Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

About adding utility and usability to FDNet
application au domaine de la construction

Achbany, Youssef; Jadoulle, Jérôme

Award date:
2004

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 25. Apr. 2024

Facultés universitaires Notre-Dame de la Paix – Namur
Institut d'Informatique
Année académique 2003 – 2004

About adding Utility and Usability to FDNet A Flat Distributed Network Architecture

Youssef ACHBANY
Jérôme JADOULLE

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique

Facultés universitaires Notre-Dame de la Paix – Namur
Institut d'Informatique
Année académique 2003 – 2004

About adding Utility and Usability to FDNet
A Flat Distributed Network Architecture

Youssef ACHBANY
Jérôme JADOULLE

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique

VTLS 20001586

Résumé

Ces dernières années, la robotique a connu d'énormes progrès, tant au niveau matériel que logiciel. La robotisation croissante des jouets, qui dernièrement abouti à la création du petit chien Aibo, augmenta l'engouement de tous : utilisateurs, concepteurs et vendeurs.

Au niveau logiciel, une grande partie des travaux a été consacrée à la conception d'architectures communes, utilisables sur nombre de robots différents, simplifiant ainsi la communication entre ces derniers et la fabrication de leurs composants.

Plusieurs grandes institutions et entreprises poursuivant ce but ont donné de bons résultats. Mais les avancées conviennent surtout à une robotique axée sur le jeu, principal moteur d'améliorations. Elles ne peuvent, actuellement, pas être transposées à d'autres milieux tels que celui de robots sauveteurs, évoluant dans des environnements chaotiques pour sauver des vies.

C'est en vue de palier à ce manque que l'L'I.R.S.I, « The International Rescue System Institute », mène depuis plusieurs années des recherches dans ce domaine. Ces recherches de création d'architecture commune à des robots de types différents donnèrent naissance à FDNet, diminutif de « Flat Distributed Network Architecture ».

La première partie de ce mémoire introduira les travaux effectués en matière d'architecture commune qui ont inspiré FDNet. FDNet, dont les concepts et particularités vous seront présentés dans une deuxième partie, conjointement à l'état d'avancement des recherches. Dans la troisième partie, nous parlerons de notre contribution à cet ambitieux projet qu'est FDNet, à savoir le rendre plus utile et utilisable pour des sauveteurs, leur permettant ainsi de l'utiliser sur le terrain.

Abstract

In the past years, robotic science has progressed a lot, in both hardware and software levels. The increasing robotisation of toys, which has recently led to the creation of Aibo, increased the interest of everybody: users, creators and vendors.

At software level, a great deal of work has been concentred on the conception of common architectures which, being usable on a lot of diverse robots, simplified the communication between them and their components' production.

Various institutions and enterprises following this aim have provided some good results. But the advancements mainly concerned game-centred robotics, which is the main robotic enhancement engine. Theses advancements can't, for the time being, be transposed to other fields such as the one concerning rescue robots; robots evolving in chaotic environments to save lives.

It is to make up for this lack that the I.R.S.I, « the International Rescue System Institute », is working on this field for some years. Theses researches about creating common architectures for different types of robots gave birth to FDNet, acronym for "Flat Distributed NETwork architecture".

The first part of this thesis will introduce the researches made upon the common architectures which inspired FDNet. FDNet, whose concepts and particularities will be presented in a second part, also describing the level of advancement of the researches. In the third part, we will talk about our contribution to this ambitious project that FDNet is, namely to make it more useful and usable for rescuers, allowing them to use it on the real world to try to save more lives.

Acknowledgments

First of all, we would like to thank Mr. Schobbens for having found us the place of the thesis and Mr. Tadokoro for its subject. Mrs Tadokoro was of an incredible help for all the administrative work too. Without theses three people, setting up this thesis would have purely and simply been impossible.

Working for Mr. Tokuda and his RoQ team was a pleasure too. The differences in our cultures and spoken language were all but problems thank to them. Working on enhancing FDNet would not have been as pleasant and instructive if theses people hadn't been present.

We are grateful to all the students we could meet in the laboratory. They were always supporting us and have been at the base of some of the best moments we could have in Japan. Thank you Takuma, Takumi, Kaoru, Ulrike, Akazawa, Minobe, Aki, Takemura, Nobuhiro and all the others.

Great thanks to I.R.S.I secretaries which helped us each and every time we were facing difficulties. More particularly, let us thank Tomoko, who was a guide, a translator and an animator but also, more than anyone else, a true friend. We will never forget what you have done for us Tomoko. Itsumo Arigato.

Mr Nicolas Lambot, one of our best friends for a long time, was also with us there and shared all the good moments with us while helping us during the more difficult ones. Nicolas, you know it, no word is good enough to explain how we feel about you.

Last but not least, we would thank our families and our friends who have supported us through our studies and without whom arriving here wouldn't have been possible at all.

Minna-san, dômô arigatô gozaimasu!

Table of content

Table of figures

Table of Index

Glossary

Cache: A temporary storage area for frequently-accessed or recently-accessed data. Having certain data stored in cache speeds up the operations of the program.

Connection: An interaction happening between a Data and a Relation. Connections can either be Readers or Writers.

Data: Any piece of information that can be used by Relations.

Event: An occurrence that is significant to a program, and which may call for a response from the program.

FDNetwork: A construction of Network Entities with the FDNet architecture.

Graphic Panel: The frame, in the Human Interface, where the Graphic Representation of the edited FDNetwork is displayed.

Graphic Network: File containing all the positioning information concerning the graphic representation of an FDNetwork. Upon loading of an FDNetwork, the Human Interface searches for this file in order to display the FDNetwork in a correct way in the Graphic Panel.

Network command: There are many network commands - connect Reader/Writer, disconnect Reader/Writer, create Data/Relation, destroy Data/Relation ... A network command includes connection information that describes an event notification.

Network Definition: A succession of Datas and Relations, themselves followed by a list of the Connections that theses Nodes have between them.

Network Entity: A Network Entity is either a Node or a Connection.

Network state: A network state represents either the value of a Node at a given time or the network connection information. Network connection information is a "network change" event and says, for example, that a Data Node was disconnected or suppressed.

Neural network: A neural network is an interconnected assembly of simple processing elements, *units* or *nodes*, whose functionality is loosely based on the animal neuron. The original inspiration for the technique was from examination of bioelectrical networks in the brain formed by neurons and their synapses. The processing ability of the network is stored in the inter-unit connection strengths, or *weights*, obtained by a process of adaptation to, or *learning* from, a set of training patterns. In a neural network, simple nodes (or "neurons", or "units") are connected together to form a network of nodes, hence the term "neural network".

Node: Either a Data or a Relation.

Reader: A link between a Data and a Relation allowing the Relation to read the value of the Data it is connected to.

Relation: A processing agent, whose aim is to take some Data in entry and to compute it in some way to create new Data.

Retro engineering: (or Reverse engineering) is the process of taking something (a device, an electrical component, a software program, etc.) apart and analyzing its workings in detail to understand how it works.

Robot: A mechanical device that performs a task that would otherwise be done by a human. Robots can be useful for jobs that are boring or dangerous for humans to perform. The simplest robots are capable only of repeating a programmed motion; the most sophisticated models can use sensors and artificial intelligence to distinguish between objects, understand natural language, and make decisions. Robots can be programmed or operated by remote control.

Sensor: An electronic device used to measure a physical quantity such as temperature, pressure or loudness and convert it into an electronic signal of some kind (e.g a voltage). Sensors are normally components of some larger electronic system such as a computer control and/or measurement system.

Server: The entity in a client/server architecture that supplies files or services. The entity that requests services is called the client. The client may request file transfer, remote logins, printing, or other available services.

Thread: In programming, a thread is one part of a larger program that can be executed independent of the whole.

Human Interface: (or User interface) The means by which a user interacts with a computer. The interface includes input devices such as a keyboard, mouse, stylus, or microphone; the computer screen and what appears on it; the way commands are given, etc. With a command-line interface, only text appears on the screen, and the user must type in commands; with a graphical user interface, windows, mice, menus, and icons are used to communicate with the computer.

Viewer: It is an important application that allows users to analyze the Network state in order to follow its evolution. This application was done by Mr. Lambot[1] and is integrated in the Human Interface.

By "**Writer**", we imply a link between a Data and a Relation allowing the Relation to write the value of the Data it is connected to.

[1] [Lambot 2003]

PART 1 - FDNet

Chapter 1: Introduction to FDNet

I. Concept :

The International Rescue System Institute [I.R.S.I.] is a Japanese organization that works on the robots field, particularly on rescue robots. The role of this kind of robot is to help rescuers finding victims after an earthquake or a disaster that destroys the environment, making it dangerous. The world in which the rescue robot operates is thus very complicated. Regarding this environment, the topography is unique in every place, every time. Unexpected situations can always occur while the robot works among the debris. Even a situation envisaged is rather complicated and can't be apprehended perfectly. The first issue is how the rescue robot can cope with that complexity in order to act in such environment.

It is not possible to prepare the robots to fit each kind of environment, one after the other. Because of various trade-offs, the conception of a complete robot behavior is impossible. Therefore rescue robot must have some form of intelligence, a software architecture that combines the various fundamental technologies and new skills dynamically learnt in that specific place. This architecture is FDNet, a Flat Distributed Network Architecture.

FDNet is especially based on three previous architectures, namely ORiN, ORCA and Open-R. In fact, there were many previous studies about robot architecture, but none of them could respond exactly to our expectations. These architectures have no structure to perform dynamic self-organization or dynamic reconfiguration, making them unusable for the rescue robots.

Instead of trying to modify theses architectures, I.R.S.I researchers took the interesting characteristics from them to create a specific and common architecture for all rescue robots: FDNet. Like its bases, FDNet is a flexible, extensible and generic architecture.

A. Flexibility:

Because of the complexity and the differences between environments, a rescue robot can't be totally autonomous. Under these conditions, the robot needs to be ordered and monitored by a human operator. But, it isn't realistic to think of a human always giving detailed movement orders to the robot. It is necessary for the robot to be half autonomous. By half-autonomy, we imply that the robot must be able to perform advanced tasks given simple orders, but to wait for specific ones when it falls under conditions where human judgment becomes necessary.

This is the reason why a common software architecture that shows flexibility regarding both software and hardware is needed, the goal being to be able to describe an intelligent system with that architecture.

To build this flexibility, the system must be able to do three things:

1. First, it has to be based on basic behaviors, not too simple to bear a minimal meaning but not too complicated to be easily ordered.
2. Secondly, the system must provide an intelligent ordering system to be able to construct advanced behaviors from the simpler ones.
3. Finally, the system must be able to create new behaviors according to the environment in which it is executed. Being able to learn from its own work will greatly improve system's performances.

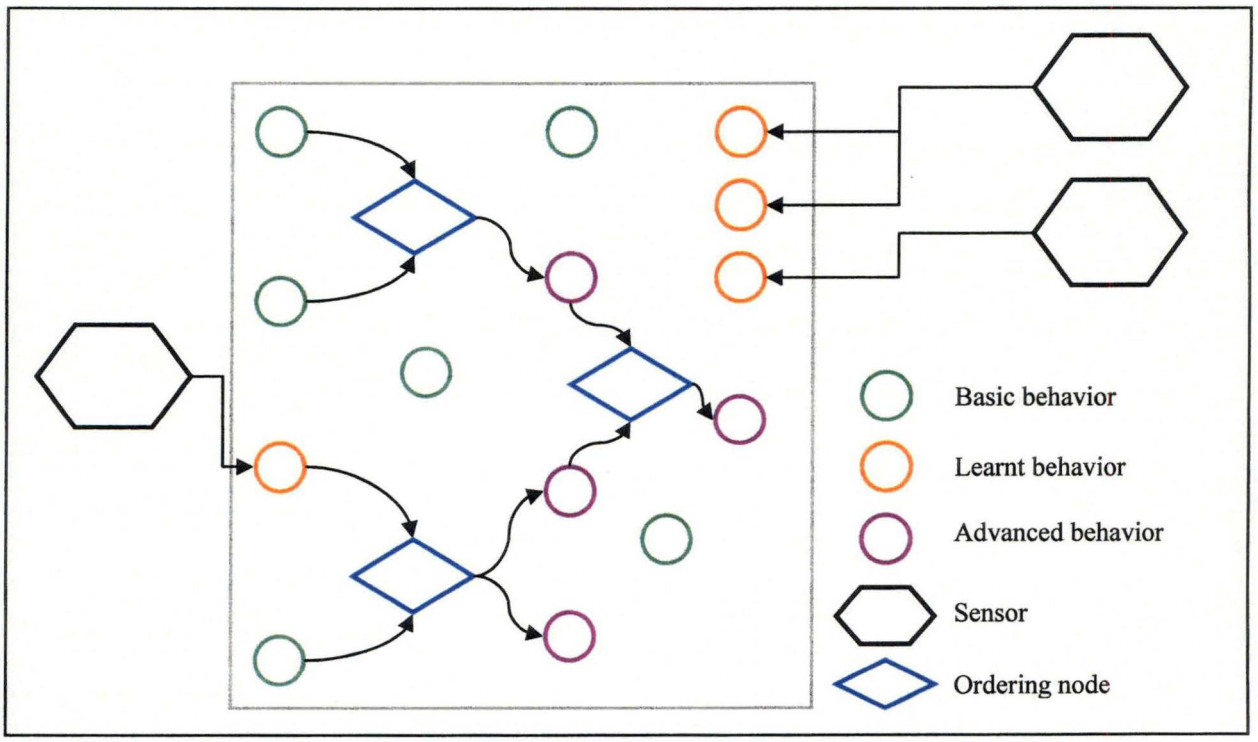


Figure 1: FDNetworks' flexibility property.

By creating advanced behaviors, based upon both basic and learnt ones, the robot will be able to find answers to problems specifically encountered on the field. Robot's reactions will then be exactly fitted for the actual environment he is maneuvering in.

B. Extensibility:

FDNet is created by using various layers. By dividing the architecture in different layers, it is easy to limit the impacts of future changes or improvements to the layer concerned by theses changes. This extensibility allows, for example, equipping robots with the most suitable sensors anytime, changing them as the environment change.

C. Generic architecture:

I.R.S.I researchers wanted to create a common architecture usable for all rescue robots whatever their type, with a common protocol allowing them to exchange data. This is FDNet's most important feature.

Rescue robots' ultimate goal is to help rescue injured people. Therefore, the best way to ensure that victims can be found and saved has to be put in place. Finding victims means to be able to get the best and the most complete information about the environment and to be able to discover the injured people using this information.

Various proposals have been made so far. For example, creating a group of robots consisting of "crawler robots", "legged robots" and "flying robots" has been thought of a lot. The flying robots provide general information about the environment and supervise crawlers and the legged robots, whose more specific researches will ensure victims can be found.

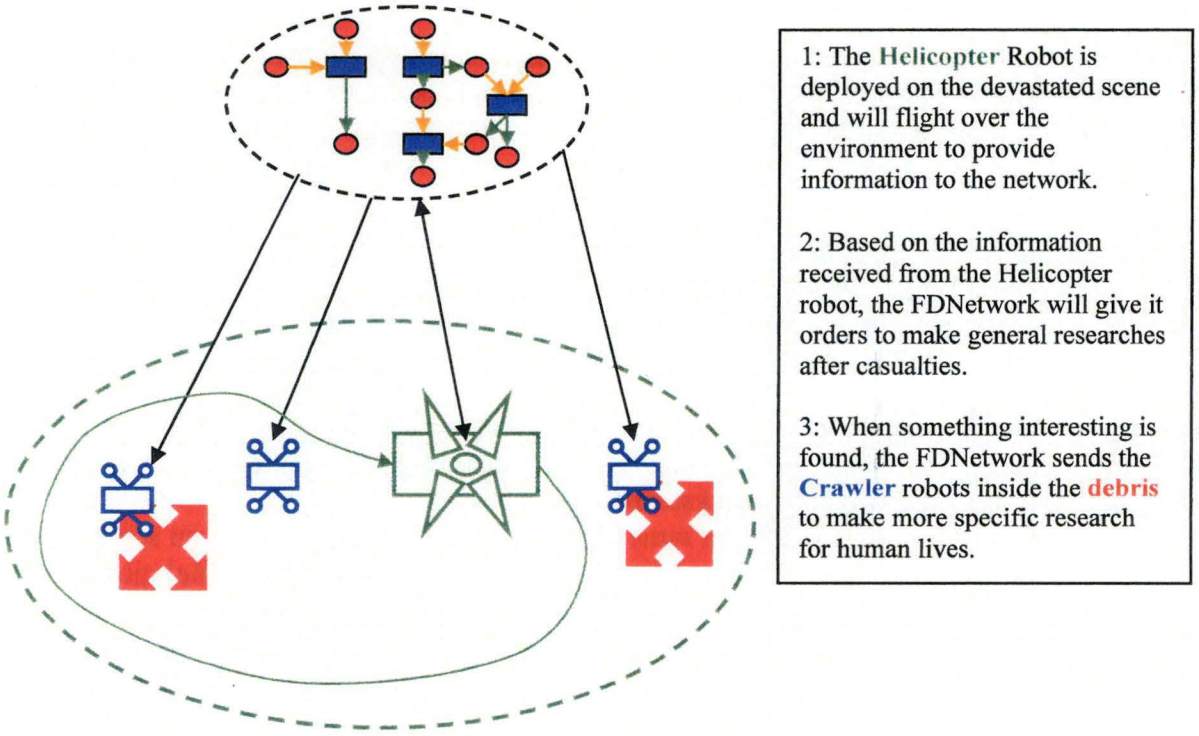


Figure 2: An example of cooperation between different kinds of robots

However, it is currently difficult to exchange information between these robots because each one has its own protocols and architectures. If all of them where using the same common protocol and the same architecture, not only would it be possible to make each robot able to discuss with each other, but it would also be possible to reduce their creation time, lots of complex problems encountered by a team being already resolved by other ones.

II. FDNet in more details:

Now that you have been introduced to FDNet's purpose, concept and origin, we will present each of its particularities in more details, basing ourselves on FDNet's definition.

> FDNet is a **F**lat **D**istributed **Net**work architecture based on the human imitation model, allowing a pool of robots to cooperate in order to help rescuers to find victims in case of disasters.

A. FDNet is a Flat Distributed Network architecture:

By Network, I.R.S.I Researchers imply that an information network is used to create robot's intelligence. The robots having to be half autonomous, they have to be able to take some decisions like, for example, determining which direction is better to reach injured persons in a fragile terrain. To represent this intelligence, a neural-like network is used.

FDNetworks are made of two main components: the **Nodes** and the **Connections**. While Nodes consist of either raw information or processing objects, Connections are to be viewed as links between the Nodes, allowing the processing objects to access the information they need.

There are two kinds of Nodes:

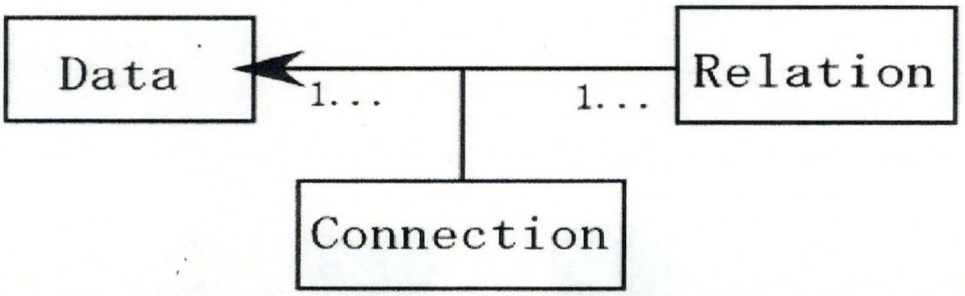
1. **Data Node**[1]**:** This kind of Node represents Network's raw information, which can either arise directly from the Base Network itself or from the environment robots evolve in, by using sensors and cameras. New Datas can also be processed by using Relation Nodes.
   All information contained in the Network is considered to be a feature. In other words, any data's value is decided in the same way: the device-level feature is decided in the same way that the system-level feature is.
2. **Relation node**[2]**:** Bears the same meaning in FDNet than the neuron specified in the recognition model or in neural networks. By using input Datas, Relation nodes can also calculate new Datas' value. In this case, the Relations will work with the help of "servants": agents having specific functions. Relations can only calculate the value of directly linked Datas. But through the use of servants, they can access FDNetwork's whole structure. This way, Relations can perform Network' dynamic self-organization / reconfiguration.

Relations and Datas can have parameters. These parameters can be used to initialize a Node and to influence its behavior. Note that parameters are only represented as strings in the FDNet architecture. It means that they have no type, and don't bear any meaning by themselves. The Nodes using them will give them their meaning.

[1] To refer to Data Nodes, the term « Data » will also be used.
[2] To refer to Relation Nodes, the term « Relation » will also be used.

Relation and Data Nodes are linked by Connections, which can be of two types:

- Reader Connection: A Connection between a Data and a Relation where the relation can read the data's value.
- Writer Connection: A Connection between a Data and a Relation where the relation can write a new value in the data.



Nodes can be linked with as many Connections as needed but a Connection, whatever its type (Reader or Writer) always connects a Data Node to a Relation Node.

Figure 3: An entity/ Association diagram representing the relations between Nodes and Connections

The example below will show you a basic representation of an FDNetwork to help you visualize how its components are organized in order to create Robot's intelligence.

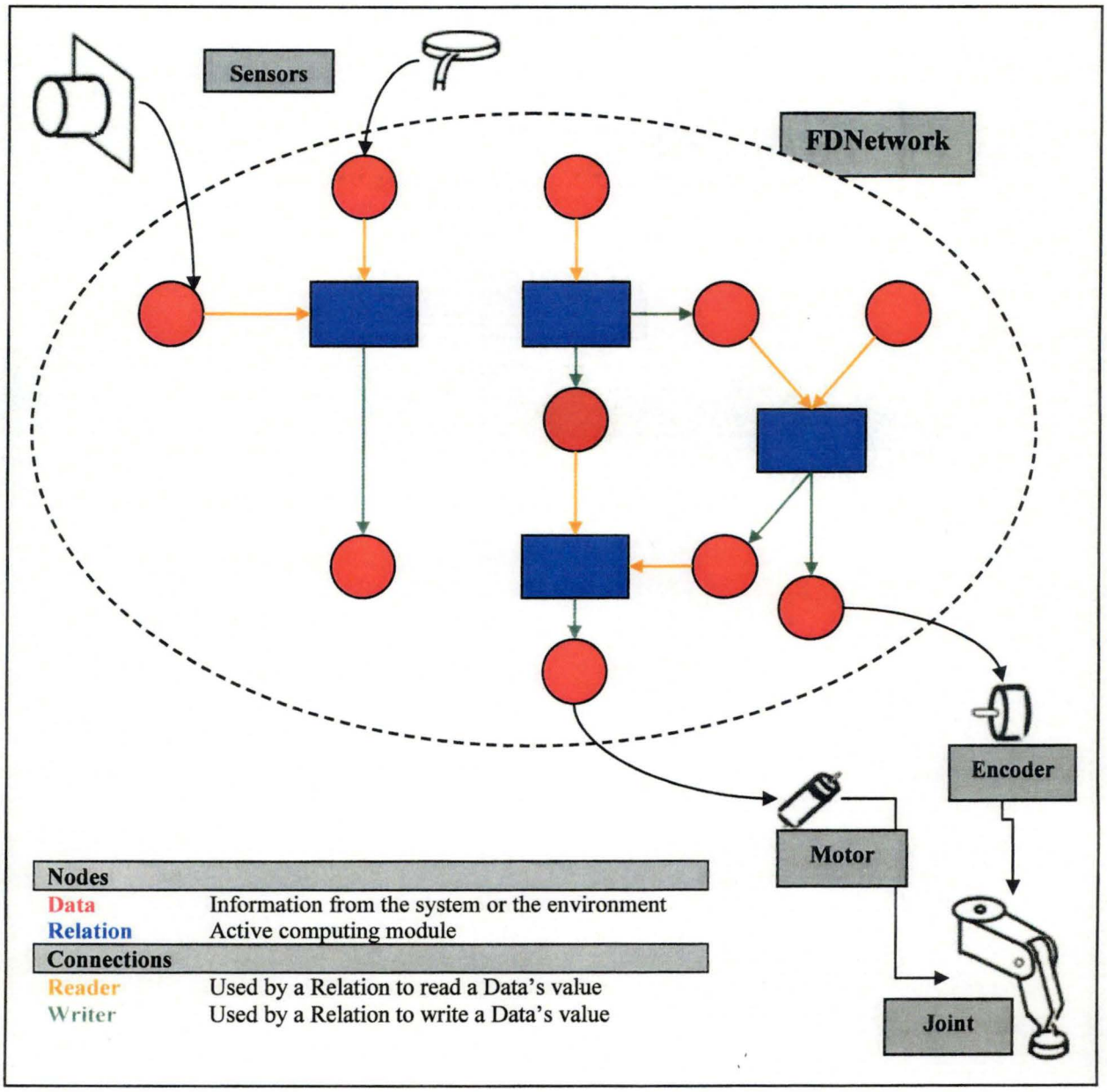


Figure 4: Basic representation of an FDNetwork

The example below will show you the dynamism that exists in FDNetworks. It represents a sub-network that can generate the motion of a robot using the cooperation of many movement formation agents. The movements dynamically created are computed basing on the information received from different kinds of sensors, whose aim is to provide real-time information about robot's condition.

