more precisely, its gateway's agent - can file at or withdraw a request from the Provider Head Agent (PHA). A received request is then put in the list of pending ones. In return the PHA notifies its client when it established or suspends a connection.

3.10.2 Allocation

In the PHA, the allocator decides which requests have to be satisfied or suspended at each entry or exit of a request, provoked by a call from the client. The allocation consists in solving the following linear program in N (set of natural numbers):

> $\max \vec{p} \cdot \vec{x}$

> under constraints $\qquad \vec{0} \leq \vec{x} \leq \vec{1}$

> $\qquad\qquad\qquad\qquad \vec{b} \cdot \vec{x} \leq B$

> where $\qquad\qquad\qquad \vec{c} = (\vec{p}, \vec{b})$ = vector of all requests

> $\qquad\qquad\qquad\qquad \vec{x}$ = Boolean vector telling whether a connection is open.

3.10.3 Dialog with user

Finally, the suspension of a connection may be improved by asking first its user whether she wants to raise the price she is paying in order to keep her connection established.

4 CONCLUSION

4.1.1 Summary

In this project, we first evaluated freely available platforms that support intelligent or distributed agents, with a particular attention to Java-based ones. Then, we retained Voyager to develop a network simulation and an agent system to pilot it. Agents were collaborating in order to balance network load, to control multimedia connection admission by dialoging with a user and to simulate competition among network providers. Auction – pricing of QoS – was used as the main coordination mechanism among agents.

Thus, we showed some problems that can arise in distributed network management and implemented ways to solve them.

4.1.2 Learning benefits

Throughout this project, we learnt about five main domains: network management, distributed systems, products' evaluation and application development process.

First, through our readings and especially through our experimenting while developing our application, we tackled various network management issues.

Second, our reading of Russel and Norvig's book [9] showed us the architecture of intelligent agents. Furthermore, we got the opportunity to see several variations of this architecture while reviewing platforms, in particular LALO and ABS.

Third, we successfully used for the first time a distributed environment: Voyager. In particular, we tackled the difficulty of initializing distributed objects as well as terminating their execution.

Fourth, we gained experience at evaluating software products in competition. More precisely, we exercised:

- Distinguishing first what the needs to fulfill are.
- Understanding how the software product works, without extensively experimenting with it.
- Finding next its advantages and disadvantages.
- Finally extracting useful ideas from its functioning, even if we do not retain the product.

Finally, we learnt about leading a development process that was in the same time incremental and object-oriented (OO). Those two notions actually appeared sometimes contradictory. On one hand, OO design requires having an idea of how we plan to develop our application; but, often, this idea is lacking when doing iterative prototyping and experiments like we did. From the experience of this project, we concluded that a general idea of the evolution of the prototype is anyway needed to serve as a base for the object model. The latter, in turn, must be as much scalable as possible, that is, accommodate further developments. That way, full benefit can be taken from the easiness of maintenance offered by an OO program.

4.1.3 Future work

Our work on this project could be immediately continued by implementing the specifications we gave in phases 7, 8 and 9.

Furthermore, several other aspects could be studied.

First, one could investigate ways for the agent system to allocate to connections more bandwidth than is available. In fact, a connection - thanks to data compression techniques - does not use a constant amount of bandwidth over time. Moreover, it could be assumed that not all connections will use all their available bandwidth at the same time.

Second, our network did not take transversal traffic into account. This kind of network traffic goes from and to end hosts that are not ours but flows through some of our routers. This situation could be simulated by having variable output rates in routers. It would make the connection allocation task much more difficult since no bandwidth could be guaranteed once for ever. Rather, the agent system would have to periodically react to network events.

Third, we suggested a connection bidding strategy for a network provider. Better strategies could certainly be invented.

Fourth, we used a simplistic network topology - a bipartite graph. Other - regular or not - configurations could be experimented with.

Finally, our agent system and real network could be bound to a real multimedia application rather than to simulated end hosts.

Also, our work is an adequate starting point for another semester or diploma project, focused on agents, for the following reasons:

- We already chose a platform.
- We implemented a network layers in two different ways.
- We built the core of an agent system.
- All programming was object-oriented, thus making classes easily reusable.

4.1.4 Acknowledgments

5 REFERENCES

[1] ICMAS'95, <u>Proceedings of the 2nd International Conference on Multi-Agent Systems</u>, The AAAI Press, 1995

[2] ICMAS'96, <u>Proceedings of the 2nd International Conference on Multi-Agent Systems</u>, The AAAI Press, 1996

[3] David Flanagan, <u>Java in a Nutshell</u>, 1st ed., O'Reilly, 1996

[4] J.-P. Martin-Flatin, S. Znaty and Hubaux<u>, A Survey of Distributed Enterprise Network and Systems Management Paradigms</u>, JNSM, 1997

[5] Patrick Niemeyer, Joshua Peck, <u>Exploring Java</u>, 2nd ed., O'Reilly, 1997

[6] ObjectSpace, <u>Voyager, Core Technology User Guide</u>, Version 1.0.0, 1997

[7] Marshall T. Rose, <u>The Simple Book, An Introduction to Management of TCP/IP – Based Internets</u>, Prentice-Hall Series in Innovative Technology, 1991

[8] James Rumbaugh et al., <u>Object-Oriented Modeling And Design</u>, Prentice-Hall, 1991

[9] Stuart Russel, Peter Norvig, <u>Artificial Intelligence : A Modern Approach</u>, Prentice Hall, 1995

[10] Andrew S. Tanenbaum, <u>Computer Networks</u>, 3rd and international ed., Prentice Hall, 1996

[11] Michael J. Wooldridge, Nicholas R. Jennings, <u>Intelligent Agents</u>, Lecture Notes in Artificial Intelligence 890, Springer-Verlag, 1995

APPENDIX A: DOCUMENTATION ABOUT LALO

The following documentation can be obtained at the URL: http://www.crim.ca/sbc/lalo