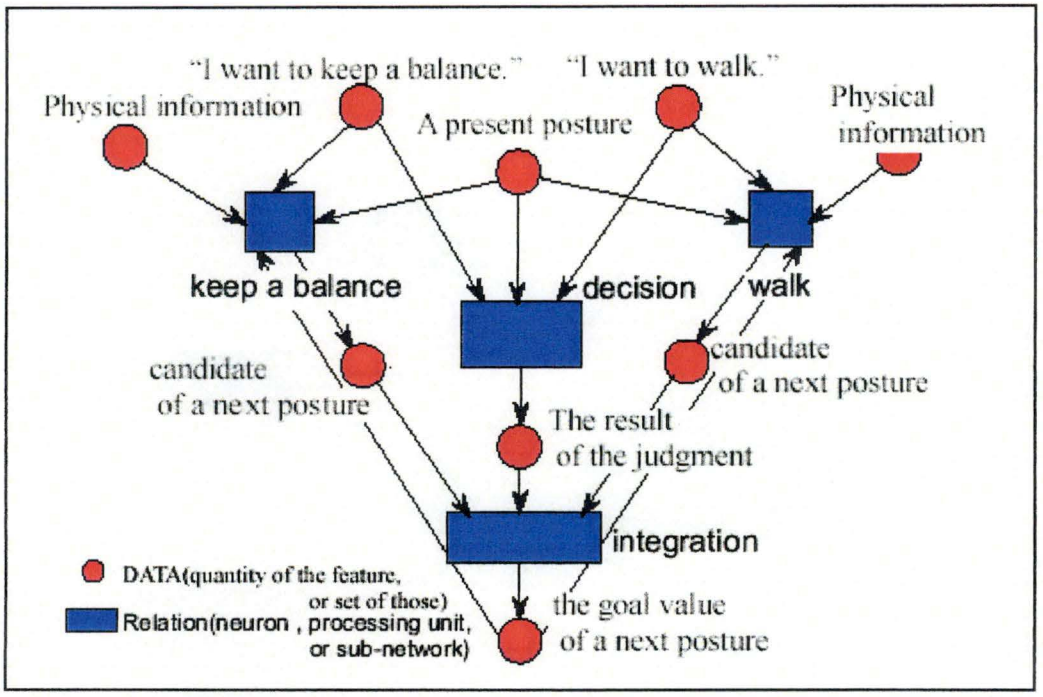


Figure 5: A sub-network that can generate the motion of the robot

The Datas in entry of this sub-network ("Physical information", "A present posture") give information about robot's current physical state and posture. Aside from theses Datas, coming from real-time sensors, the sub-network also needs to know robot's intentions ("I want to keep a balance", "I want to walk") in order to decide robot's next posture.

Theses Datas are read by different Relations ("keep a balance", "walk") who independently calculate a candidate value for the next posture. Theses candidates will, along with a Data representing the result of robot's judgment about moving or not, will be integrated to calculate the next posture the robot has to take.

Note that all is not necessary white or black. The decision can be, for example, that the robot has to move, but just a little. In this case, the integration will give a higher importance to robot's balance but will not just stand still for all that.

In fact, as all Relations are just programming, anything wanted can be computed. The main problem faced by researchers is not to compute Datas the way they like but to possess the right information at the right time and to know exactly what to do with it in order to create a usable output.

What we want to say here is that, although intelligence programming can be somewhat difficult to achieve, the researchers have to focus more on what intelligence is to be

programmed instead. This is one of the reasons why our participation to the FDNet project is, as you will see in the following chapters, an important one.

In the example above, it is important to note that "The goal value of a next posture" is, in fact, the same data as "a present posture". This Data just receive a new value from one of its "child" Relation.

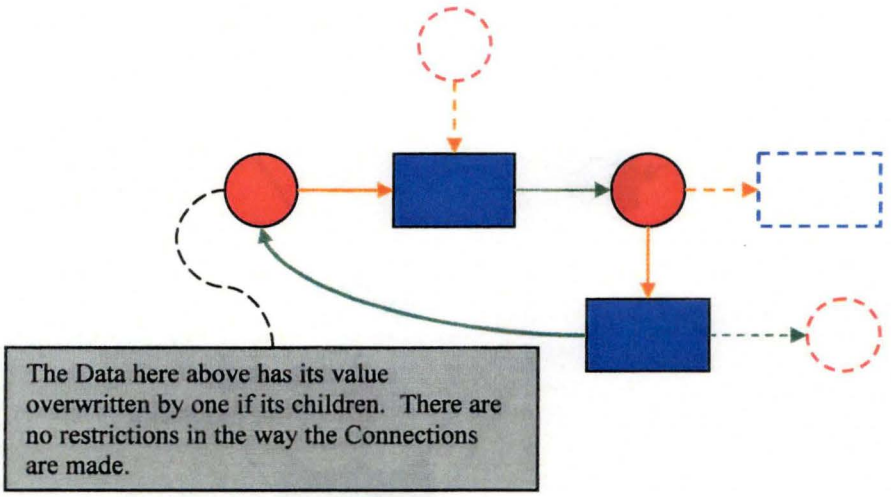


Figure 6: FDNet's "Flat" characteristic

This fact represents the "**Flat**" characteristic in the definition. The Networks are said to be "Flat" because, through the use of Connections, any Relation can be connected to any Data, whatever their meaning. Inside an FDNetwork, there is no hierarchical structure or different component levels. All Nodes and Connections have the same status in the Network and are processed in the same way.

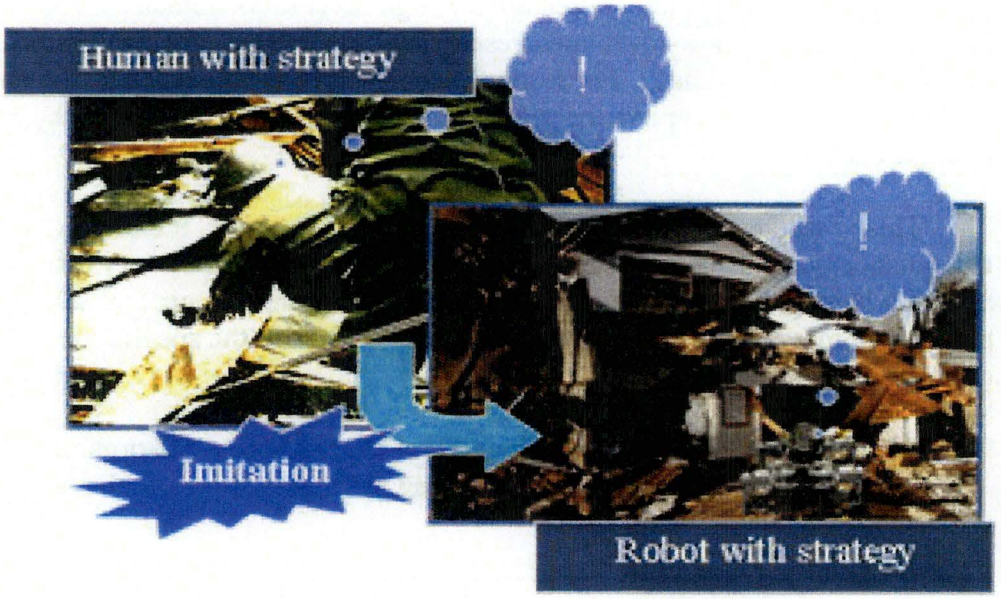
Self-organization comes from this fact. New features can be dynamically constructed using information coming from any Node in the Network, whatever the type of information they bear.

The term "**Distributed**" is used to point the fact that, in FDNet, a group of robots and machines can work together from the intelligence of a single Network. It means that each robot doesn't especially have its own FDNetwork but can share it with other robots, all contributing to Network's global intelligence.

For example, in the Figure 2, the flying robot can provide information to the network that will be very useful for the crawler robots to narrow their researches.

B. FDNet is based on the Human Imitation Model:

Before the specification of FDNet's architecture, the researchers have adopted a recognition model. In FDNet project, the outline of human imitation is assumed for the rescue robots to solve most of the problems. They have done many researches in which they transposed some human features to a robot and studied the application to the robot. Next is an overview of this research.



Human Imitation Model is used in FDNet for more than one purpose. First of all, Network's intelligence is based on the neural model, meaning that human intelligence is, to a certain extend, imitated by FDNet.

After researches about what rescuers do when working in devastated environments, I.R.S.I. Researchers found that humans do two particular movements when moving around to find victims: first, they tap the ground where they are planning to move in order to see if it is stable and after, they slowly move their weight forward and analyze their own position in order to be sure that the new position they are in is stable too.

Following the "Human Imitation Model", I.R.S.I researchers tried to transpose this information about movement inside FDNet model. Conclusions made clear that it is not only important for a robot to move around to imitate human rescuers actions but also, it is necessary to take care of the way it moves and to analyze the "feeling" he has about its own position in order to take the best movement choices.

To make the best work possible here was quite important because moving on devastated environments is one of the main tasks of FDNet robots. If this task cannot be performed correctly, none of the following ones will be possible to achieve.

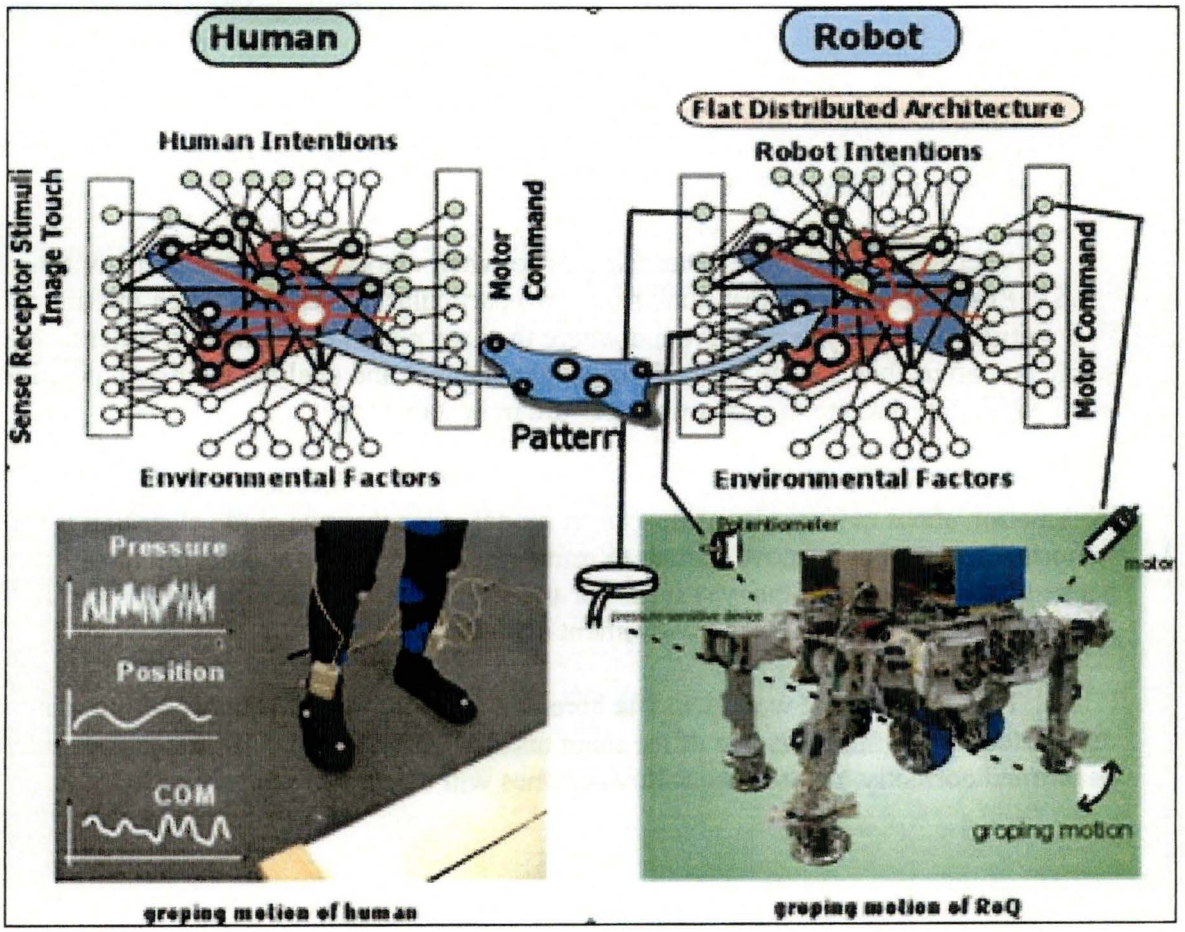
In order to achieve this requirement, FDNet model was expanded to include an "Active movement sense". This means that, gathering information coming from sensors (tapping the ground in order to see if it is stable), motion, intentions and environment, FDNet robots develop a perception-like ability. Their movements thus become more precise.

The information acquisition and recognition was realized by creating an "FDNet neuron formation" efficient for the specified purpose. This formation can be updated each time a common recognition model between robots and humans can be found.

Note that what are transposed are the high-level tasks humans execute. By doing this instead of transposing low-level ones, researchers can enhance their robots with abilities specifically thought for them. For example, four-legged robots (such as the one you see in the picture below) will never move the same way as human being do even if the intentions about moving are the same.

This is where self-organization can show its true power. It can become the base for great enhancements because it can produce features that would be very difficult to find while examining Human beings. Indeed, if it is quite easy to create a general scheme of the way a human being moves, it is a lot more complicated to find all the specificities of the same movement.

Basing on the transposition of high-level Human tasks, self-organization will be able to reconstruct a "Human-like model" by finding an organization which allows producing the same results as a human being.



C. FDNet's aim is to help rescuers to find victims in case of disasters:

FDNet is, as said before, a common architecture especially created for rescue robots. Indeed, previous studies on robot architectures were for entertainment (Open-R) and industrial robot (ORiN and ORCA) but none of them could be easily transposed to be used with rescue robots, theses ones having particular needs.

As rescue robots are the base of the architecture, FDNet has been specifically created to ensure it can respond to theses robots needs and thus is able to solve the important problems presented first[1].

[1] Report to FDNet's concepts in this chapter.

III. Current FDNet implementation:

First of all, the creation of a complex rescue robot has been started some years ago and is still under development. The rescue robot that can be seen here below, named RoQ (Robotic Platform for Rescue), serves as the base platform used to construct FDNet and test its abilities.

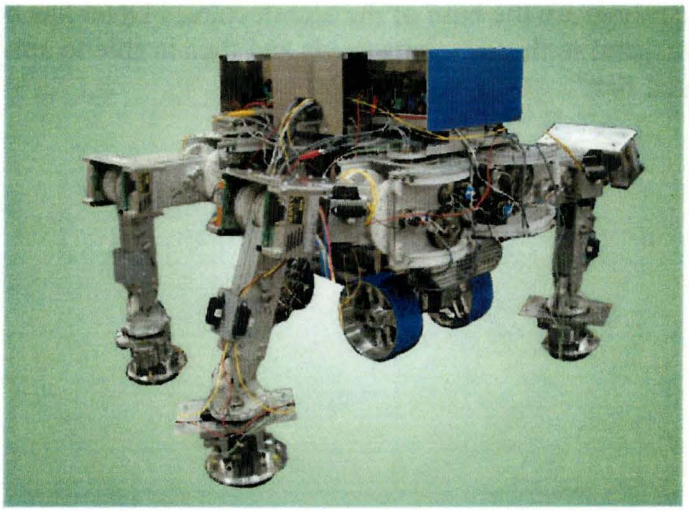


Figure 7: The robot RoQ

FDNet implementation started at the same time. It is still a work in progress and much time will be necessary to have it work perfectly, though a beta version is already showing some of its potential.

IV. Development choices:

A. RoQ's base Hardware:

- Quadruped robot TITAN-VIII
- PC (Pentium-III 800MHz, 512MB RAM)
- Device Network
  - Angle of inclination meter, Infrared rays sensor
  - Ultrasonic sensor, CCD camera
- Tactile sensor at the sole
- Wheel movement mechanism
- Ankle mechanism

B. FDNet environment:

- The Linux operating system (kernel Linux 2.4.4)
- Real time extension RTLinux 3.1
- Java language(Java2 SE 1.4.1 01)
- postgreSQL DataBase (V. 7.1.3)
- CORBA Middleware (OpenORB 1.2.0)

C. FDNet A.P.I :

Without entering too deeply in the details, we can say that FDNet's API is defined within the following layers:

- The Network layer (CORBA): It is the place where Connections and information transmission are implemented.
- The Programming language layer(Java, C++): It is where the real functions and behaviors of the Relation and Data objects are implemented.

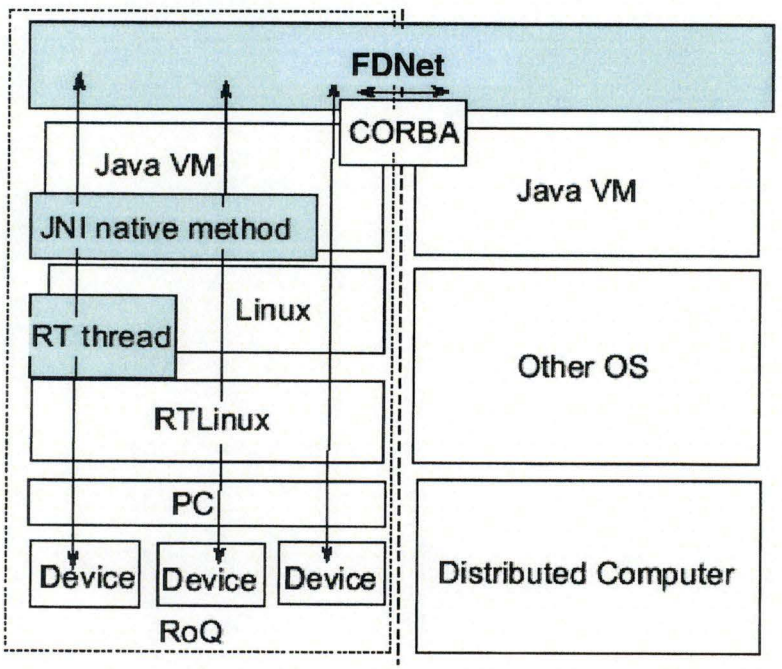


Figure 8: Implementation of FDNet in various layers

While Java/CORBA was selected because of Java's portability feature the implementation on RTLinux is used to control the parts that are time critical (i.e. a defined response time is expected). For example the control of each RoQ Robot's joint is mounted as a real-time task of the RTLinux and can be available directly for the other components.

V. Conclusion:

Following FDNet's definition, we gave you an idea of FDNet's aim, its capabilities and its potential power. Though it is still far from being a completely usable architecture, its implementation is advancing well and tests that have already been made, mostly about human imitation model, are satisfactory enough to give a strong will to further continue its development.

Of course, being under construction, a lot of parts still need improvements and enhancements (and, as you will see in the following chapters, some very interesting enhancements have been thought of) but the base is well defined already, putting a powerful rescue robot architecture at researchers disposal.

In the near future, the test robot – RoQ – will be improved and more and more capable prototypes will be created. At the same time, FDNet will have its bugs corrected and the tests between the network architecture and the robot will lead to providing answers to some of the questions at the base of FDNet development.

Later on, FDNet's distributed component implementation will allow seeing how a pool of robots using the same intelligence can perform and will give life to lots of future researches.

Adding to this all the ideas that nobody can think of at the time being, we are certain that FDNet research holds a big potential which will clearly lead to important advancements, at least in the rescue robot field and probably in general robotic science too.

Chapter 2: State of the Art

FDNet's architecture is based on 3 main architectures: ORiN, Open-R and Orca. FDNet researchers have deeply studied and analyzed theses architectures to find their advantages and weaknesses in order to improve FDNet's conception.

The main ideas of theses three architectures will now be presented to give you a glimpse of FDNet's origins and working.

I. ORiN: A common object model for robotic systems

As a three-year project of NEDO (New Energy and Industrial Technology Development Organization), JARA (Japan Robot Association) started "the Development of a standard interface that provides a unified access mean" from 1999. The outcome of this project was ORiN[1] (Open Robot interface for the Network). In other words, ORiN is a system for standardizing communications interface between personal computers and robot controllers.

In general, robots' accessing methods differ from manufacturer to manufacturer. By standardizing this access, ORiN transfers data[2] stored in the controllers of various industrial robots onto personal computer. Once transferred, this data can be easily accessed via networks and shared amongst application monitoring robot operations, equipment diagnoses and even for production control.

ORiN is expected to improve the productivity of manufacturing facilities, these having only one standard to follow. More than this, it would also expand the scope of automation application through Know-how accumulation, every manufacturer being able to use concurrent concepts in their own creations. By standardizing the interface and data file specifications (through the use of standardized applications), this system also enables users (software houses) to exchange data with any robot conforming to the ORiN interface.

ORiN provides the following advantages:

- Uniform data exchange is possible between robots created by different manufacturers.
- ORiN being an Open specification, conform application can be developed by third parties.
- Ease of configuring multi-vendor systems.
- Worldwide standardization through proposal to ISO.

ORiN expected to bring about the following economic effects:

- Increased competitiveness in manufacturing.
- Expansion of the robot market.
- Entry of the software industry into the robot market.
- Creation of a robot engineering industry.

[1] See [JARA 1999], [Inukai 2003]
[2] For example: Robot's position information, number of parts to be assembled, number of defective parts, etc…

To achieve the above-mentioned objectives, it was decided to configure ORiN with provider, kernel and application logic layers.

The provider layer compensates for the differences in expression and/or protocol of robot controller data among various manufacturers and transfers data to the kernel layer, which is configured based on RAO (Robot Access Object) and RDF (Robot Definition Format).

RAO applies DCOM distributed object model technology to provide network transparency and uniform robot access, while RDF uses XML to provide files for defining structural models of robots with expandability. This enables ORiN to accept individual robot differences thus allowing it to be continuously used in the future.

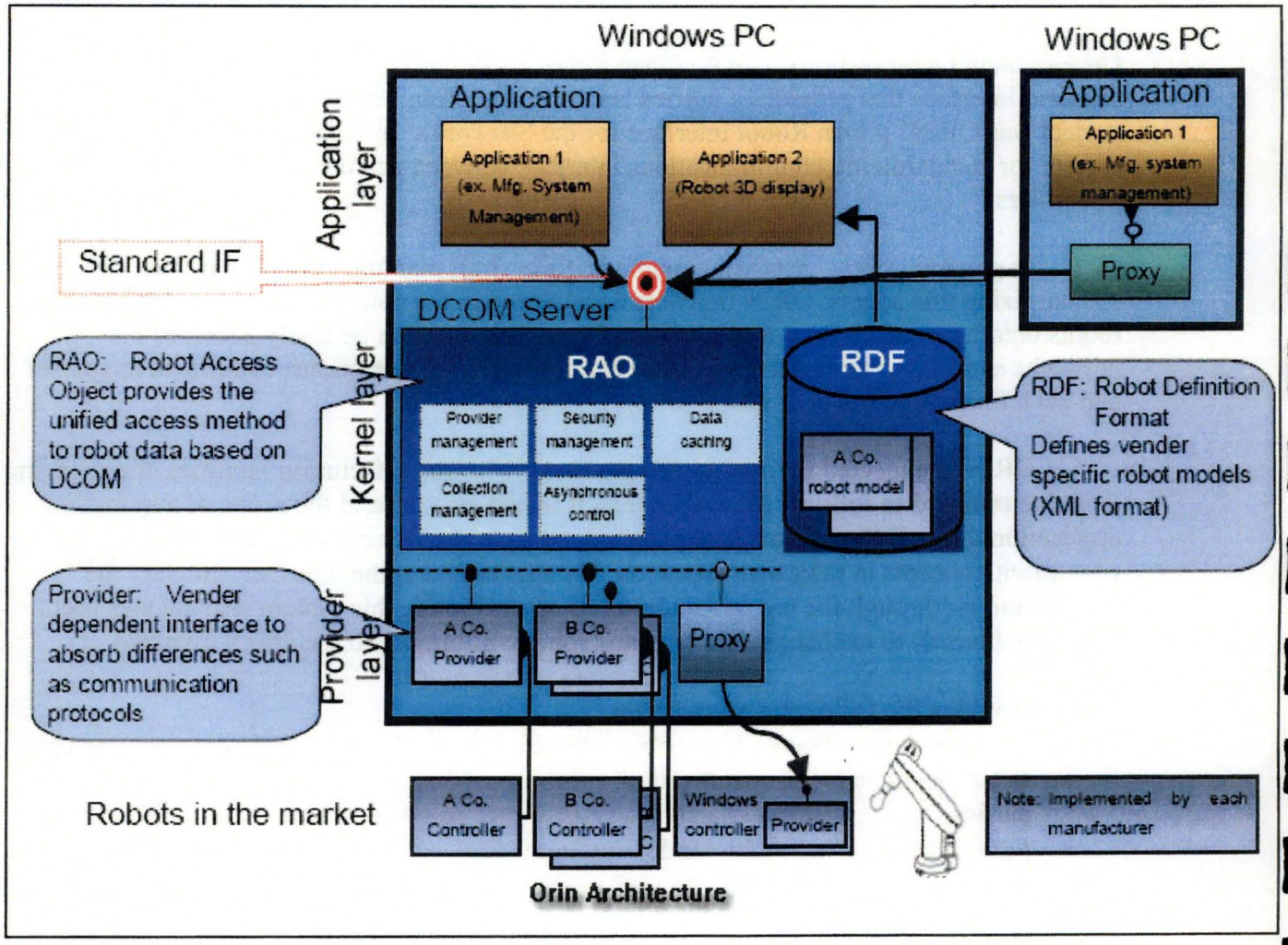


Figure 9: Scheme representing ORiN's implementation.

II. Open-r: An Open Architecture for Robot Entertainment

Sony Corporation has proposed an open architecture for autonomous robot systems, which aimed particularly, but not exclusively, entertainment applications. In order to achieve system extension and reconfiguration capabilities for mechanical, electrical, and software systems, they have proposed an architecture with the following features:

1. A common interface for various components such as sensors and actuators;
2. A mechanism for obtaining information on functions of components and their configurations;
3. A layered architecture for hardware adaptation, system services, and application providing efficient development of hardware and software components. A software platform provides an environment for agent design so that designers can customize their recognition and control algorithms. This is based on Apertos, a fully object-oriented real-time distributed operating system which allows each physical and software component to be defined uniformly as an object.

The outcome of this project is Open-R[1] and its goal is to establish a draft standard for mobile robots and their software systems. This standard would allow different companies and researchers interested in entertaining robots to build their own products and prototype systems using readily available components which meet Sony's specifications.

Their open architecture and standard target entertainment applications for three reasons:

1. **Complete Agent:** A robot for entertainment requires a complete autonomous physical agent. Instead of research and development activities focusing on specific perceptual functional components such as speech and visual cognitive subsystems, a complete agent promotes and accelerates research activities involving combination of subsystems and whole robot systems.
2. **Technology Level:** Robots for entertainment applications do not require such high performances in speech recognition and visual information processing that are required in mission-critical industrial applications. While there exist special and difficult requirements in entertainment applications themselves, limited capabilities or performances can cause a certain kind of excitement to users in most game playing situations such as RoboCup(soccer games by robot agents). This implies many existing AI technologies can be implemented for these kinds of applications.
3. **Emerging Industry:** Sony's researchers believe that they will be able to create a completely new market in the near future by introducing this kind of robot product sharply focused on entertainment applications. After the Gold Rush of Internet and cyberspace, people will eagerly seek real objects to play with and touch. Robot Entertainment provides tangible physical agents and an undoubted sense of reality.

By establishing a standard for entertainment robot software as well as robot parts, manufacturers can produce and sell their own commodities using the standard. AI researchers often spend large amounts of time customizing hardware. Readily available components allow researchers to construct customized robots for their research platform minimizing time consuming hardware and software hacking.

[1] See [Fujia & Kageyama 1997]

Below are Open-R system architecture's main features:

- **Open Architecture:** Open-R defines a set of standard interfaces for physical and software components and a programming framework, so that anyone can design extensions to the basic robot system within this standard.

- **Configurable Physical Components:** Open-R defines a common interface for all robot components for flexible and extensible robot configuration. This includes a mechanism for obtaining information on component function and configuration for interactive applications. Along with object-oriented software architecture, the Open-R provides Plug-and-Play capabilities for physical robots.

- **Object-Oriented Robot OS:** Open-R employs Apertos, a fully object-oriented distributed real-time operating system. This enables to define all physical and software components uniformly as distributed "objects".

In Robot Entertainment, there will be various applications, such as a pet-type robot, a game-type robot, or a tele-presence robot, which may be fully autonomous, or remote controlled semi-autonomous.

For these applications, the following are considered to be common requirements:

- stand-alone application
- extensibility
- friendly application development tools.

The AIBO entertainment robot uses OPEN-R as a standard interface. Facilitating modularised hardware and modularised software, this interface greatly expands the capabilities of AIBO entertainment robots.
The main advantage of working with Sony's AIBO is that it is an accomplished and stable development platform. In addition, it features state of the art hardware and a free and downloadable software-programming tool. This enables universities to fully gear resources and focus to programming in the area of Artificial Intelligence.

Sony Corporation has developed a prototype small biped entertainment robot "SDR-4X" that can adapt its performance to its environment and situations found in the home to further develop the possibility for a biped-walking robot.
The robot uses the same OPEN-R architecture as Sony's four-legged autonomous Entertainment Robot "AIBO". Two technologies applying the OPEN-R architecture, the "actuator" that moves the joints and "Whole Body Coordinated Dynamic Control" for real-time control of the joints realize the biped walking motion of the SDR-4X.

III. Orca: Open Robot Controller Architecture

To make robot technology become widely used, and to make various robots appear in the market, robotic parts - including mechanism, hardware, and software - should be produced as components with open interface. A lot of activities have been done on research and development for robot technologies in the world. However, the robot technologies so far cannot be reused because of incompatibility of the robotic parts. With the open interface, it becomes possible to use the robotic parts to build a wide variety of robot systems. Some people are confident that robotic technologies and know-how's can be accumulated for reuse with these reusable robotic parts.

To reach this hope, the Toshiba Corporation researches leaded to the creation of Open Robot Controller Architecture (ORCA[1]) which allows making the reusable robotic parts (software / hardware) to enable easy built-up of robot controllers. ORCA also allows manufacturers to quickly and easily integrate robotic parts developed by third parties into their systems, achieving efficient development of advanced robots in a relatively short period. Speech processing systems, image processing systems, robot control systems, and so on, are easily mixed up to build a robot system.

Toshiba's researchers have proposed Robot Technology (RT) reference model as an RT software layer structure. The structure consists of five layers:

- The physical layer;
- The I/O link layer;
- The actuator control layer;
- The motion control layer;
- The task layer.

ORCA has been defined according to the RT reference model. With the RT reference model, a developer can concentrate on a layer for which he develops software, because the software can utilize software for the lower layers with the open interface.

ORCA utilizes distributed object technology to abstract the communication between robots and between components. The distributed object technology enables developers to program a whole robot software system in object oriented manner. In ORCA, we can treat all robots as objects, and all the objects are defined with open interface. With the use of the distributed object technology HORB, the robot objects can be distributed anywhere in networks and they can be used directly from any node in the networks.

ORCA also consists of various interfaces and classes containing robot control software and acting as the ORCA's API specification. In order to use ORCA, developers have to implement the APIs defined in the interfaces. This ensures that any ORCA user will be able to use ORCA-based controller created by any other developer in the world.

[1] See [Ozaki 2003] and [Toshiba Corporation 2003]

Toshiba Corporation had developed a sophisticated home robot that could carry out multiple tasks around home. "AprilAlpha" integrates voice recognition and voice synthesis technologies that allow it to hold conversations with people, and image recognition technology that allows it to recognize people whose features are recorded to the robot's memory.

AprilAlpha integrates Open Robot Controller Architecture (ORCA), which allows simple additions of new functions and upgrades of present functions.

PART 2
Our contribution to FDNet

Chapter 3: Retro engineering

I. Motivations:

The work we had to do had strong relations with the work already done by FDNet programmers. In fact, we had to create programs that would be used between Robot's intelligence (FDNet) and the users, allowing them to unleash the full power of this network architecture without having to deal with its complexity.

In an environment like this one, it is clear that we had to understand FDNet's logic, structure and architecture completely before even thinking about our own work.

A common way to achieve this work (understanding how FDNet works) would be to have a general explanation about what the application does, to see it run (if possible), to read its specification and its source code and to discuss all along with the team, to understand their way of thinking and their point of view about the way they want the application to be done.

But we had to face here with a major problem: the language spoken and chosen to write the documentation, the Japanese language.

First, the programmers couldn't speak English enough to allow us to discuss with them about the project. They could read English but no spoken interaction was really possible. In such an environment, reading the most possible documentation, specification and notes and trying to understand the most part by ourselves is far more preferable.

But we had to face the problem that no documentation was available for us. In fact, there was very little general explanation about the project and no specification at all. The only thing that seemed to be present in this field was code documentation…which was written in Japanese. As the programmers had no time to translate the comments and/or to explain us how the whole application worked, we had to read the source code, without any comment at all, and to understand it the most possible.

Doing this kind of work is a very difficult task. To maximize our comprehension capacity, we decided of a structure allowing us to write down every thing we understood and, by advancing in our understanding, to recreate our own documentation.

II. Analysis method:

This is the structure we decided to use for documenting all the source code we read. Note that the code was also a work in progress and was still being heavily modified when we started this "Retro-Engineering" process.

Class Name	The name of the class described. This name is Case Sensitive If the class is abstract, its name will be written in **blue.** If the class is an interface, its name will be written in **orange.**
Extends	The name of the class that this class extends; null if no extension.
Implements	The name(s) of the interface(s) this class implements; null if no implementation is made
Aim of the class	The main aim of the class. This is a general explanation of what the class has been created for. This explanation must help any programmer to understand the structure of the code inside the class better and to give him an idea of the relations that this class has with the other ones (if any).
Comments	Any specific comment that doesn't fall In the "aim" group here above. Questions are written here in red color.

Property name	Name of the property. Case sensitive
Property use	The reason why this property has been created.
Comments	Any other comment concerning this property Questions are written here in red color.

Method name	Name of the method. Case sensitive
Method use	The reason why this method has been created for. The explanation must be clear and general. It doesn't explain the inside of the method, only what it does.
Comments	Any comment, technical or not, fall here. Questions are written here in red color.

The aim was to make the documentation in several loops, each time answering the questions written in red and writing new questions down.

At the same time as this documentation's creation, we created schemes of the dynamic interactions between the classes, schemes of the database tables, general schemes of way FDNet worked logically and so on. These schemes allowed us to have a better overview about the work done by FDNet.

We decided to limit our work to FDNet's core packages. Theses packages contained all the base information concerning network entities, the relations between them and so on. This limitation was set because we had little time for us to produce quite an amount of work too.

III. Conclusion:

To use a retro engineering process in a case like the one we faced was really a difficult job. It took us more than one month (with three people) to read the core packages and to have a first draft of FDNet's implementation. To produce the schemes required us to read the code more than one time and we must admit that we still have some questions which haven't found any answer.

Nevertheless, the retro engineering system we decided to use (for which an example is given in annexes 1 and 2) appeared to be a good choice. It allowed us to understand the most of FDNet, which is already interesting, but, more than this, it constituted our documentation for the rest of the training session. Without this retro engineering system, it is clear that achieving the work we were asked to do at the very beginning of the training session wouldn't have been possible.

Chapter 4: The Logger

I. Introduction:

A. General purpose

We have created a system which acts as a buffer between the FDNet core and the database because the database and the core don't work at the same speed. Indeed, the core creates so much information that it cannot be saved in real-time. A buffer system has then been implemented; its role being to save all data to be put in the database in memory (thus being able to keep up with the core's speed) and to save it in the database at a later time (thus working at database's speed).

This system can also share information about the network states because we wish to analyze the states of the FDNet network in real time (through the FDNet viewers).

To attain these objectives, we have created a network state logger. This logger is able to store the network states and share them rapidly with the FDNet viewers (which have to work in a constant delayed time).

B. General view

To understand the function of the logger, we must have a general view of the FDNet project. Below, we have a scheme that sets the place of the logger in this project and the interaction with the other elements that already exist.

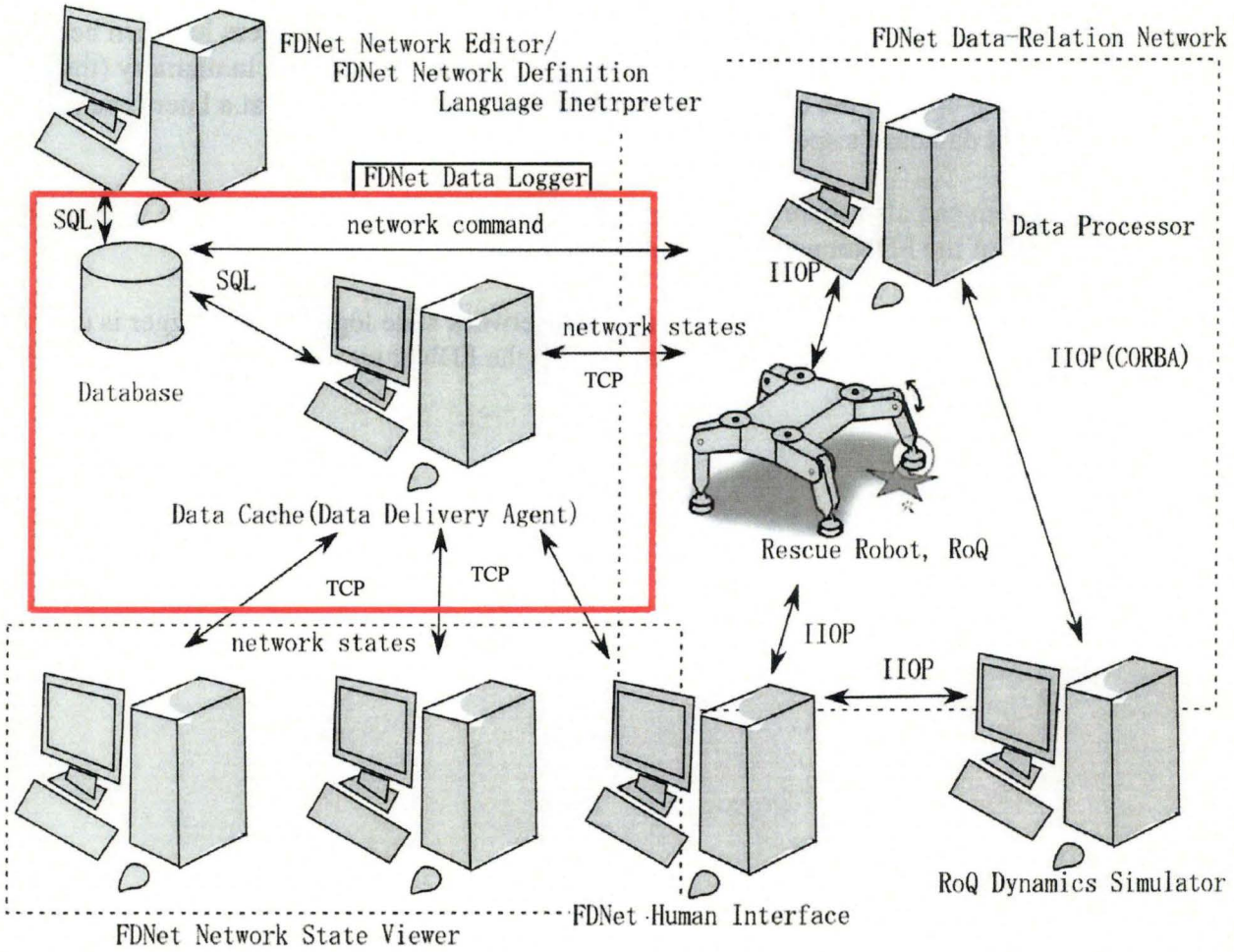


Figure 10: General view of FDNet that show the relations between the different applications of FDNet

II. Specification:

A. Logical specification

This part of the document was used to agree, with all people that work in the FDNet project, on what the logger must do and how the logger interacts with the other components of the project.

We define all elements that must intervene in the creation of the logger and the FDNet cache.

The Logger:

Considering that the database server is slow compared to the processing speed of the logger, the logger must have a cache. The events sent by the core are stored in this cache. A delayed writing must also be implemented between the logger and the database.

This cache must also allow the viewer to monitor some events created by a specific object. As a reminder, the logger must work in real time with the core and the viewer. So the access speed to this cache is very important and has to be very fast. We must also manage the multi access and take care of the integrity of the values contained in this cache.

The cache must also make a difference between the events monitored[1] and not monitored. Only the events that are not monitored can be deleted from the cache and saved into the database.

The Core:

The FDNet core is the heart of the project. It represents the low level control on the rescue robot and also contains all basic functionalities needed to create, configure and update the network. It also sends some information about the state of the network.

A cache must be implemented in the FDNet core. The utility of this cache is to receive some events from a node and store it. After that, the cache must send theses events to the logger.

This cache uses the same logic than the cache used by the logger. But the core doesn't need to monitor events contained in its cache. So we must define a general concept for the cache that can be used for the logger and the core.

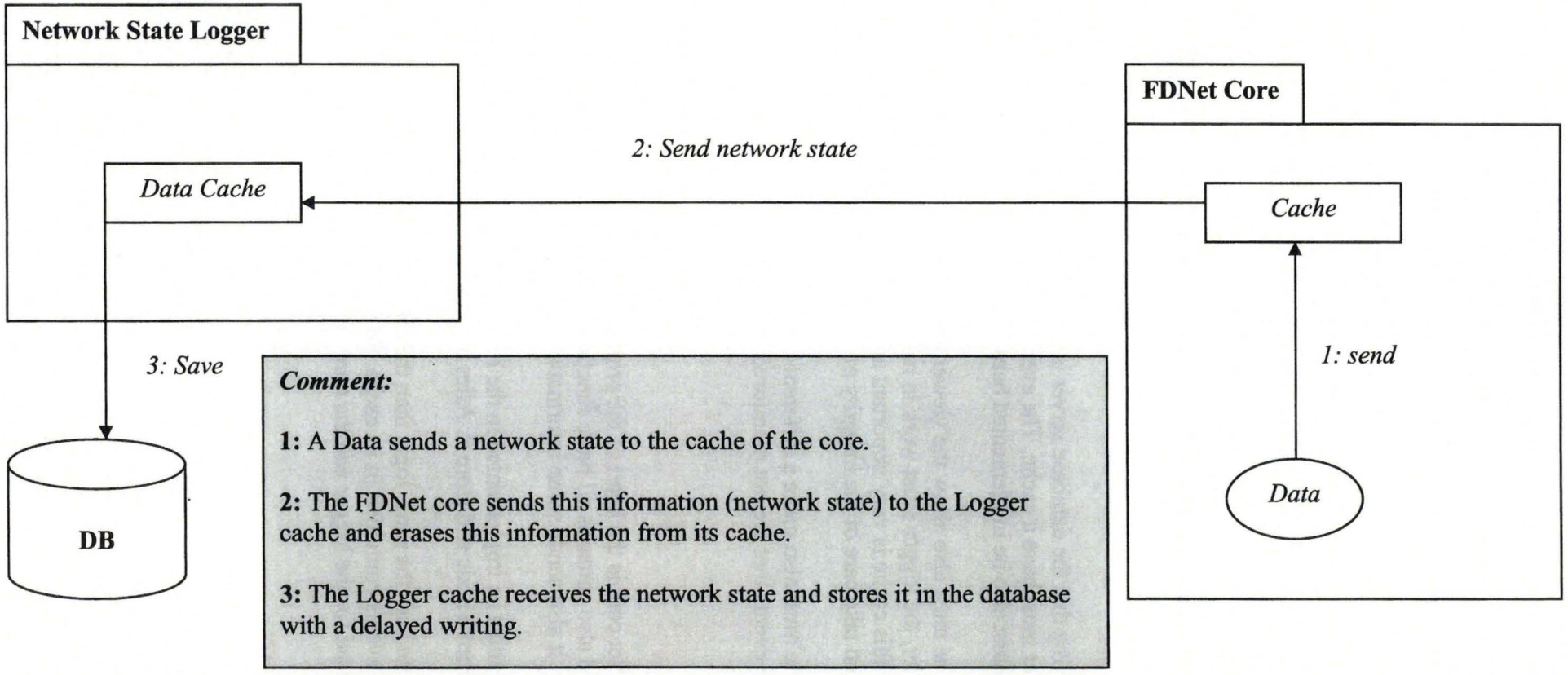
The viewer:

The logger has network states that must be accessible by the FDNet Network State Viewer. We want to analyze the network by using the interface and the viewer. To analyze the network, we need the network state information contained in the logger cache.
So, to allow the viewer to access to specific network state information, we must implement a server on the logger for the viewer.

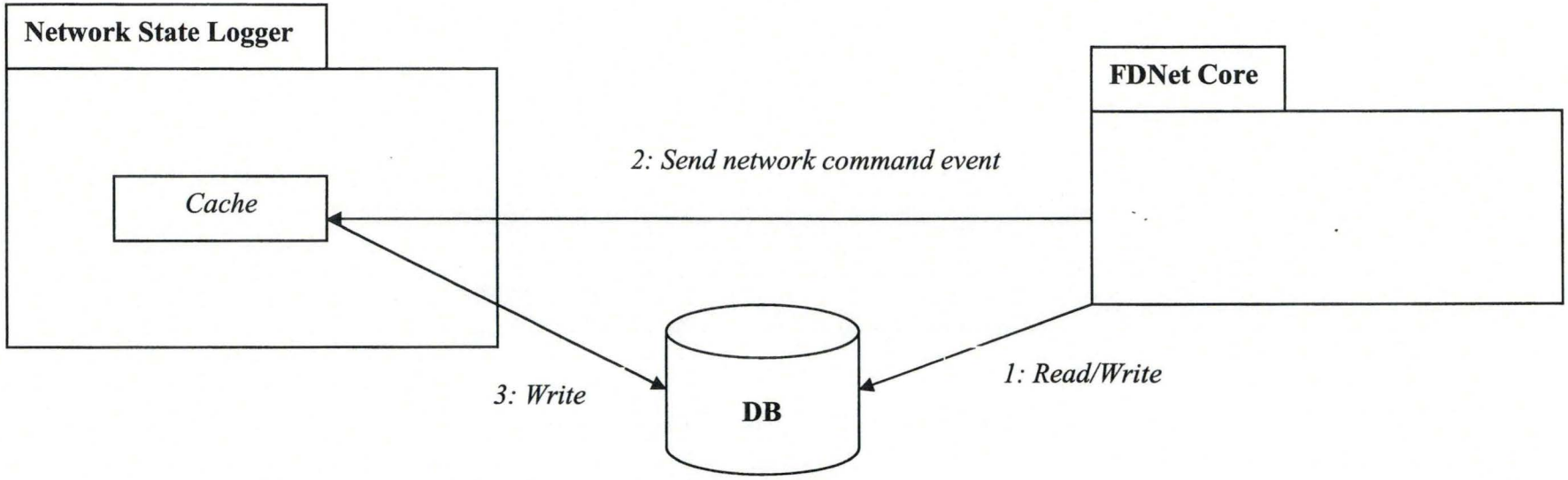
[1] An event is monitored by the viewer to analyze the network state

B. Sequence diagrams

Scheme 1: The FDNet core sends a network state, here value of a data, to the Logger.



Scheme 2: The FDNet core reads/writes connection information and sends network command event to the logger.



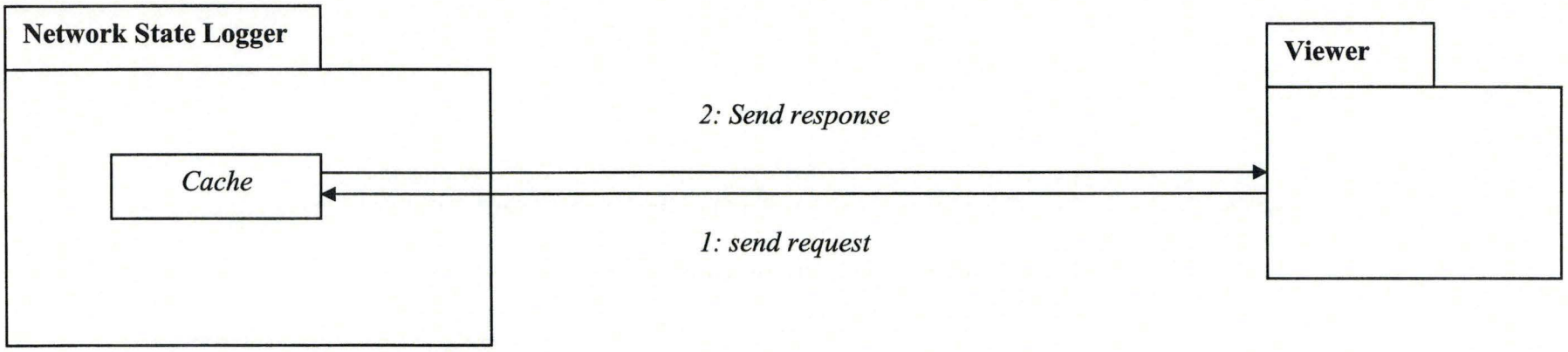
Comment:

1: The FDNet core reads/writes current connection information directly in the database.

2: The FDNet core sends a network command event because we want to analyze the network state by using the Logger information.

3: The logger writes this event in the database.

Scheme 3: A viewer wants network information.



Comment:

1: A viewer tells to the logger that it wants to receive the state of a certain data.

2: The logger receives this information and sends all events requested by this viewer.

C. Physical specification

Data Base:

We use the relational database PostgresSQL 7.0.

Programming Language:

We use the object oriented language java, because we want to use an object oriented method to program the logger and it must work on different operating system.

Operating system:

Using the Java language, the logger works on the Windows and Linux operating systems.

Communication:

- *Logger - Viewer:* We use a TCP connection and a streaming service
- *Logger - FDNet core:* We use a TCP connection and a streaming service

D. Improvements

In this part, I will present you some ideas to improve the logger functionalities.
Creating an interface for the logger containing information described here below would be interesting. Note that this interface has been accepted and done. You can find more details about it at the end of the logger part.

Logger's parameters:

These are the parameters used by the Logger at work time:

- Logger's address: The IP address used by the FDNet core or Viewer to connect to the logger
- TCP port used by the FDNet core
- TCP port used by the viewer
- Cache's size: The memory size available for the Logger cache during the execution.

This information could be changed while the logger is not working. That way, it is easier for a person to change the Logger properties.

Logger's real-time information:

This information shows how the logger is actually working:

- Size of the cache used
- Size of the cache free
- Information contained by the logger cache: I don't know if it is very useful to see this information. We can for example search all information about a certain data and display it. This part is going to be more specified later.
- Number of viewer and core connected to the logger with their IP Address.

With this interface you can also start and stop the logger.

III. Database of the Network State Logger:

A. Preface

In this document, we specify the database tables necessary for the network state logger.
To save data and relation values, we have tables '*datavalue_log*' and '*procvalue_log*'. And to save network command events, we use the table '*netcommand_log*'.

B. Specification of the tables

Table:
datavalue_log

Role of the table:

This table stores all values of a data node during its life.

Scheme:

datavalue_log
sourceid: integer time: bigint value: text typeValue : Byte
Pk: sourceid,time

Definitions of the attributes:

- *sourceid:* The FDNet identification (ID) of the data
- *time:* The time when the value was assigned to the data
- *value:* The value of the data at this moment.
- *typeValue :* The type of the value (Boolean, long,...)

Table:

procvalue_log

Role of the table:

This table stores all values of a relation node during its life.

Scheme:

procvalue_log
sourceid: integer time: bigint value: text
Pk: sourceid,time

Definitions of the attributes:

- *sourceid:* The FDNet identification (ID) of the data
- *time:* The time at which the value was assigned to the data
- *value:* The value of the data at the present moment.

Table:

netcommand_log

Role of the table:

This table stores all network event commands.

Scheme:

netcommand_log
id_db: integer sourceid: integer sourcetype: text command: text params_id: text params_type: text time: bigint
Pk: id_db

Definitions of the attributes:

- id_db: The database identification of the tuple. It is set automatically by the database when a tuple is inserted in this table.
- sourceid: The identification of the source that creates this command
- sourcetype: The source type
- command: The command type of the event. There are four commands (connect, disconnect, create and destroy).
- params_id: The FDNet identification (id) of the command parameter.
- params_type: The Parameter type of the command. We have four type (reader, write, data and proc)
- time: The time when the value was assigned to the data

C. Remark:

We have split the table "*logValue*" (defined by the student Mr Pujol) into two tables, "*datavalue_log*" and "*procvalue_log*" because we want to improve the access speeds to the database.

With two tables, we also use less memory space than with one table, because the type of the value is not necessary to know if the value is from a data or a relation.

IV. The Cache:

A. Preface

For the logger and the FDNet core, we use a cache to stock events that represent some changes in the network. We have two kinds of events, network state events and network command events. We have defined a common cache architecture. This architecture is designed to be generic and is very fast.

B. Cache architecture

General architecture

In this cache we have two event containers. One is for the events that represent the network state. The other is for the events that contain the network command.
We must also monitor some events, in particular network state events. To do that, we have created a special event container that extends the normal event container and adds the monitoring of the event.

So in the cache, we have an event container for the network command events and an event container with monitoring functions for the network state events. In the future, if we also want to monitor the network command events, we just have to change its event container. It is also easy to add another type of event in this cache. This makes the cache architecture very flexible and generic.

The Event Container architecture

An event Container stores all events. Each event has the identifier number of the data or relation that creates it. A same Node (Data or relation) can create a lot of events at different times. An event is accessed by using its identifier and also, occasionally, the time when the event was created. Then to improve the access to a specific event, we must have an index on the identifier and the time of the event.

This index is implemented by using an array that contains the identifier of all nodes. Each entry of this array contains all events that the node has created and these events are sorted by time in the array. This is an easy way to get all events (sorted by time) that a specific node has created.

The Event Container Monitor architecture

An event Container with monitoring functions is the same as a normal event container, but with some improvement to implement the monitoring of the events.
A monitor can be set on a node. When a monitor is set on a node, all events that this node has created must be available for the entities that need them. When an event has been sent to all supervisors, this event is no more interesting for the supervisors and becomes unavailable for them until a new supervisor is added for the node that has created this event.

To improve the access and the processing, we have duplicated the index of the event container.

There is an index for all events monitored and an index for the others.

The added index has the same structure than the other index. In other words, this new index contains an array with the identifier of all nodes, and for each entry of this array we have another array with all events created by this node and sorted by time.

C. Scheme of the cache

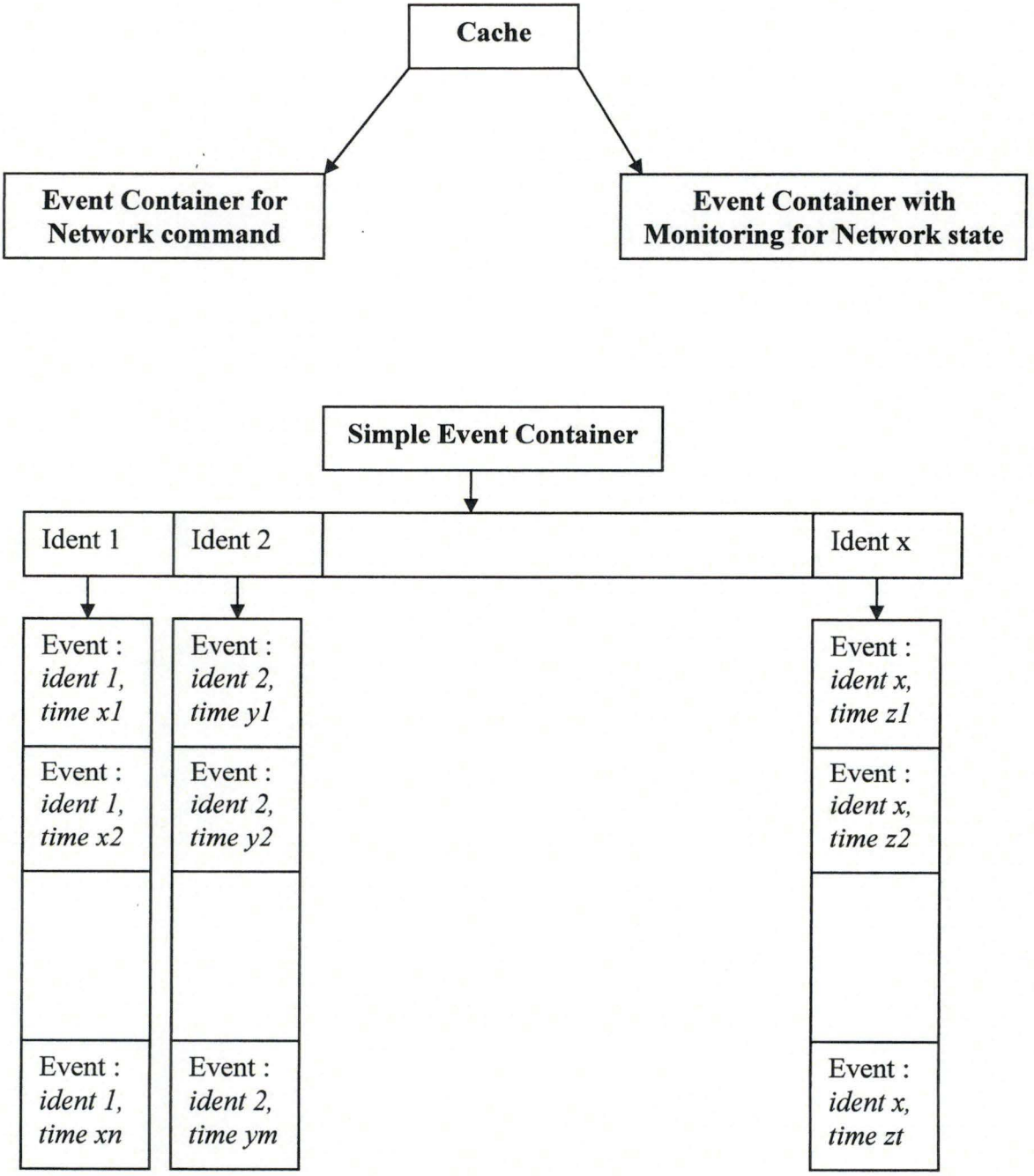


Figure 11: Scheme and structure of the cache and the simple event container

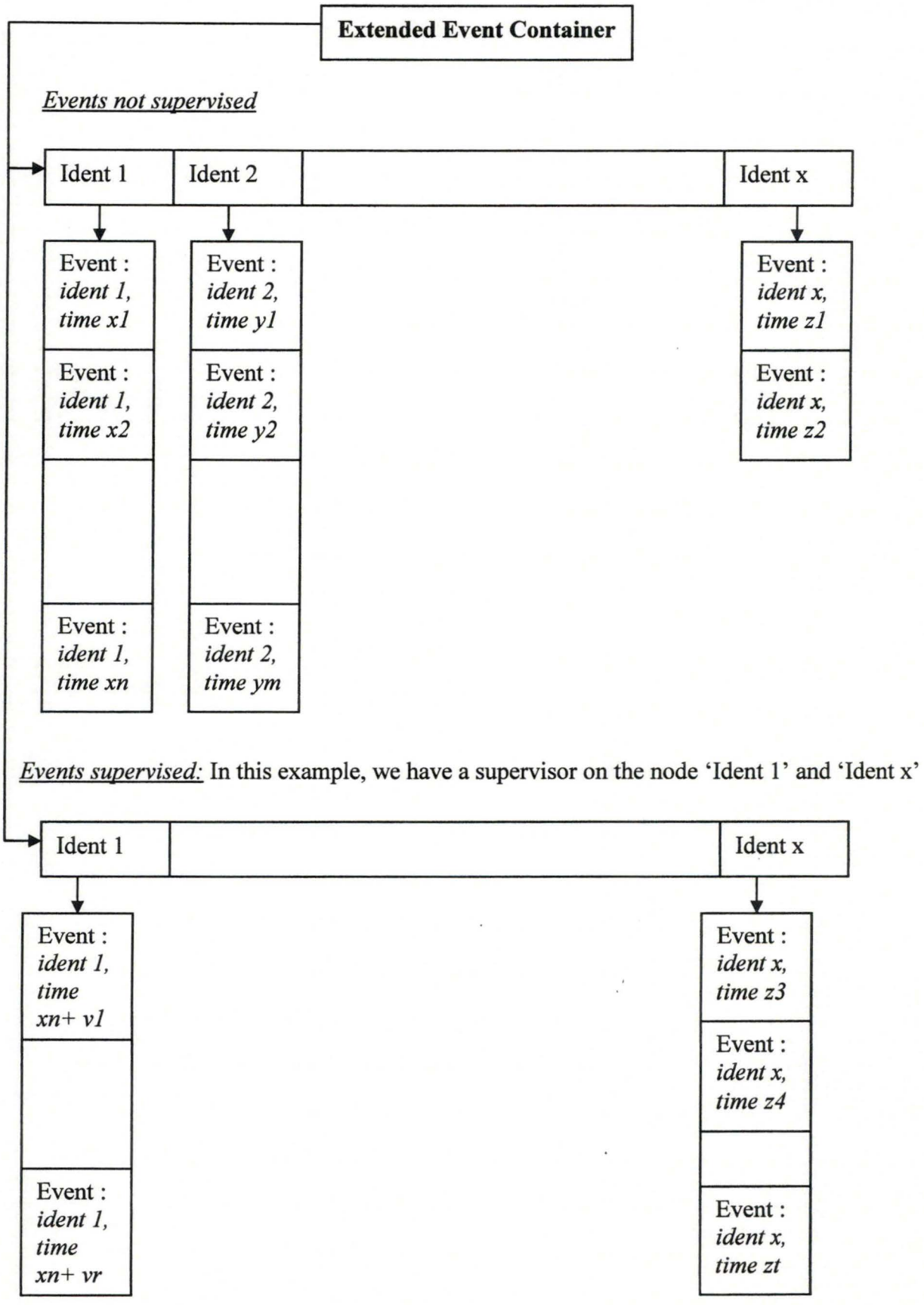


Figure 12: Scheme and Structure of the extended event container

V. The Server

A. Preface

To add and read information contained in the logger, we have implemented two TCP servers. One of these servers must receive events from the FDNet core and store them in the logger's cache. The other server must give all events requested by viewers. To respond to these requirements, we have defined a TCP server in a general way.

B. Server architecture

General architecture

The server must be able to receive many clients. Then all processing necessary to respond to a client must be done in another thread.

To respond to this requirement, we use an architecture with three layers for the server: server, factory and connection.
The server creates a factory and the role of this factory is to receive all client connections from the server. When a new client is received by the factory, it creates and delegates a specific thread, a connection, to respond to this new client.
So for each client received by the factory, a new thread is created.

Advantage of the architecture

This architecture has many advantages. With this architecture, the server doesn't respond itself to the client request. Its mission is only to get the new client and send it to its factory. So the server is available very quickly and often. The server can thus receive and serve a lot of clients simultaneously.

another advantage of this architecture is that the factory creates and manages all connections with the clients . It is possible to limit the number of clients served by the server in the same time. You can also check the state of all client connections (activated, ended,...). It is very useful with an interface where you can follow the connections established with the logger.

C. Scheme of the architecture

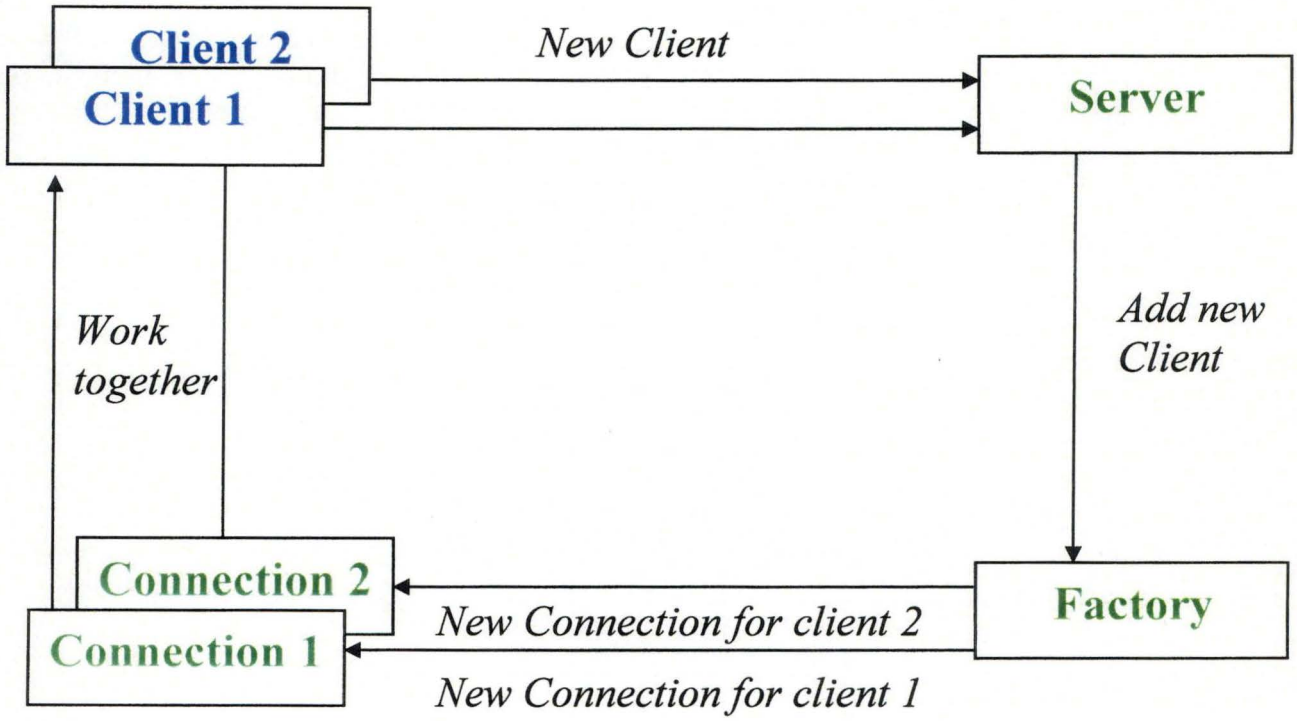


Figure 13: Architecture of the server in three layers used for the Logger

D. Remark

We saw that two servers were used for the logger. Each server uses the architecture described above. Thanks to the object oriented language, we use many common components for the two servers. In fact, only the connection class is different for the servers. We use two different child classes of the connection class.

When the factory creates the connection object for a client, the factory gets the constructor of one of the children of the connection class, and creates an instance of this child. Like this, we have created a very generic server and factory. To use this class and architecture for another server, you only need to create a child of the connection class for your server. After that, you just have to call the constructor of this class to the factory and your server is ready to work.

VI. The Logger:

A. Role of the Logger

The logger has three principal roles in the FDNet project.

The logger must act like a buffer between the FDNet core and the database. Because of the speed difference between the core and the database, we need something that receives all events sent by the core, stores them in a cache and writes them in the database.

The logger must be able to receive all requests from viewers and respond to them by sending all events they ask for. To achieve that, the viewers also need to get the events stored in the logger cache and in the database.

To summarize, the logger must receive all events from the FDNet core and store them in the cache. It must be able to write all events from the cache into the database. Finally, it must also be able to send all events to the viewers to respond to their requests.

B. General architecture

Like explained before, the logger has three main missions. The architecture of the logger follows the same logic, because the components are designed by objective. Here is a presentation of every part of this architecture.

Server Receiver

This part of the logger must receive events from the FDNet core and store them in the Logger cache. This server uses the same methods as the TCP server described in the beginning of this document. The cache also has the same functions as the cache defined before.
This part has been improved to empty the FDNet core cache in a very fast and efficient way to avoid filling up the core's memory completely.

Logger writer

The mission of this logger's part is to read all events from the logger cache and to write them into the database. This part is very important because if the logger takes too much time to achieve these tasks, the cache may fill up more rapidly than it empties. The consequence of this problem is that the memory could be too shallow to stock all events. Consequently, this part has been improved to be very fast and to keep the integrity of the events in the same time.

Server Sender

The third and last part of the logger is the server that sends all events requested by the viewer. This part must check the database and the cache to make sure the events requested are enabled for the viewers.

C. Scheme of the architecture

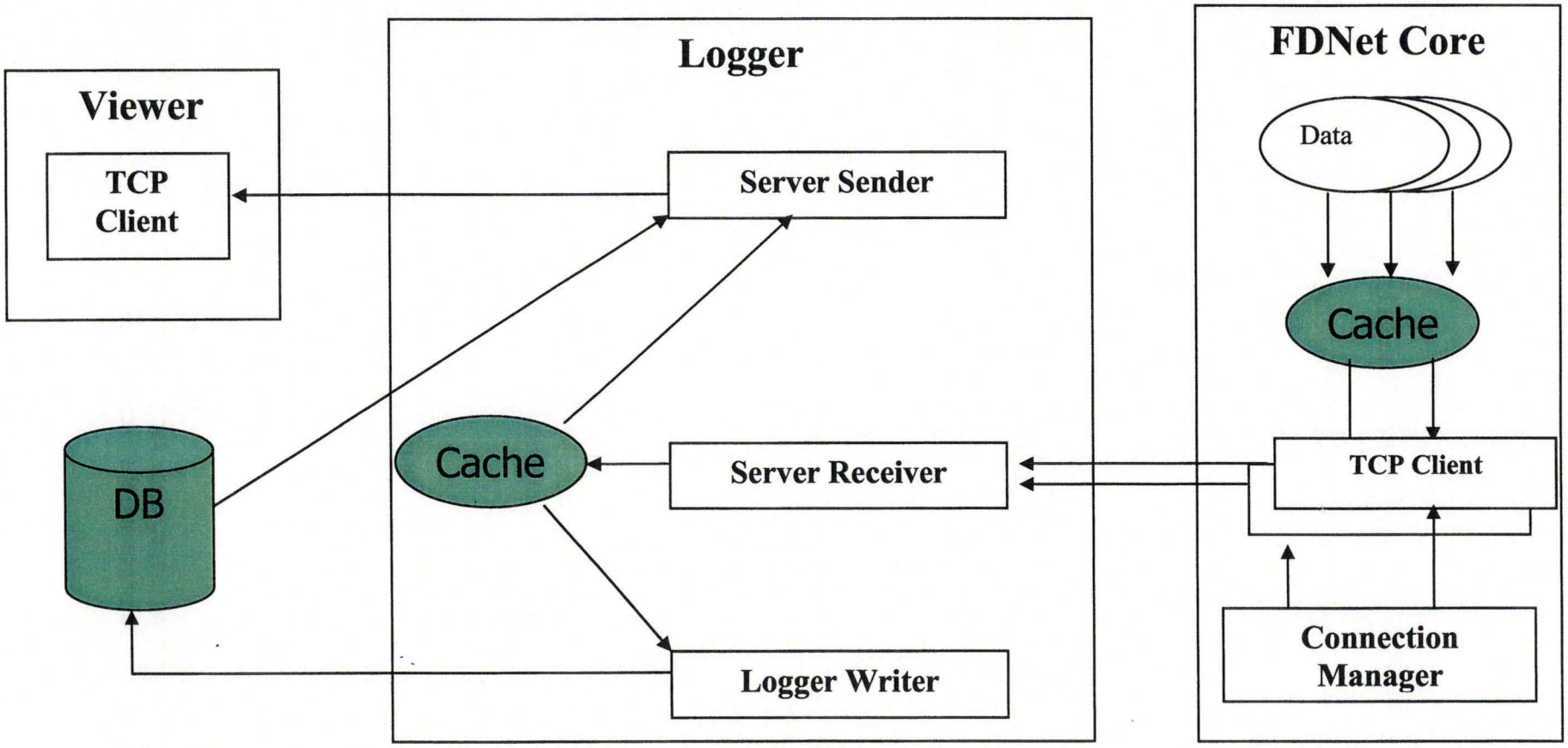


Figure 14: General architecture of the Logger

VII. Improvements

A. Introduction

The logger must be very fast to succeed in its three missions. It must receive and fill its cache very quickly and efficiently to avoid using up all memory of the core cache. But it must also empty its cache rapidly and write all events in the database. Finally, it must also reply to the viewers' requests by sending all events they ask for. The logger uses only one cache and each one of the three logger's parts use this cache to add, get or delete its content. So we look after the integrity and value of the events.

Another difficulty in the logger's improvement is that some improvements can affect others. For example, if you want to respond rapidly to a viewer request, you must avoid accessing the database because of its slowness. Consequently, you must keep a lot of events in the cache. But if you want to keep more events in the cache, you can't empty it and send its events into the database. A lot of other problem like this exists and may arise when trying to improve a mission of the logger.

In this part, we present the most important improvements that we have applied to the logger.

B. Database access

The access to the database is very slow and the logger must empty the cache in a very fast way. In fact, a lot of events come from the core and are stored into the logger's cache. The exchange of information between the logger and the core must be done in real time. So this cache fills up very fast and is emptied very slowly because of the database access. The cache has a size limit and an 'out of memory' exception appears when we overstep this limit. This exception is a way to crash the logger and to lose many events.

We must improve the access speed to the database. Write one event per access is not a good idea. We must use a buffer between the logger and the database. This buffer must contain all events that we want to write into the database. This buffer is filled with events and when it becomes full, we write the content of the buffer into the database in one single access.

With this technique, there is a need for a timer that starts in the beginning. If the buffer fills up before the end of the timer, it is stopped and the content of the buffer is written. After that, the timer is restarted. If the timer expires and the buffer is not empty, the content of this buffer is written into the database. The timer is used because we don't want to keep an event longer than 'x' seconds into the buffer ('x' represents a value in seconds >0).

We must also pay attention to the consistency and the integrity of the data. The cache of the logger is accessed to write and read events. The logger must share some events with the viewer. To present the events to the viewer that requested them, the logger must check the database and its cache to find the events. To keep a good coherence, the buffer must be emptied (as a reminder, the buffer contains the events to save into the database) and its content must be written into the database before reading the content of the database.

To understand how this technique works, let's analyze the following schemes:

- The buffer becomes full before the end of the timer

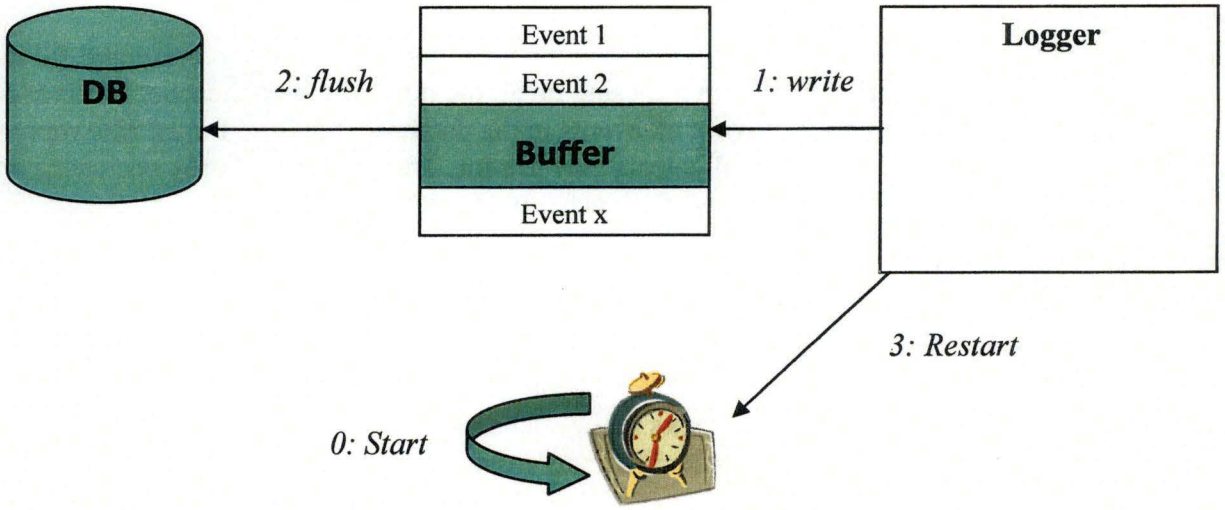


Figure 15: Database access improvement – Scenario 1

- The timer expires and the buffer isn't full

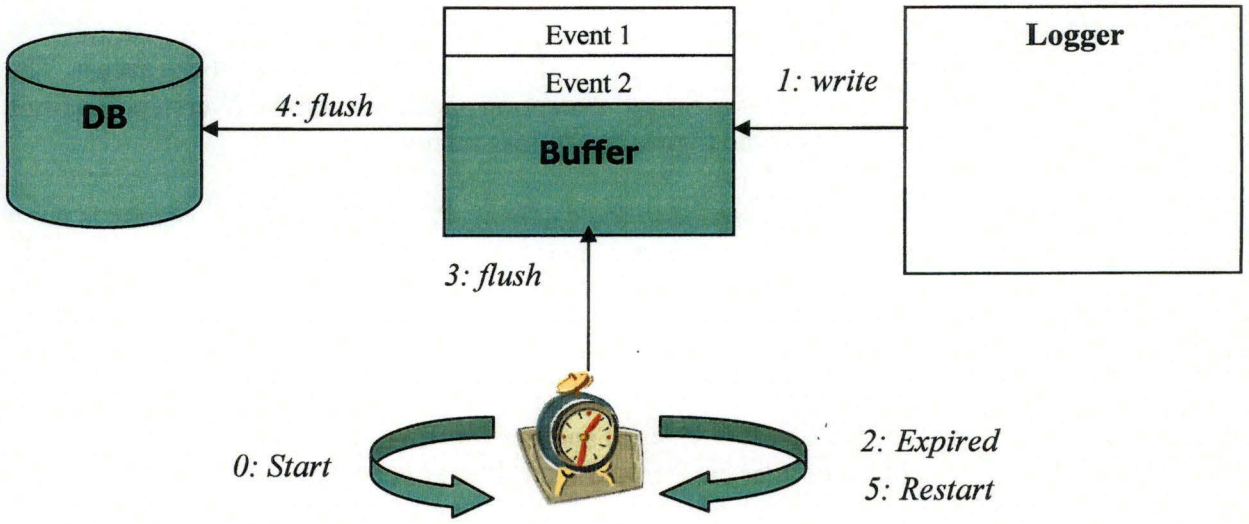


Figure 16: Database access improvement – Scenario 2

C. Thread priority

Different improvements can be done in this domain, but as we have seen, an improvement can influence others. For example, if you want to increase the efficiency of the information exchanged between the logger and the viewer, you must keep many events in the logger's cache. To do so, you must decrease the amount of exchanges between the logger and the database, and postpone the writing of the events. You must also be careful for the size of the cache, because it fills up very quickly. The operations to fill the cache (receive events from the core and write them into the cache) are happening faster than the operations to empty it (read events from the cache and save them into the database). Improvements are definitely not easy to do. You must pay attention to the way you implement the improvements. It is easier to choose a static way to implement the improvements, but it is not very efficient and viable.

The best way to solve this kind of problem is to choose a solution that checks the most of the requirements and this solution must also create some dynamic priority for all the improvements. For example, you can empty the cache slowly, but when this cache become nearly full, you must revalue the priority of the improvements.

This solution with dynamic priority has been chosen in the construction of the logger. It is implemented by using the thread priority. Let's see the functioning of the thread priority. The logger is composed of three main threads: two Servers and a thread that writes all events from the cache into the database (we call this thread the writer). In the beginning, all threads have the same priority. Because of the slowness of the database access and the amount of events exchanged between the logger and the FDNet core (as a reminder, the logger and the FDNet core work in real time), the cache size increases rapidly. When the cache size reaches a limit (a critical size defined by the user) the priority of the writer thread is increased (normal priority to high priority) to empty the cache faster. When the cache reaches a normal size defined by the user, we decrease the priority of the writer thread (high priority to normal priority).

To implement this technique and to check the current available size of the cache, we need to know the total cache size which is given by the user and expressed in Byte, Megabyte or Gigabyte. We also need to define the size of each event stored by the cache and the number of events stored in the cache at a specific moment. The critical and normal thresholds, represented by a percentage of the cache size, must also be defined by the user. To calculate the size of an event, represented here by a java object, we use the 'Serializable' property of java.
First, we calculate the maximum number of events that the cache can store. Once this has been done, we define the critical number of events that the cache can store by using the critical percentage and the maximum number of events. We do the same operation to obtain the normal number by using the normal percentage instead of the critical percentage. To know if the priority of the writer thread has to be changed, we compare the critical and normal number with the number of events stored in the cache at this particular moment.

Let's see the following scheme to understand how the thread priority works:

- In the beginning, the cache is empty and all threads of the logger have the same priority

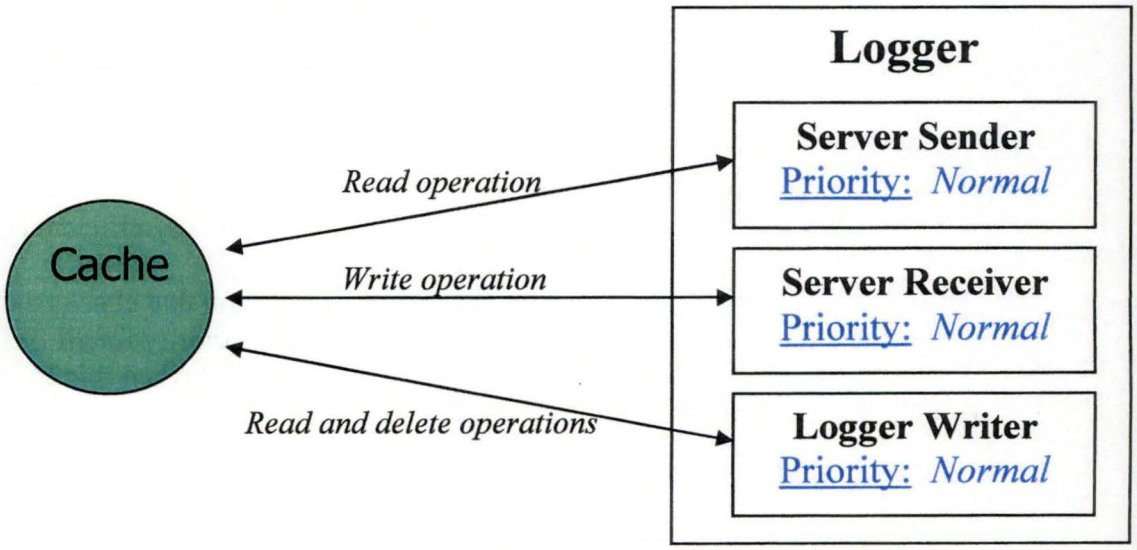


Figure 17: Thread priority improvement – The states of threads and cache in the beginning

- The cache reaches the critical level and the priority of the writer thread is changed

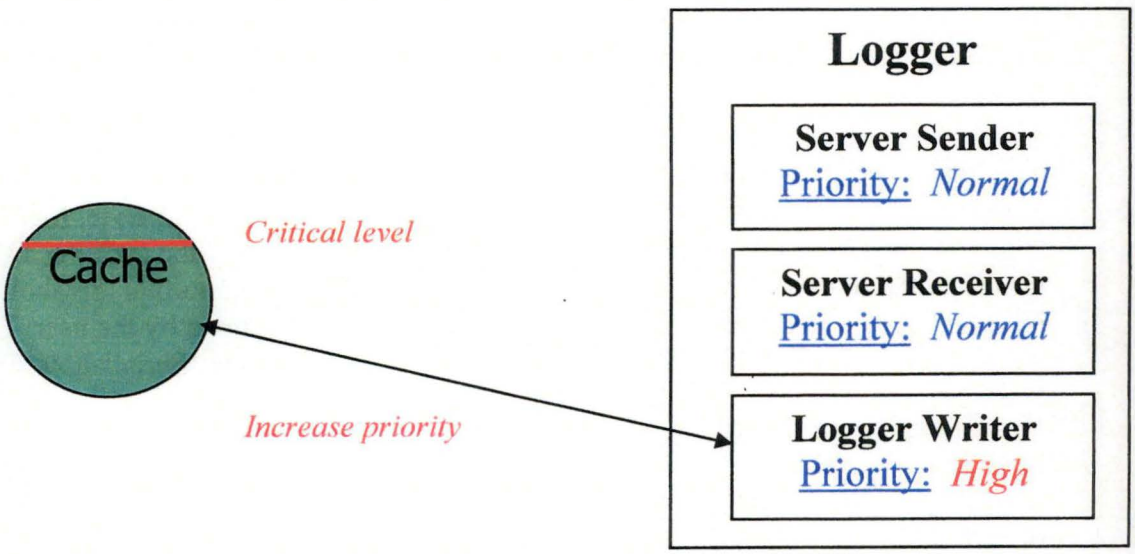


Figure 18: Thread priority improvement – The cache reaches the critical level

- The cache reaches the normal level and the priority of the writer thread is changed

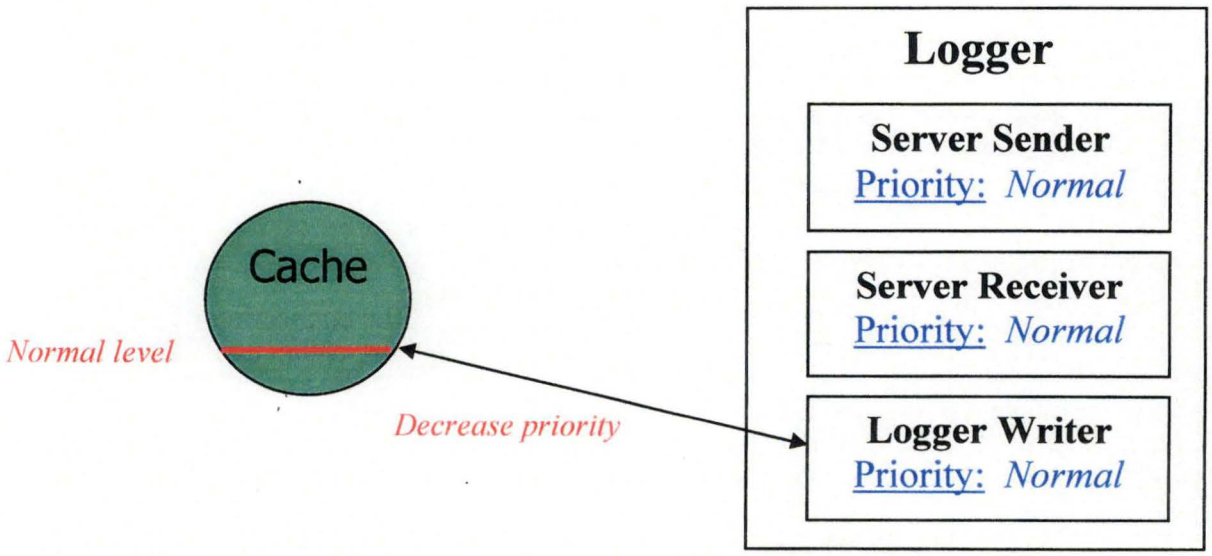


Figure 19: Thread priority improvement – The cache reaches the normal level

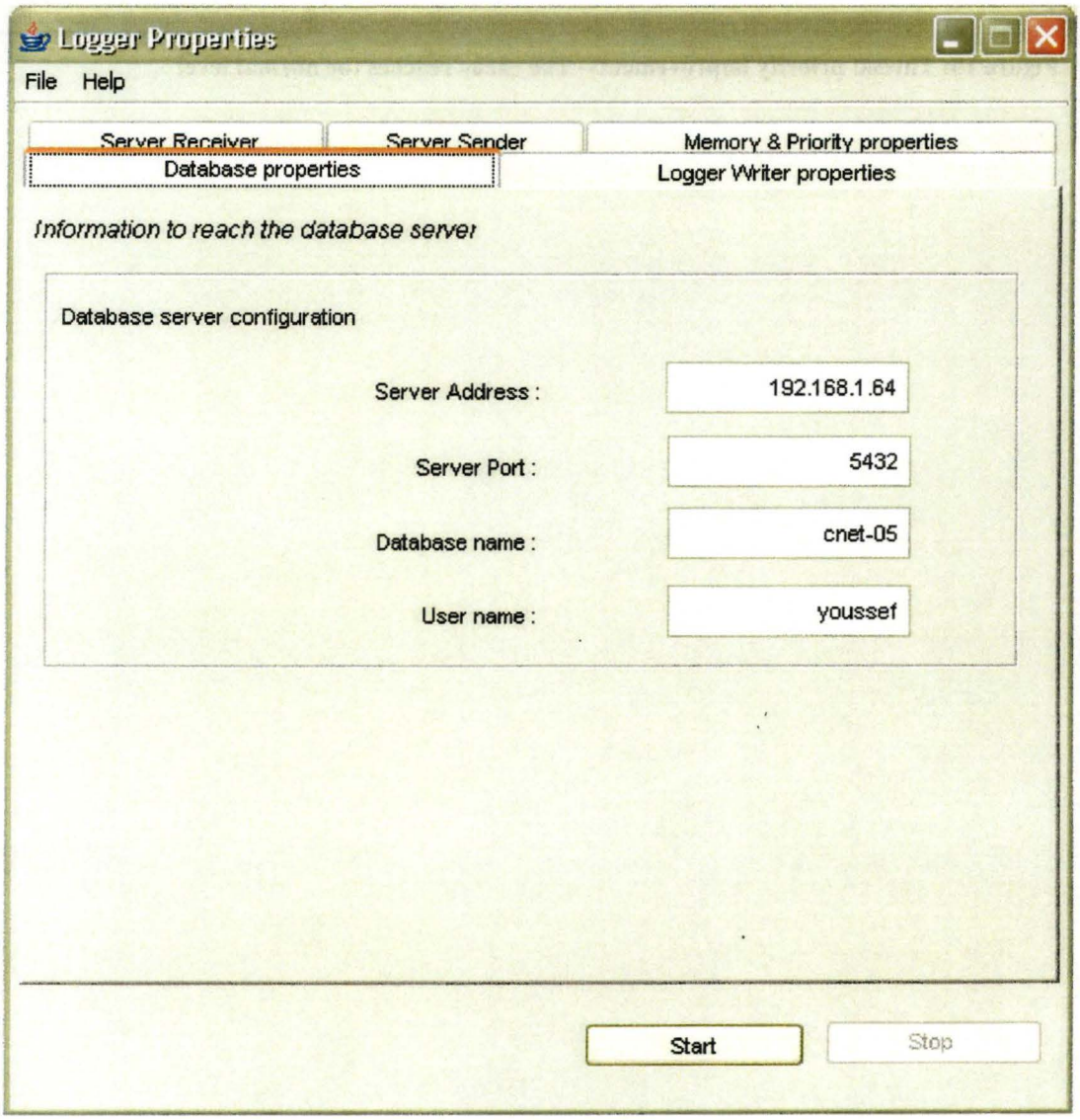
VIII. The interface

A. Preview

All information that you enter in the logger interface is saved in a file that is read when you launch the interface. If this file doesn't exist, the fields of the interface are set to default values.

B. Database information

This part of the interface is used to set the properties of the connection with the database server. The logger must save information into the database, which can be on another computer. To improve the utility of the logger, we have created a panel in the interface to allow the user to enter the IP address of the computer where the database server is running, the port of this server and the name of the database.



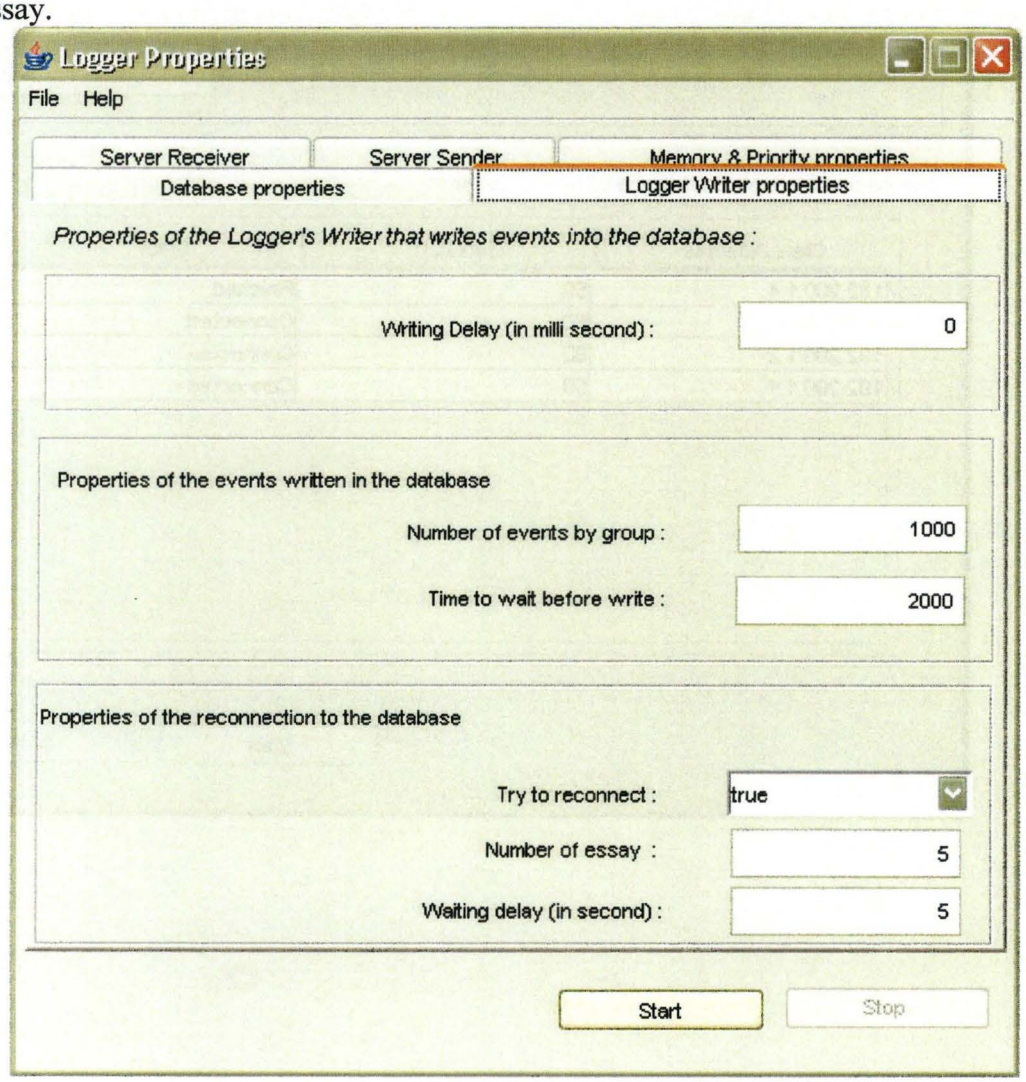
C. Logger writer

This panel is used to set the properties of the thread that reads events from the logger's cache and that writes them into the database (the writer thread). The panel is divided in three sub-panels; each sub-panel represents a different part of the writer thread.

The first sub-panel is used to enter the delay writing in milliseconds used by the logger to write into a buffer used between the logger and the database. So if you want, you can delay the writing of the events. It allows keeping more events into the cache and it also allows sharing the events with the viewer faster.

The second sub-panel is used to improve the access to the database. In fact, it is possible to send a packet of events simultaneously instead of one by one. The size of this packet as well as the value of the timer can be defined. This timer represents the maximum interval between each flush of the buffer that contains all events that we must write into the database. This part of the logger is explained in the section about improvements.

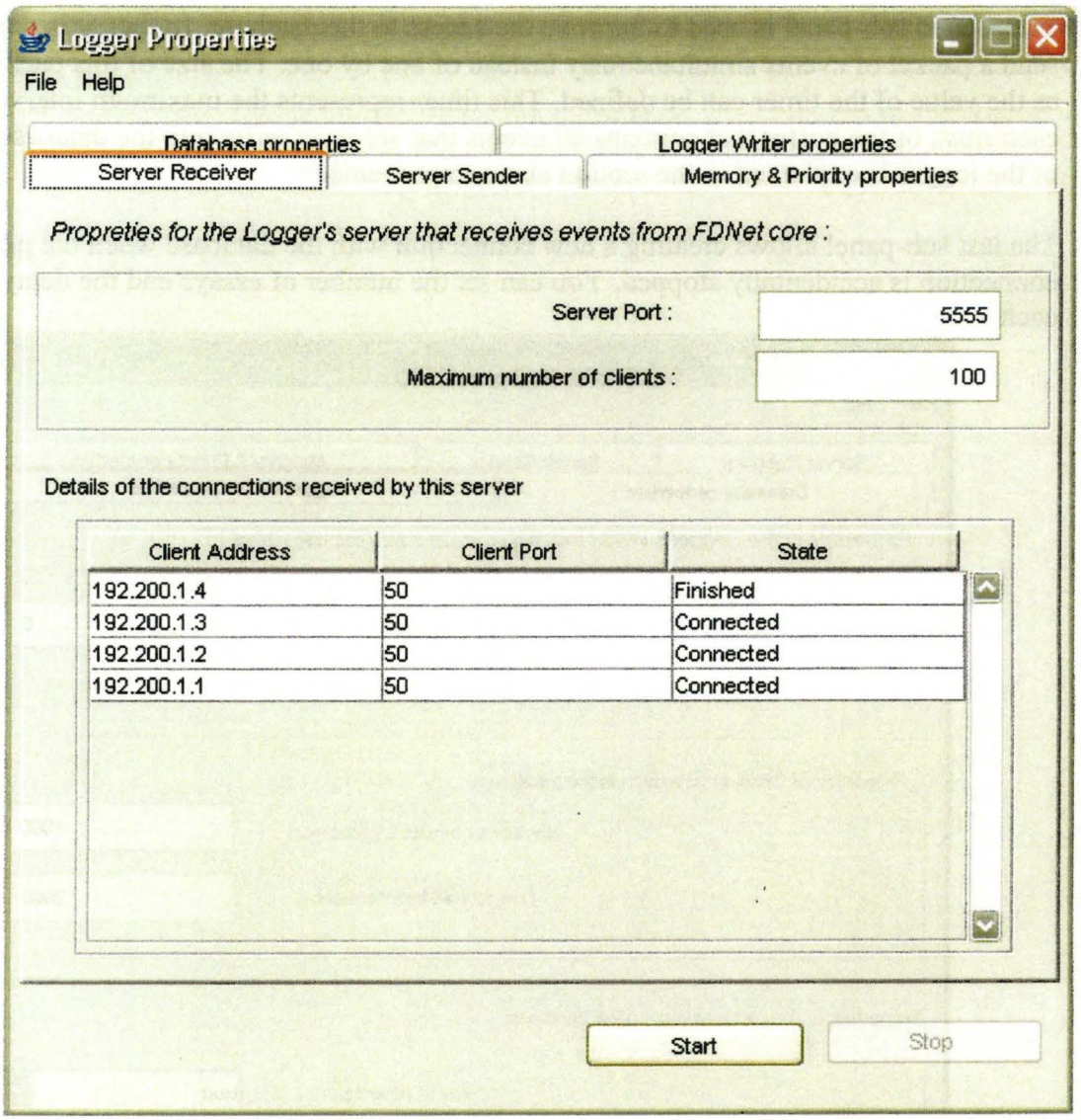
The last sub-panel allows creating a new connection with the database when the previous connection is accidentally stopped. You can set the number of essays and the delay between each essay.



D. Server Receiver

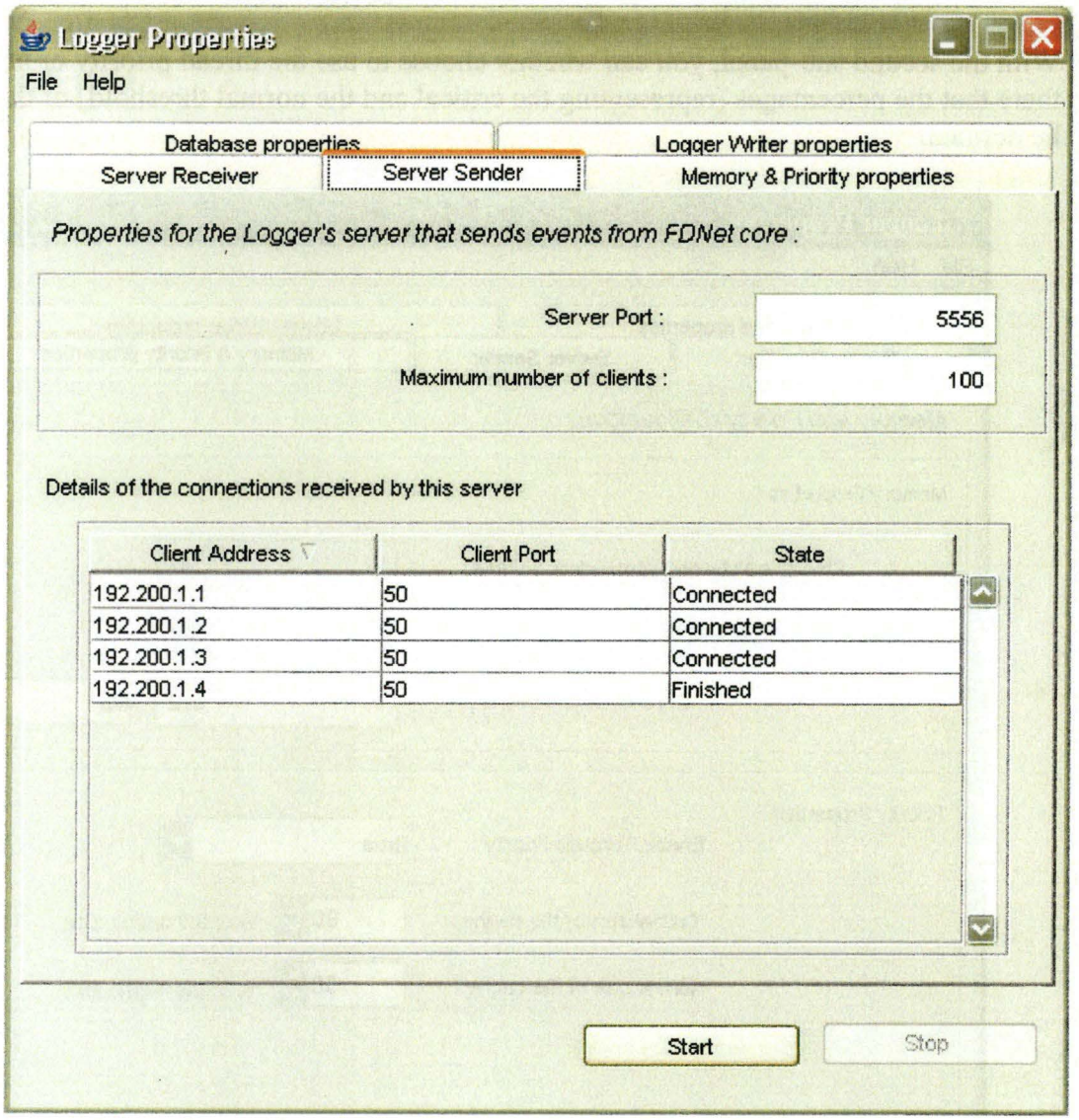
With this frame, it is possible to set the properties of the server that receives all events from the core.

The first sub-panel is used to set the port and the maximum number of client that it can serve simultaneously. The second sub-panel is used to check the state of all client connections that are connected with the server. The value of the connection state can be 'waiting', 'activated' and 'stopped'.



E. Server Sender

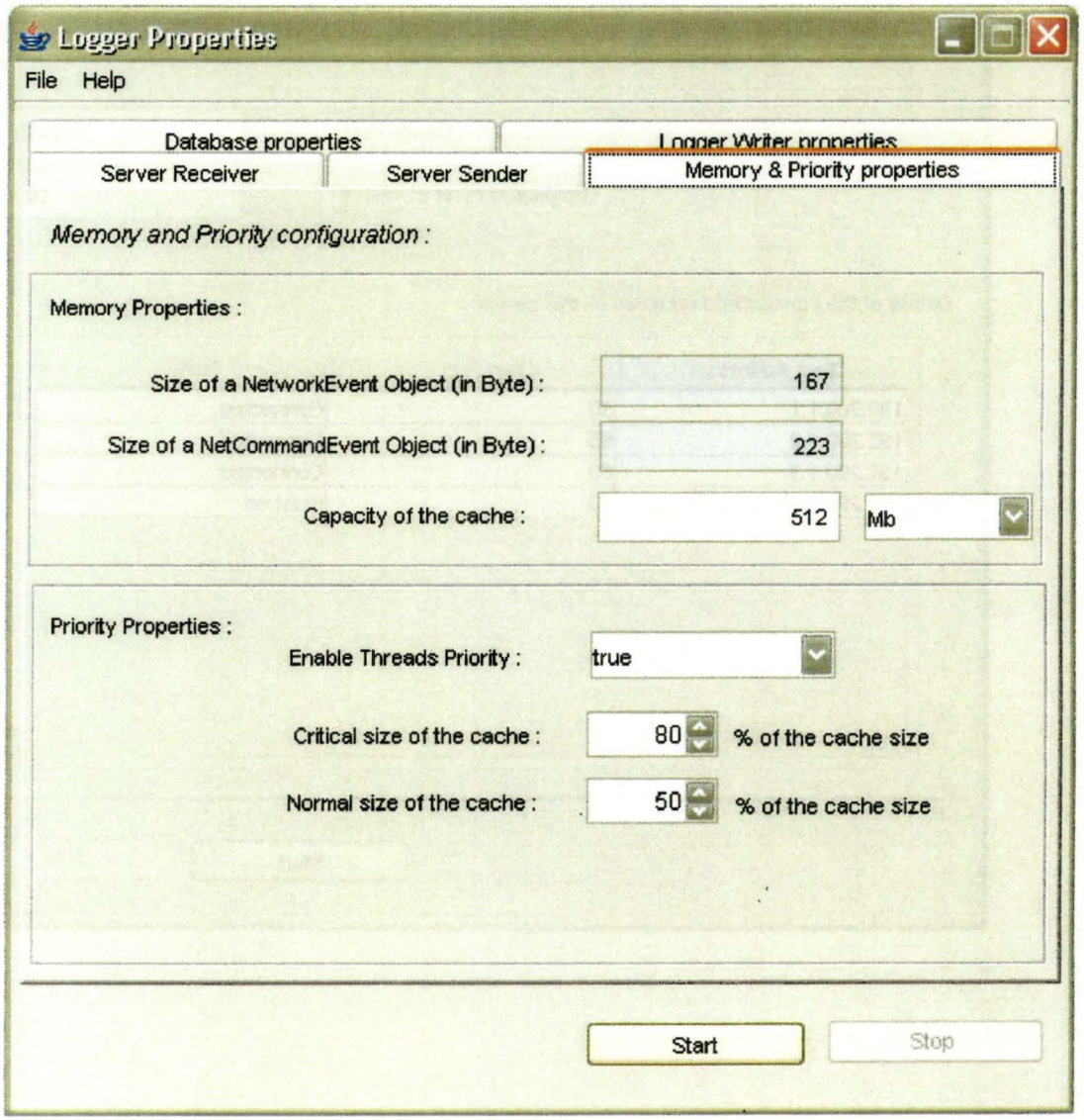
With this frame, it is possible to set the properties of the server that responds to viewer clients by sending all events they ask for. The content of this panel has the same meaning as the previous panel.



F. Memory and thread priority

This panel is used to set all information needed to use the thread priority improvement. The first sub-panel contains the size in Bytes of the different events that are used in the project represented by "NetwokEvent" and "NetCommandEvent" objects. It is also possible to set the size of the cache that can be used by the logger.

With the second sub-panel, you can whether choose to use the thread priority or not. It is also there that the percentages (representing the critical and the normal threshold) of the cache can be defined.



Chapter 5: The Human Interface

I. Introduction:

As said in the introduction, the FDNet technology is powerful but quite complicated to use. In the optic of FDNet's creators, the technology is to be used by a lot of people, ranging from computer scientists (of course) to electronic scientists, which, even if they know some things about computers, are perhaps not at ease with every piece of knowledge needed to create an FDNetwork from scratch. The fact is that we think that they shouldn't have to!

One of the main problems faced in Robotic Rescue Systems field is that the people creating the robots (electronic scientists) have to learn skills which are separate from their field of work (computer skills). They have to because, in general, teams are too tiny and only composed of electronic scientists. Often, this lack of computer scientists leads to poor results because the skills required to produce computer programs are not polished enough to produce good quality softwares.

The skills required to become a good computer scientist and a good electronic engineer being quite different, we cannot force electronic engineers to become good in computer science too. They should only have to bother about their own specialties. As a programmer will never have to be able to create computer circuitry to make good computer programs (even if it can help him become better), we cannot ask an electronic engineer to be able to code perfect high level applications.

Electronic engineers should only have to bother about creating robot abilities and design the way they work using applications handling all computer related specificities for them. This would allow them to work a lot faster (lifecycle would be highly reduced because of lots of problems solved automatically by the application) and a lot more well (specialists would only be asked to use their specific capabilities).

FDNet's Human Interface is an application going this way. Programmed by true computer scientists (at least wanting to become part of them), its architecture is designed to ensure that electronic engineers will be able to focus on their own work while relying on the Human Interface for all details they should not be bothered with.

Of course, for the time being, the Human Interface is only at its infant stage and quite a fair amount of work still has to be done in order to achieve this idea we are writing about but the version we could complete gives a glimpse of the potential of using an application of this kind to work on the programming part of rescue robots.

II. Human Interface's Aims:

Prior to the existence of the Human Interface (Further called H.I.), FDNet users had to write down the FDNetworks they wanted to test completely by hand. It means that if they wanted to test a special part of their robot (R.O.Q), they were forced to write down a huge quantity of code just to make simple tests. This had some serious implications:

- Users could not easily test robot's additions or modifications.
- Due to the amount of work required to write an FDNetwork, even a little one, more complicated tests were impossible to handle without an application helping the tester.
- Writing a complete FDNetwork including robot's full intelligence was impossible, or at least, working on an efficient intelligence was unfeasible.

H.I's main goal is to hide all computer related aspects of writing FDNetworks to allow Electronic engineers to focus on their own work. Giving a simple way for FDNet users to edit the FDNetworks allows them to manage the robot(s) before and while they are functioning.

Editing FDNetworks easily allows electronic engineers to:

- Reduce the Lifecycle of robot prototypes by allowing them to test directly the physical modifications they make to the robots.
- Enhance robot capabilities by giving them the power to create more advanced networks in a highly reduced time.
- Create more robust networks by hiding them their complexity and by controlling the actions available for every part of the network they are working on.

The H.I. is then a very important addition to FDNet and its use will allow FDNet to fulfill its greatest contract: Providing rescue robots engineers with a completely distributed network without forcing them to acquire new skills, unrelated to their own work

III. Definitions:

> The aim of the human interface is to give a simple way for FDNet users to manage the robot(s) before and while they are functioning.

By **managing**, we express the following things:

a. Allowing adding new information and processing agents dynamically and connecting them together to make them work in a specific way.
b. Allowing modifying the currently used network "on the fly".
c. Load network definitions, make them work and save the potentially made modifications.

By "**Data**", we imply any piece of information that can be used by relations.
By "**Relation**", we imply a processing agent, whose aim is to take some data in entry and to compute it in some way to create new data.
A "**Node**" is either a data or a relation.
By "**Reader**", we imply a link between a Data and a Relation allowing the Relation to read the value of the Data it is connected to.
By "**Writer**", we imply a link between a Data and a Relation allowing the Relation to write the value of the Data it is connected to.
By "**Connection**", we imply an interaction happening between a data and a relation. Connections can either be Readers or Writers
By "**Network Definition**" we imply a succession of data and relations, themselves followed by a list of the connections that theses nodes have between them.

IV. Human Interface architecture:

The Human Interface will consist of 2 different main parts. Theses are:

a. **The logical base:** this part consists of the Network Repository and its access system. This is the base of the architecture. Its aim is to handle all the low level tasks that have to be performed in order to be able to interact with FDNetworks.
b. **The end-user part:** While the first part only focuses on handling low level access to FDNetworks, this part will handle all the interactions with the end-user. In fact, this part consists of what is generally called the front-end. It will be divided in three specific points:
    - The static capabilities;
    - The dynamic capabilities;
    - The display of all this information in an easy to understand yet powerful way.

Following will be a specification of each of theses parts. At the end of the specification, you should be able to understand how FDNet's Human Interface really works and know every aspect of the methods we used to resolve the problem of its creation.

V. Specification of the logical base:

The first thing that the Human Interface must be capable to do is to save user-created FDNetworks to allow them to reuse the networks at a later time. But to reuse a network doesn't only means that the Human Interface has to be capable to load/save it. More than just this, the information loaded has to be easily usable at work-time; it means that, when the Human Interface displays graphics, all the content behind them has to be fetched in single logic entity.

The Human Interface is divided into two main parts:

- The logical part; whose aim is to share all information about a loaded FDNetwork.
- The graphical part which will use the logical part to display the Network in a user friendly manner and will allow users to edit theses networks without showing them all their specificities.

The repository system used to store FDNetworks is based upon the XML file and the system we used to share all FDNetwork information in work-time (i.e. inside the Human Interface) is called the "NetworkInfo"[1].

A. FDNetwork specification:

Find in annex n° 3 the DTD file used to ensure that FDNetworks stored in XML files are valid. In this DTD are written all constraints about FDNetworks, constraints which will have a great impact on the Human Interface itself.

Note that, when we were asked to program the Human Interface, the FDNet repository system was already decided and the DTD was totally defined too. Inserting changes in this definition was risky because the FDNet project was already running for two years when we were teamed up with the Japanese people and FDNetwork files had already been created. We did some modifications to the Network definition, as you will see with the introduction of FDNet Modules, but the original idea concerning the definition was not modified at all (the modifications followed the same logic as the part already defined).

[1] "NetworkInfo" was originally the name given to the principal Java class implementing the Logic part of the Human Interface but we decided to use it to define the whole logic system too.

NetworkInfo system aims:

As explained here above, the "NetworkInfo" system has been created to share all the logical information about an FDNetwork inside the Human Interface. It can be seen as an intermediate layer between the FDNetwork repository (the XML structure) and the Human Interface.

Its aim is double:

- Handle all operations on the repositories, thus saving and loading FDNetwork definitions.
- Place all the information read from the repository at the Human Interface's disposal in an easy yet powerful and complete manner.

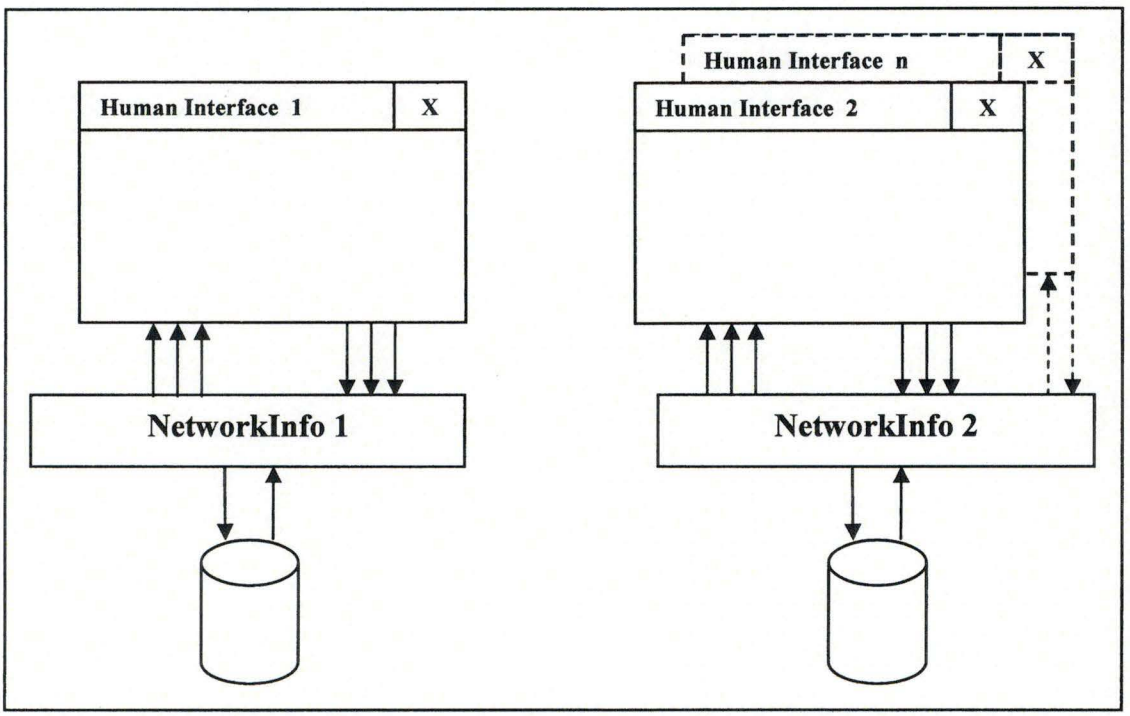


Figure 20: The NetworkInfo structure provides FDNetwork information to the Human Interface

The Figure 20 shows that the NetworkInfo system directly interacts with the repository to provide the information read in a more convenient way to the Human Interface(s).

This way of doing things is really interesting for two reasons:

- Architecture choices in the repository imply changes only in the NetworkInfo layer, the Human Interface is not even aware of them.
- Specific needs by Human Interfaces can be programmed easily, as long as the information can be found in any way inside the repository.

This latter point was used lots of time since the Human Interface we programmed had some very particular needs. With the system here above, once the NetworkInfo system reads FDNetwork's information inside the repository, it can compute it in complex ways in order to provide it to the Human Interface.

NetworkInfo system architecture:

Following is the NetworkInfo system architecture. This architecture is implemented as a set of Java classes. Each level (Network, Module, Node…) provides particular services based on information it stores.

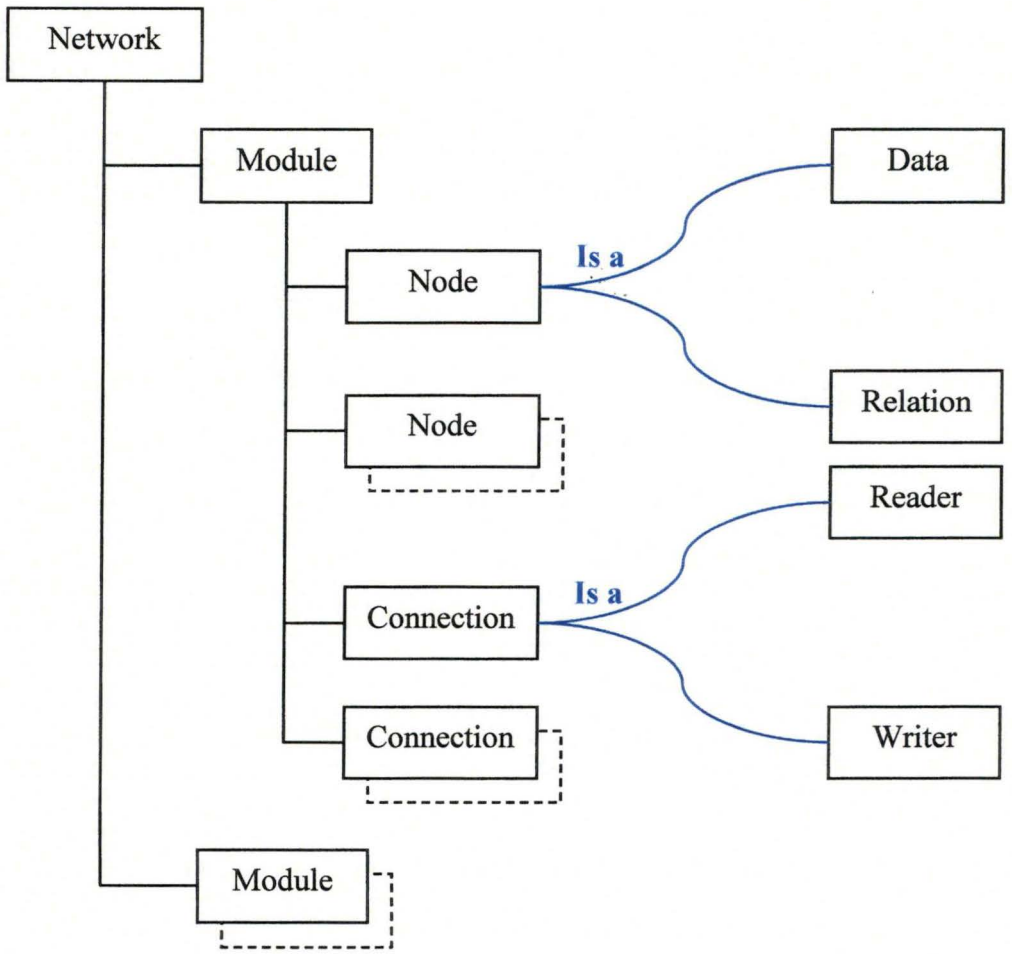


Figure 21: The NetworkInfo system architecture

As you can see, a Network is a gathering of modules, each of which can contain Nodes and Connections. Nodes and Connections bear the same definition as the one given in the first chapter.

The Network level is the most general one. Aside from containing all information about the FDNetwork read from the repository and services applying to the network itself, it also contains all the services applying to more than one module at a time.

Following is a set of services that can be found at this level:

- Add/delete a module to/from the network;
- List all the modules found in the network;
- Change the name of a module;
- Replace one node by another one (handling integrity constraints related to the node replaced)
- Find all connections linking a specific node (search is made inside all modules the network is made of)

Services provided here can be very advanced and help to create astonishing abilities really easily. As all information can be found starting from the NetworkInfo level, even very specific changes can be done from here.

The Module level is an enhancement we decided to apply to FDNet structure as soon as when we started our analysis. The reasons why we decided to add this level will be explained just here below, after the explanation of this schema. For now, let's just say that the Module level just allows dividing an FDNetwork into more tiny parts, each of which can be treated separately.

At this level, you will find all information related to one specific Module:

- Module's name;
- Its load modifier (Does this Module has to be loaded in memory when user starts the network or not?)
- Datas, Relations, Readers and Writers quantity contained in the Module

You will also find here all services related to gathering of Nodes and Connections, such as:

- Retrieve information about a Node/Connection existing inside the Module.
- Retrieve all Connections related to a specific Node (existing inside the Module).
- Delete a Node/Connection existing inside the Module and, in the case of a Node, delete all the Connections related to it too.

Nodes and Connections, apart from being the most specific levels in an FDNetwork, are also the only ones containing "real" information about the network itself.

Network and Module levels are only structural levels: they allow putting some order in a network but they don't contain a bit of information about network's capabilities. The levels that are really modeling an FDNetwork are the **Node and Connection levels:**

- Data Nodes contains the code allowing to fetch (from sensors) or create (from information inside the Network while it is working) values and to provide them to the Relation Nodes.
- Relation Nodes contains code related to Robot's intelligence. They read values found in the Data Nodes and compute them in a specific way to create new Datas as output.
- Reader and Writer Connections are the links between the Data and Relation Nodes. As such, they contain information about how the links have to be done, the way the links work and so on.

Services related to gathering information about the Nodes and the Connections can be found in these levels.

It is also interesting to note that Nodes do not know links they have with other Nodes. All information about linking is to be found in the Connections. Nodes can only provide information about themselves but Connections provide information about the two Nodes they are connecting (A Data and a Relation). This is an important fact since it has serious implications in the whole architecture.

VI. The Module extension:

The module extension is an improvement we decided to bring about in order to improve FDNet scalability and utility. In fact, before using the new module level, all Nodes and Connections were part of the same unique entity. This was quite a limitation:

- Two Nodes could not have the same name;
- Since an FDNetwork was a single entity, working on a specific part of it was impossible;
- Dividing an FDNetwork into smaller, clearer pieces was impossible;
- Upon work-time, FDNetwork had to be entirely loaded before it was possible to launch it.

As we had to create a User Interface that could be used easily, we had to look at FDNet's structure with a sharp eye, in order to find the best way possible to make it simple without limiting its power.

This way of looking at FDNet allowed us to find this flaw in the architecture as soon as in the first part of our work: the understanding part. The "Module extension" could then be thought directly in the analysis part and be simply included in the Human Interface at implementation time.

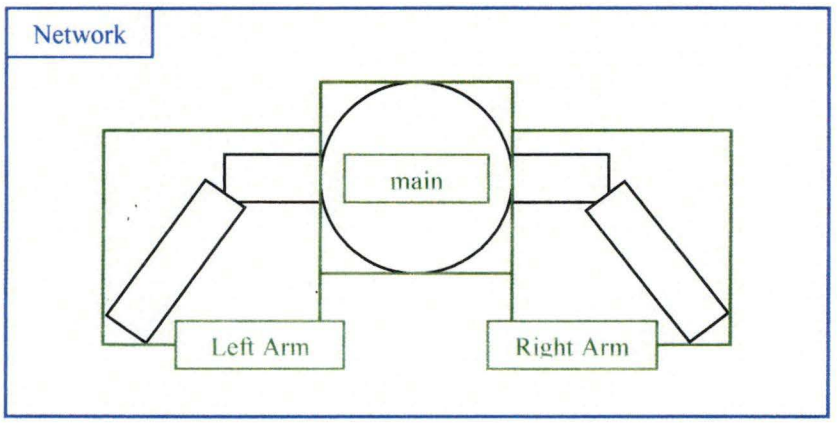
Inserting this improvement at the very beginning of the process even enhanced its own utility: Every time we had to think about ways to implement specific parts of the Human Interface, the Module system was present in our minds and gained capabilities that would have not been possible to discover otherwise.

We will now explain the main improvements we found out through the use of the Module extension. For each improvement will be explained how can Modules enhance FDNet utility and usability. Some examples will also be provided and a little discussion about future possible modifications will close the subject.

A. Introduction of a more complete division of FDNetworks:

Problem :	Two Nodes cannot have the same name in the Network
Solution :	It is now possible for Nodes to have the same name, as long as the Modules in which they reside are different

Example:



Without Modules, all the information defining the robot would exist in the same namespace: the Network.

With the introduction of the « Module level » enhancement, all robot information can be separated in more than one namespace.
As you see here in the example, the network has been divided into 3 parts:

- The main part (containing all mandatory information, robot's base) ;
- The left arm ;
- The right arm.

Due to constraints (2 nodes cannot have the same name in the same network), Nodes from the left leg and the right leg will have to be named differently. For example, thoses names can be chosen:

- LeftArmShoulder
- LeftArmFront
- RightArmShoulder
- RightArmFront

Note that, using this style, it is not possible to ask questions like the following ones to the Network:

- "How many arms does the robot have?"
- "Of how many parts consists one arm?"
- "Are there fingers to robot's Arms?"

By using the Modules as an intermediate level, we can change the constraints to: « Two Nodes cannot have the same name in the same **Module** ». This adds a tremendous clarity to the robots (and the bigger the Network, the bigger the profit) and allows us to add a whole new set of possible services to the FDNet architecture. Questions like the ones asked before can now find an answer easily.

Figure 22: Consequences of Module subdivision

As you see in the Figure 22, adding the Module level is really interesting for the clarity of the network. It is far easier to work since you can divide the network in pieces, each one of them having a particular meaning. With a good division into Modules, an FDNet user will make networks easier to understand and to maintain.

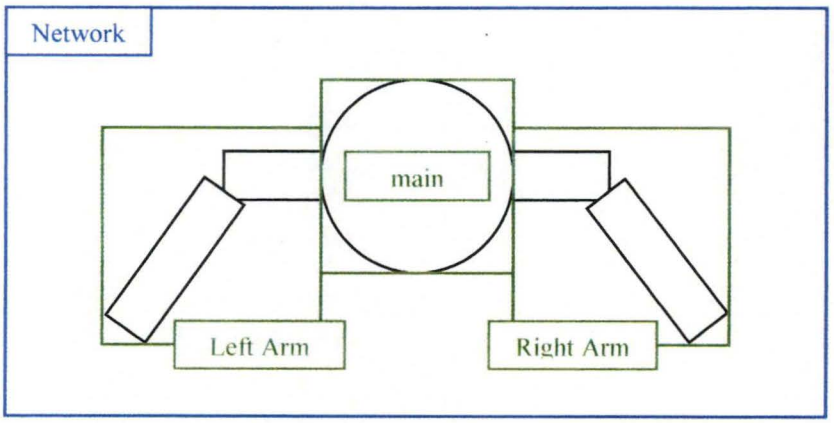
Also note that, with the introduction of this division, a whole new set of questions about network's structure can be resolved easily.

B. Incremental loading:

Problem :	The entire network must be loaded in memory at the same time
Solution :	Every Module contains information about when it has to be loaded: At start time or not.

As you know since we have discussed about this already, FDNetworks can be quite huge. Loading entire FDNetworks as soon as a little part of them is needed can be a big problem (in terms of speed, of memory usage, on utility,…)

The use of the Module extension corrects this problem quite well by attaching some information to every Module contained in an FDNetwork. This information is called the "LoadModifier".



ModuleName	LoadAtStart
Main	Yes
LeftArm	Yes
RightArm	no

Figure 23: Incremental loading of Modules

Figure 23 shows that when the user will want to load the FDNetwork in memory to use it, two Modules will really be loaded: "*Main*" and "*LeftArm*". The "*RightArm*" Module will not. All information relative to it will still be available but the Module will simply not be usable as soon as it is not loaded.

Also note that the "Main" Module is a special one. It MUST be loaded at start time. In fact, this Module is always present in an FDNetwork and its aim is to contain all the core functionalities.

For example, the creator of a vacuum cleaner robot will always want the robot to be able to clean the floor. But, vacuum cleaners can come with lots of accessories, which you don't especially use all the time. The "Cleaning" part of the network will be written in the "Main" module while all the functionalities related to the use of accessories will be written in Modules.

Currently, the "Incremental loading" only consist of telling if a Module has to be loaded at start time or not. It will be interesting to bring this improvement to a further step (Load Modules only if other Modules are loaded, load after some time, unload automatically upon reception of special events,…) if the current implementation is really used.

C. Real Time modifications:

Problem :	It is false to think that creating real network functionalities in real-time is possible
Solution :	Modules can be loaded any time needed and thus, prepared before work-time and started when they are necessary, thus giving the impression that functionalities are added to the Robots *while* it is working.

One of the requests we had to fulfil was that FDNetworks had to be still modifiable in real-time (i.e. while they are loaded in memory and are actually doing something interesting). This job had to be handled by the Human Interface too and could be improved a lot too.

In fact, before the introduction of Modules, if a user wanted to add some functionality in FDNetworks, he had to create all the network entities (Nodes, Connections) on the fly! Since creating a real functionality (even a basic one) requires at least 20 entities, we can say that to modify FDNetworks' structure in real-time is not possible.

We realized that the only things that are modifiable in real-time are Nodes' parameters. Even if it seems to be very little, setting special values to parameters of Nodes can totally change the way an FDNetwork works. Creating special Entity constructions whose aim is to receive parameters in real-time and to react to them is quite straightforward.

The idea here is to create a complete network at the beginning (i.e. at edition time) but to fill it with parts that will do nothing unless special values are read from datas modified in real-time.

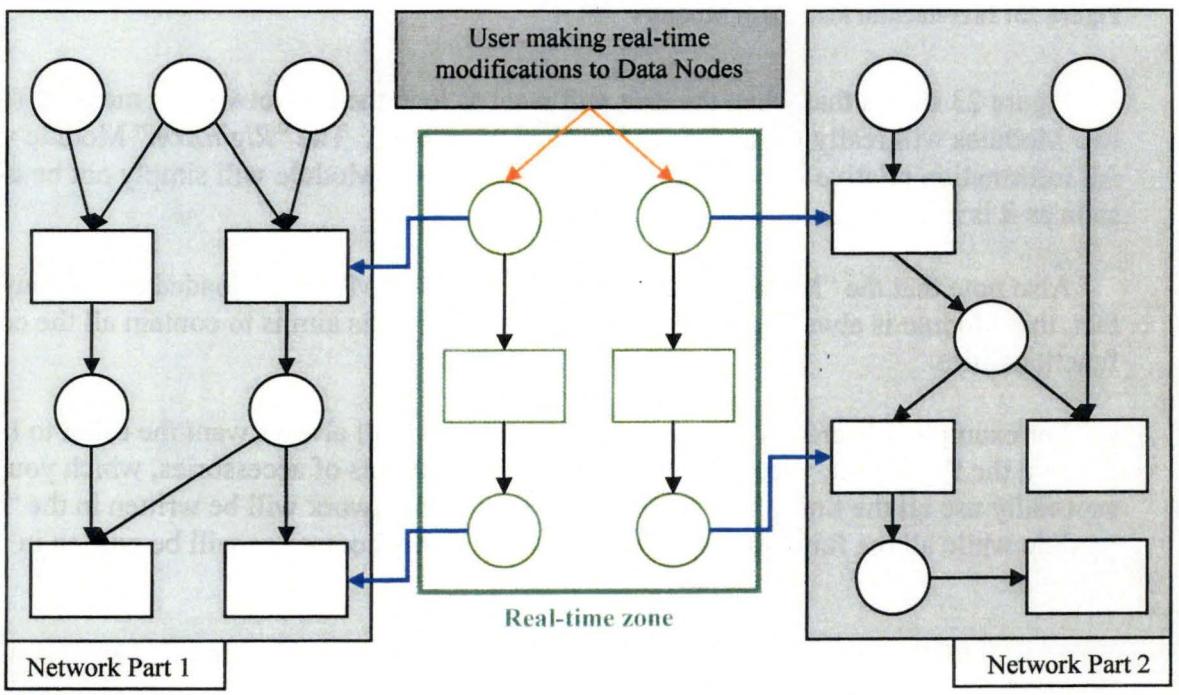


Figure 24: Trick allowing real-time modifications

In the example shown in Figure 24, the three parts (Parts 1&2 and Real-Time zone) are loaded in memory and the FDNetwork is already working. Parts 1&2 will never be directly updated by the User; only Data Nodes from the "Real-time zone" will be.

At the beginning, the Datas from the real-time zone are filled with values such as they don't have any influence on the values computed by Relations reading them. It means that the network will work as if the real-time zone had no influence at all.

Upon modifications made by the User, (through the Orange arrows), Relations reading Datas in the real-time zone will change their way of working. Of course, theses Relations have to be prepared to the values they read from the "real-time Datas" but, if this is done, it can really become possible to drive a Network easily.

Note that the real-time Datas can be read by "real-time Relations" too, which will create other real-time Datas (which can, of course, be used to further direct Network's work).

This system can work very well but will become messy on big networks. Having lots of parts loaded but doing absolutely nothing most of the time (real-time datas staying unmodified) will lead to misunderstandings and is not very efficient in terms of memory usage.

Using the modules will simply wipe this problem out. In fact, As FDNetworks are separated in Modules, allowing to load some of them or not (may it be at start time (Point 2) or not) becomes a trivial problem. They allow a far more advanced network handling than the original version of FDNet.

Solving this problem will simply consist of creating specialized modules at edition time, theses modules consisting on Nodes and Connections constructions handling specific functionalities, and to load/unload them at work-time just whenever needed. Real-time modification is then properly handled, the modules loaded in memory automatically reading Datas needed to perform their functionalities.

Of course problems arise:

- What to do with modules that need Datas from other Modules (dependencies)?
- How can we solve problems consisting of Modules incompatibilities?
- How can we handle more than one Module doing the same activity?

Lots of problems like this one have to be solved in order to improve the way FDNet works but this is an implementation problem (which, in fact, can and has been (partially) solved).

D. Easy network edition:

Problem :	When editing an FDNetwork, so many entities can exist that it is nearly impossible to understand network's structure and edit it with full comprehension.
Solution :	At edition time, allow users to Show/Hide Modules.

Perhaps the most interesting thing about Modules is this ability. Modules containing information about all the entities they contain, it is really easy to show/hide Entities on a Module basis. We can also think about enhancements that benefit from this fact. Setting all the Nodes and Connections of Modules to special colours, for example, will also greatly help the user while creating the FDNetworks.

The end-user part:

This part consists of all what the final user can see and all the functionalities created to enhance his experience as an FDNetworks creator.

The end-user part of the Human Interface can be divided in three distinct branches:

- *The static capabilities:*
  All the capabilities related to the preparation of a Network. It means its creation while it is not loaded on any Robot.
- *The dynamic capabilities:*
  All the capabilities related to synchronization between the interface and the Network while it is working (i.e. : real-time capabilities)
- *The display:*
  This part consists of the wrapping used to make it possible for the user to use all the capabilities included in the Human Interface.

VII. The Interface's static capabilities:

A. Load/save networks:

The Network repository being used, loading and saving FDNetworks is easily achieved. In fact, file structure is not even known by the end user. Users don't have to bother about file format anymore.

This is a great improvement from before since, without Human Interface, users had to create their FDNetworks entirely by hand, forcing them to understand XML File format and to resolve by themselves the constraints.

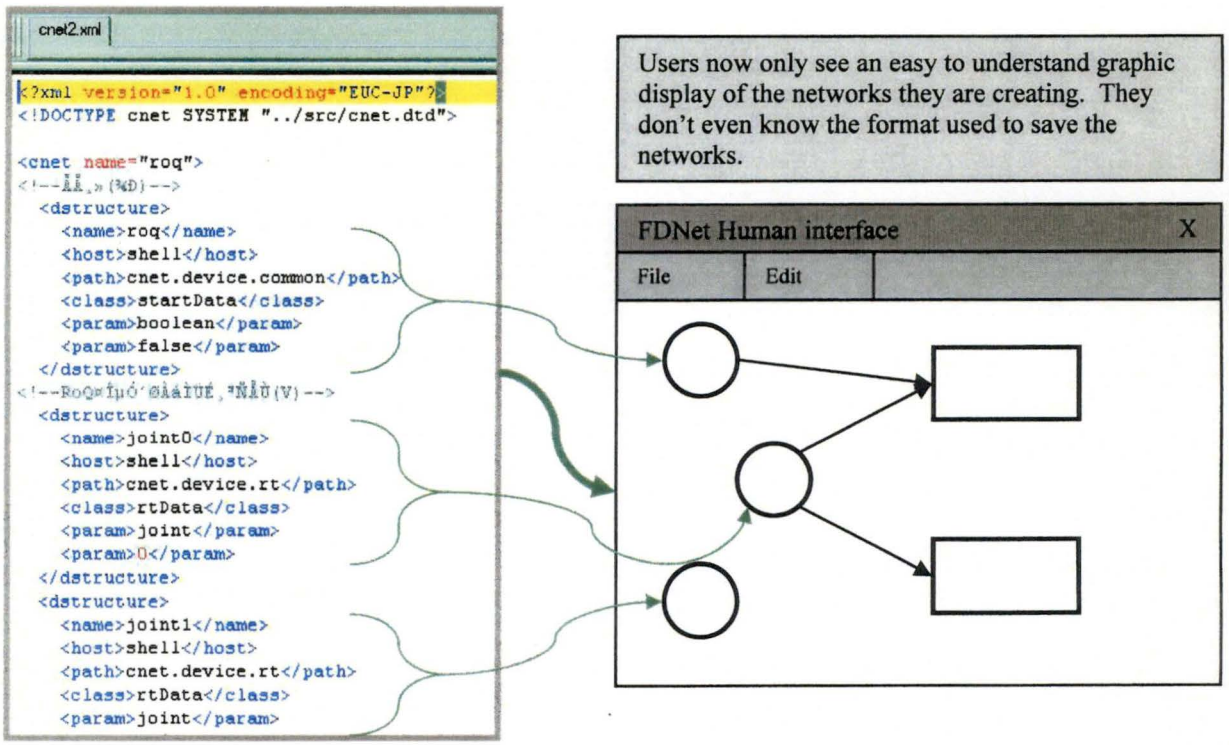


Figure 25: The file format used is hidden to the final user.

B. Network entities edition:

The main aim of the Human Interface is to allow users to create and edit FDNetworks, may it be from scratch or not (editing a previously loaded FDNetwork). Editing a network means to be able to edit all the entities a network consist of. Theses entities are:

- The Network itself;
- Network's Modules;
- Modules' Data & Relation Nodes;
- Modules' Reader and Writer Connections.

We have separated the edition task in two parts, parts related to dependencies. In fact, editing an entity can have implications of two types:

- Basic Edition: Edition have influence on the entity edited only;
- Advanced Edition: Edition also has influence on other entities as well.

Theses two kinds of edition will be specified and described here below.

Basic edition:

With the current version, Network's basic edition only allows to change Network's name. The name of a Network, as its identifier says, only allows giving a name to the Network. This parameter has no serious implications for the time being.

Modules can have their loading modifier edited. As said before, this parameter allows users to specify if a Module is to be loaded directly when the Network is loaded in memory or not. Note that, for the time being, no dependency checks are made to know if Modules have relations with other ones.

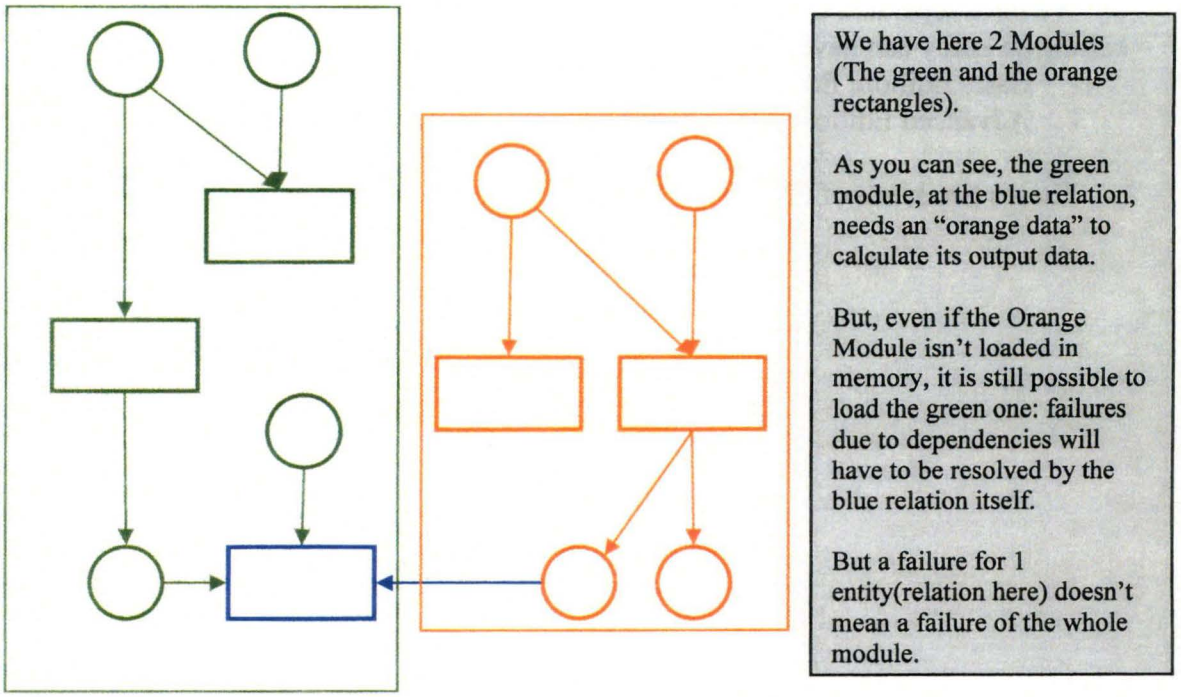


Figure 26: Interaction between Nodes from different modules

It would be interesting, for future versions, to ask the user if he wants to resolve Modules dependencies and to load what is necessary for "his" Module to work properly.

Nodes, may they be Datas or Relations, can have their parameters modified at any time. It is possible to add, edit and/or remove Node's parameter without any problem.

Advanced edition:

Nearly all the edition possible can be seen as advanced edition. This is due to the fact that, every FDNet information being related to each other, every time we edit information, we have to ensure that related information is still coherent. Dependencies must be kept correct.

On the Module level:

Adding a new Module to the Network:
Upon Module addition, we have to ensure that the name of the module is not yet used inside the network (As seen while explaining the NetworkInfo system, two modules cannot have the same name).

Deleting a module from the Network:
Module deletion is a little trickier: If the Network consist of more than one Module (which will nearly always be the case), there will be interactions between modules. While deleting a Module, interactions of its entities will have to be resolved. The resolution method chosen is the deletion of every Connection related to Nodes deleted, whatever the Module they exist in.

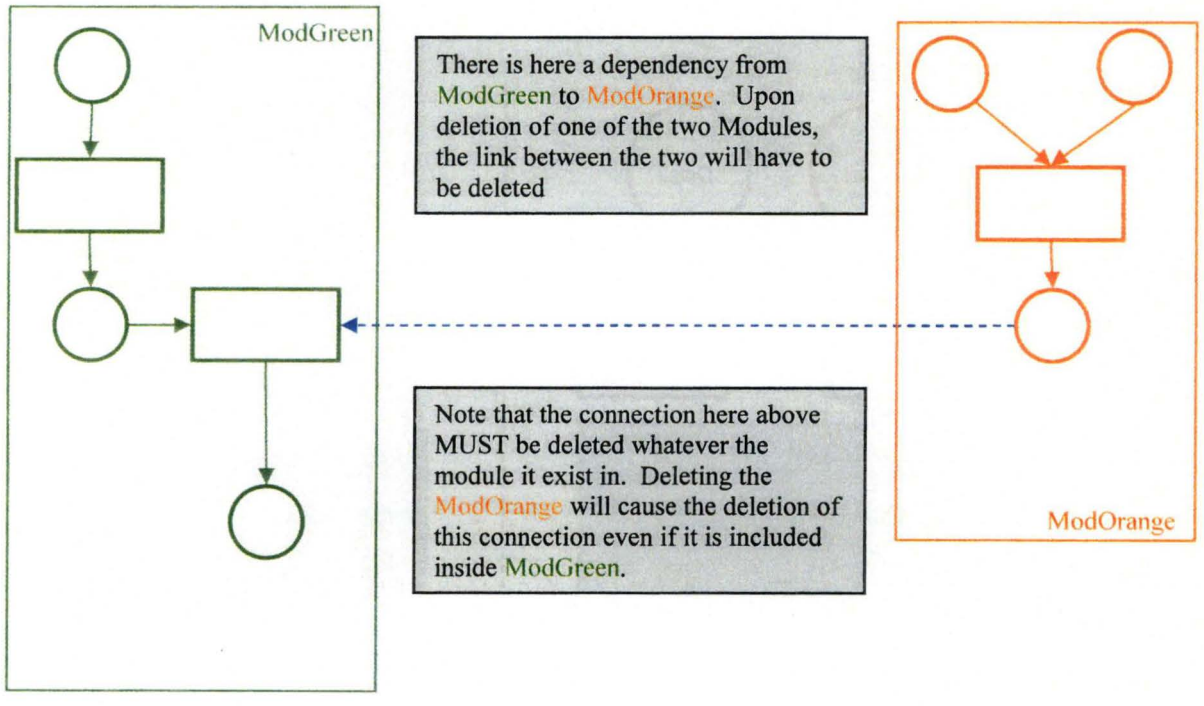


Figure 27: Modification of Connections upon changes in other Modules

Note that, even if the Connections are deleted, no Node is. Nodes dependencies are not resolved, but we will discuss about it a little bit later.

Changing Module's name:
It will only be possible to give another name to a Module if none has the same.

On the Node level:

Due to the enhancements provided through the use of Modules, Nodes can now have the same name if they exist in different Modules. But, what happens when you try to give to a Node a name already used inside the same Module?

In fact, instead of just preventing the user to do so, we have created two functionalities allowing him to choose the best way to make changes inside the Module. Theses two functionalities are called "Replace Node" and "Merge Nodes".

They allow very powerful, high level modifications in an instant. They will now both be explained to you.

Replace Node: Overwriting a Node with another one, deleting all Connections from the replaced Node and using only the ones from the Replacement one.

Replacing a Node will handle for the user all the problems related to finding existing Node with the same name, to delete it and to respect constraints from the Connections.

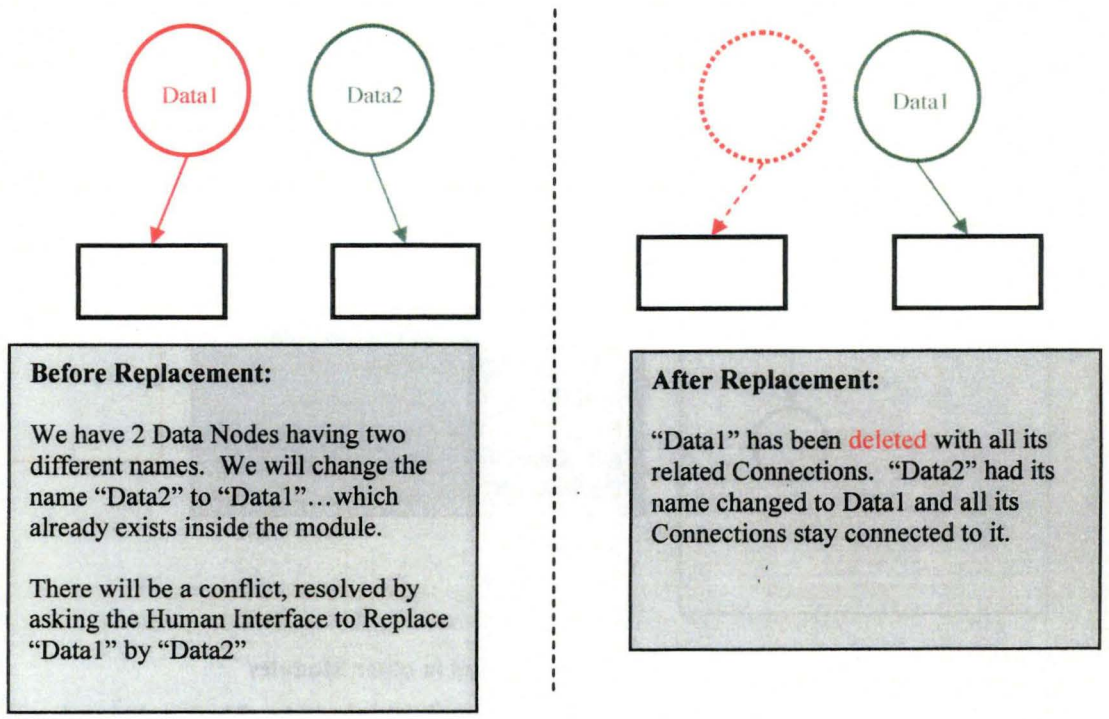


Figure 28: Replacement of a Data Node

Replacing Nodes is a really interesting system. Frequently, users will want to override old versions of Datas with new ones. This system allows users to create "beta versions" of their Datas, replacing the currently used version of a Data with its enhanced beta version when this one is ready to be used.

Merge nodes: Using information from two Nodes to create an only resulting one. All the previous Connections will be redirected to the merged node.

Merging Nodes is the most powerful ability currently implemented in the Human Interface. Although it is easy to understand the meaning, it is not that easy to use it. But a good comprehension of this ability will provide the user with shortcuts while editing networks.

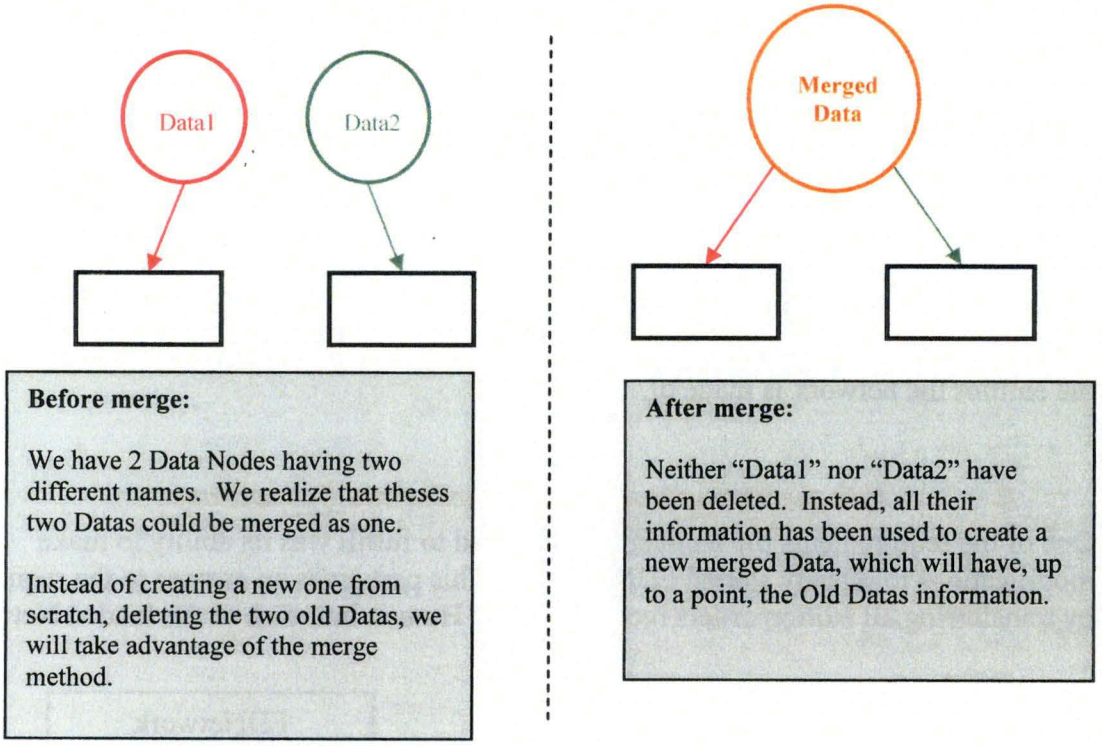


Figure 29: Fusion of Data Nodes

It is possible to merge any kind of Nodes, Datas or Relations.

If, while merging Nodes, it is found that information is duplicated, the user will be asked which information he wants to save in the resulting Node. For example, while merging parameters, the user will be asked if he wants to save parameters from the first Node only, from the second one only or from both of them.

Note also that all the Connections from the old Nodes are redirected to the merged one.

VIII. The dynamic capabilities:

As explained before, through the use of the Human Interface, it must be possible to view loaded network's work. More than this, the Human Interface must also provide users with tools allowing them to edit FDNetworks at work time.

This part will explain the functionalities implemented to ensure that users have a good control over the networks they are using.

Note that the implementation of some of the dynamic capabilities has been done by another Belgian student while we were in Japan. If you want more information about how theses particular parts of the Human Interface have been done, please refer to Mr Lambot's thesis[1].

A. Start/Stop the Network:

The Human Interface allows, of course, to start selected FDNetworks and to stop them whenever decided. Starting a network means to connect to the computer which will host the Network itself and to create this network by providing to the host all information related to the entities the network is made of.

B. Edit the Network: Add/Delete Nodes and Connections:

One of the requirements the Human Interface had to fulfill was its ability to make modifications to already started FDNetworks. This part gives an answer to the requirement by transferring all edition orders received by the Human Interface to the loaded Network.

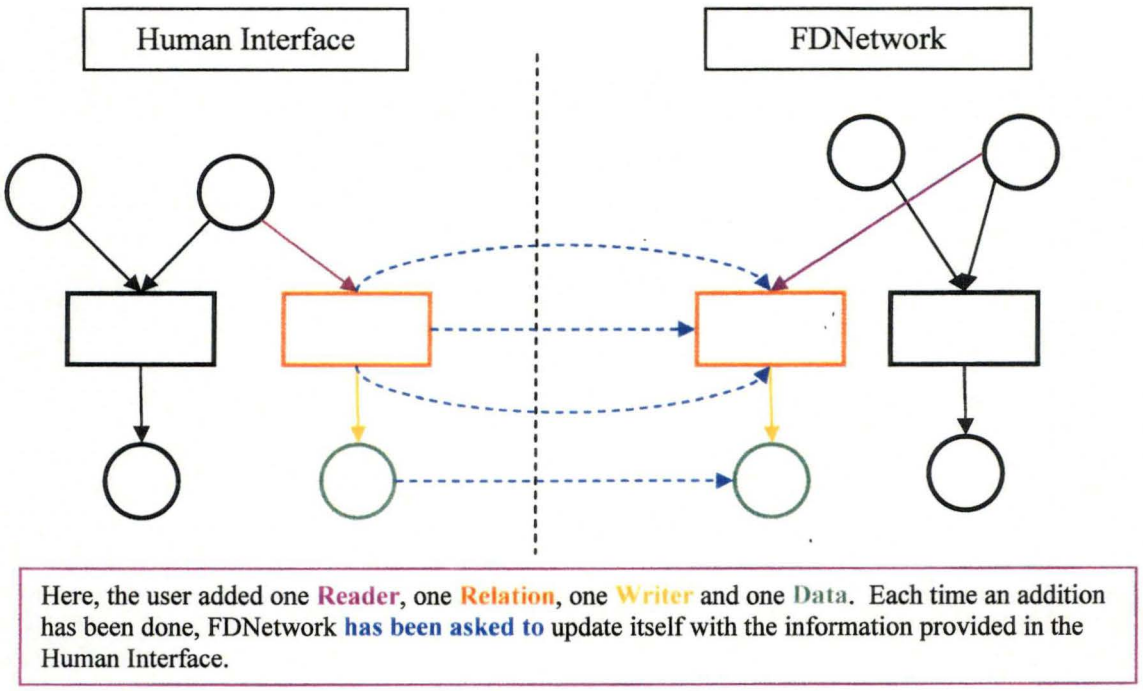


Figure 30: Repercussion of modifications in the Human Interface on the FDNetwork

[1] See [Lambot 2003]

Note also that, as loaded FDNetworks can produce new entities by themselves too, the Modifications made inside the network also have to be transferred to the Human Interface.

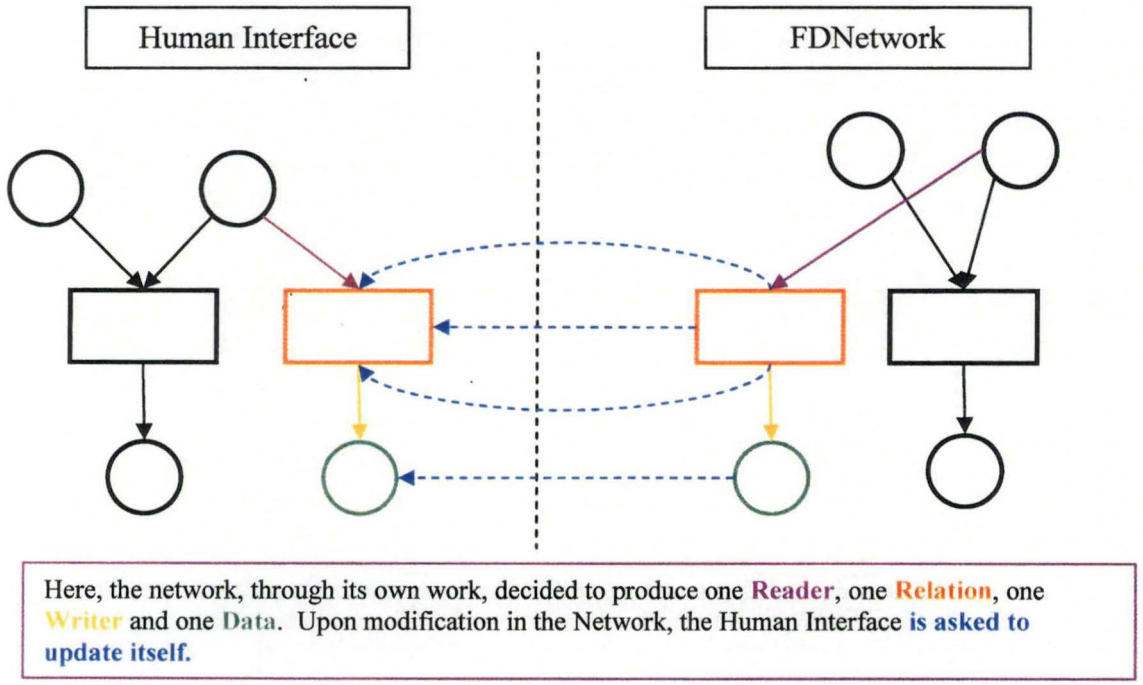


Figure 31: Repercussion of modifications in the FDNetwork on the Human Interface

C. Load/unload Modules:

As explained when presenting the "Module extension", it is possible to modify the network in Real-Time through loading and unloading Modules anytime. While unloaded Modules will simply stop to work, newly loaded Modules will start their own job depending on the entities already available and provide their output to any Relation needing them. As FDNet is a flat network, no dependencies between Modules are checked. It is the role of an entity to handle the cases when necessary information (a Data existing in a currently not loaded Module) is unavailable.

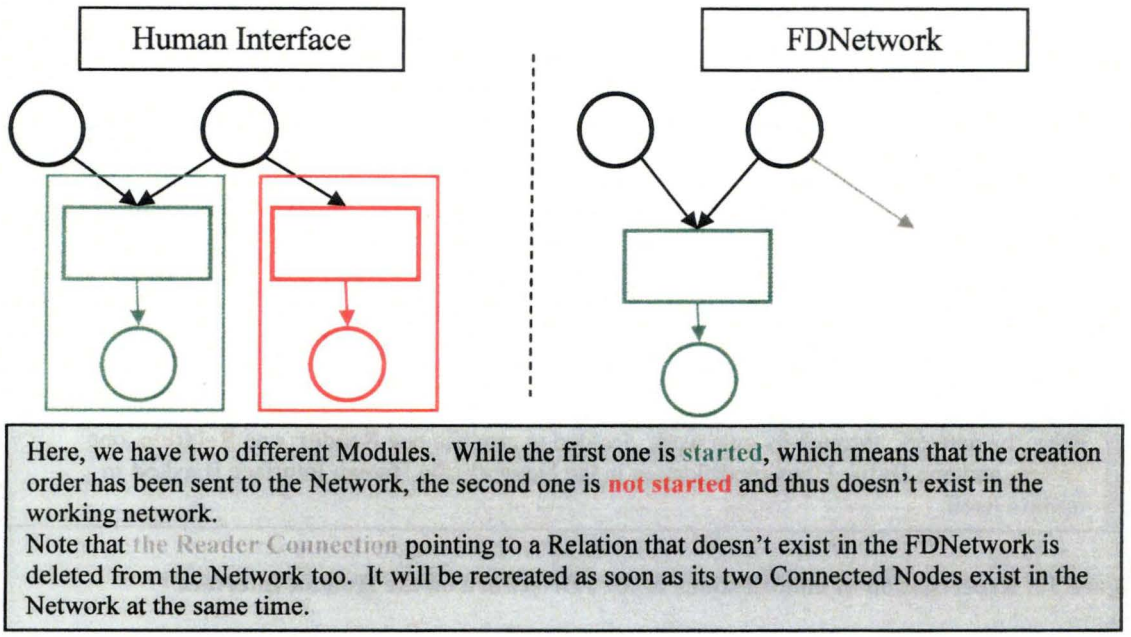


Figure 32: Unloading Modules in a working FDNetwork

It is currently impossible to unload Modules from the working network without having to delete them. FDNet specification (in its current state) doesn't allow this kind of ability (upon "unload Module order", all the Nodes and Connections plus their dependencies will be deleted from the network). We think that the working FDNetwork should be able to put Modules "on hold" without necessarily delete them. This could enhance FDNet in some cases.

D. View Network evolution:

One of the biggest requirements concerning the dynamic abilities was to be able to follow Data Nodes' values evolution at work-time. This required creating what has been called the "FDNetwork State Viewer", which allows users to see the evolution of Data's values in Real-Time in a graphical manner.

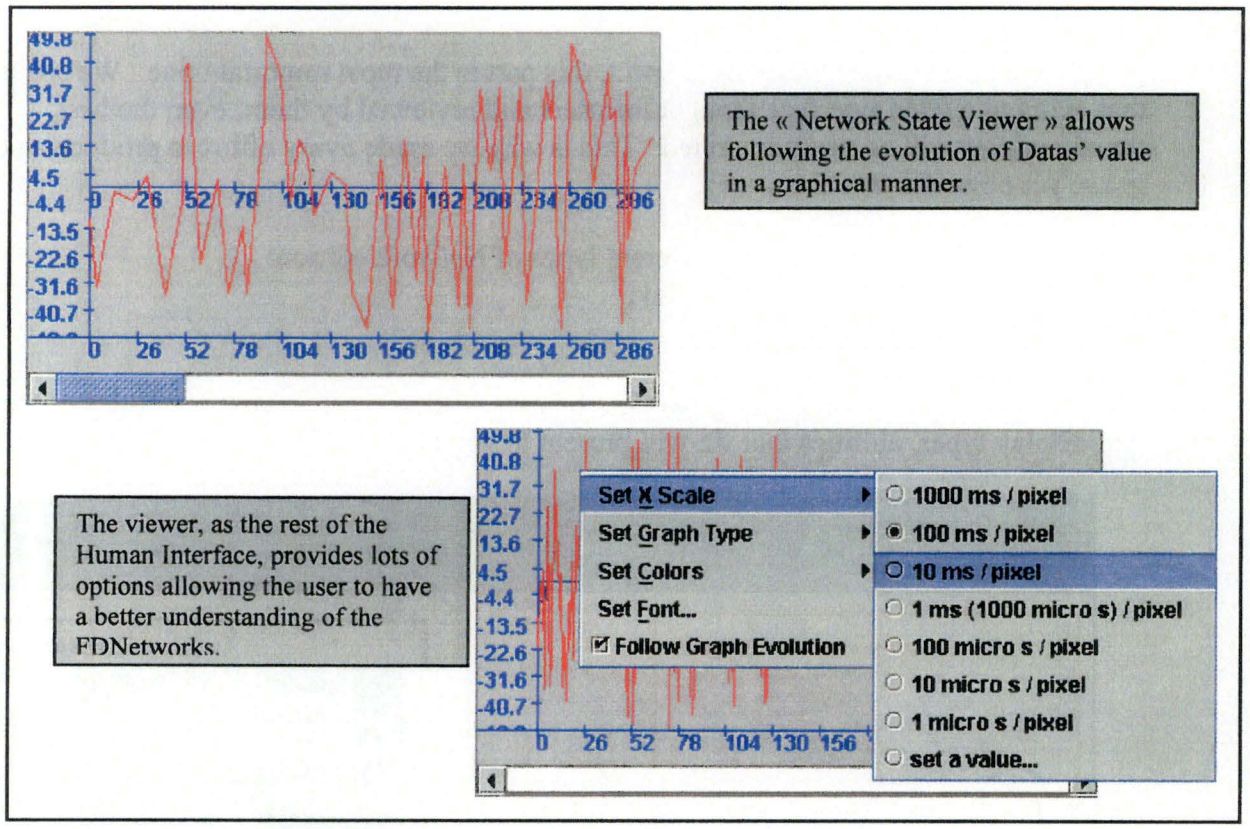


Figure 33: Using the Viewer to analyze the evolution network

As the "Network State Viewer" constitutes the main work of Mr Lambot, please refer to his thesis[1] in order to have a full explanation of its capabilities.

[1] [Lambot 2003]

IX. Human Interface's Display:

This section will consist of the specification[1] of the display the Human Interface provides. The display is just the choices we made to ensure the user will be able to use the Human Interface to its greatest extend. It can be seen as the wrapping used to appeal the user.

In terms of logic functionalities, this part produces nothing. It doesn't provide the user with tricks allowing him to create better networks. It has no influence on FDNet's "Utility" aspect.

But concerning the "Usability" aspect, this part is the most important one. We are sure that, without a good interface thought for users and reviewed by them, even the best functionalities will be underexploited. This is why we made every effort to produce an easy to use yet powerful interface.

The Human Interface provides 2 different types of Network edition:

- A graphically-based display;
- A text-based display.

During the edition of its FDNetworks, the user will certainly need the abilities of theses two display types, abilities that we will present to you now.

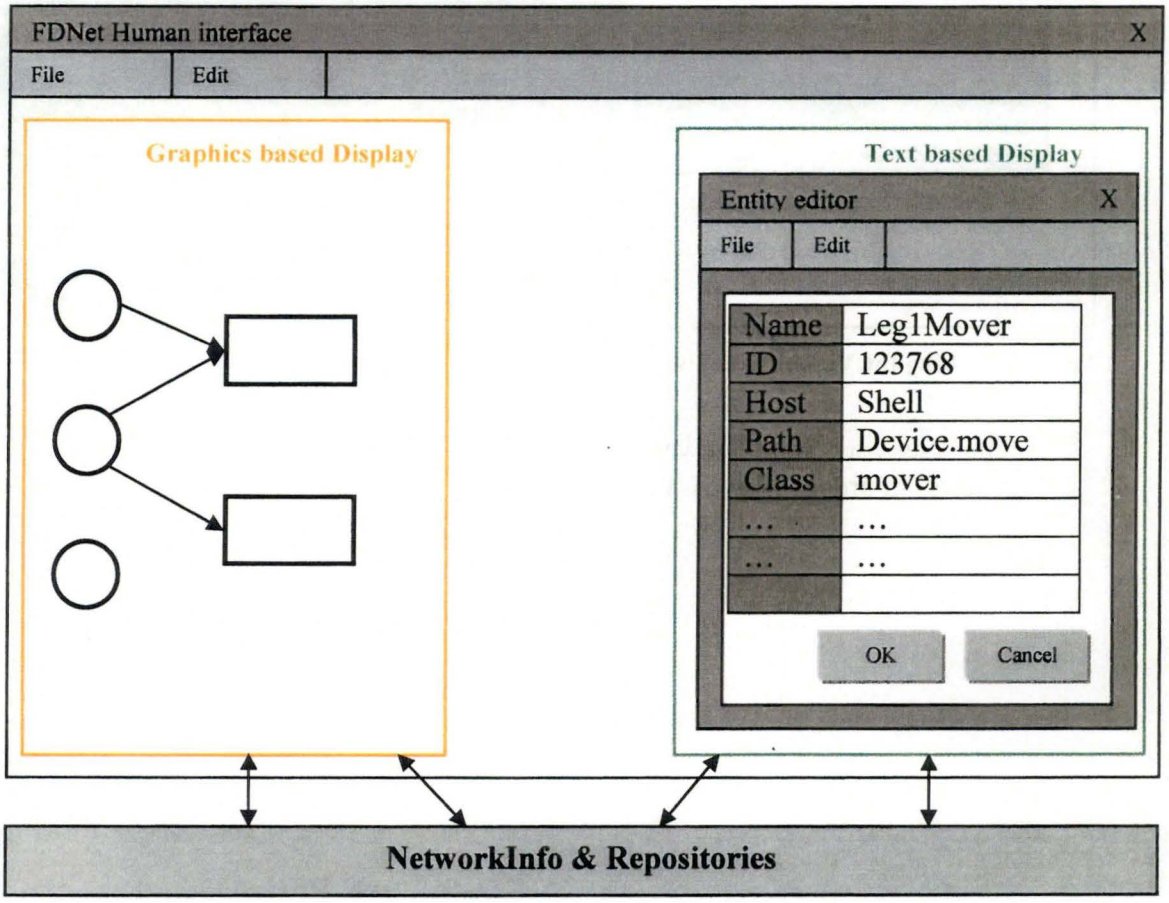
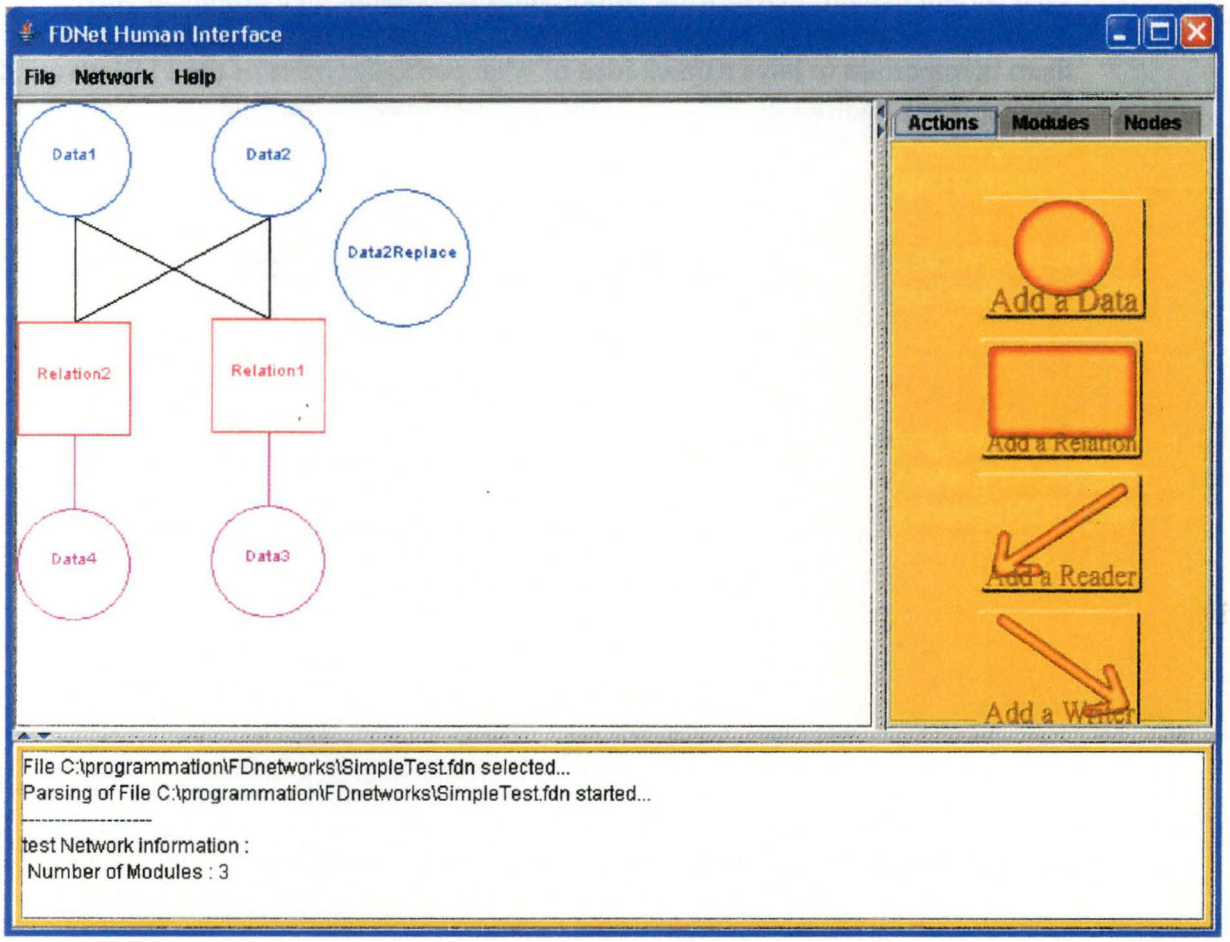


Figure 34: The Human Interface's display – Graphics and Text based display

[1] As the Human Interface is quite complex to apprehend, remember to use the Glossary to have quick explanation of terms you are not familiar with.

A. The Graphically-based display:



The graphically-based display is the main response to the usability requirement. Indeed, as it was very important for users to have a user-friendly interface at disposal. As we said in this chapter's introduction, electronic engineers don't want to and don't have to be programming gurus to create their robots' intelligence.

Pay attention to the fact that we do not say here that programming the robot intelligence can be done by someone who doesn't understand computer science. But, as it is one of the aims of FDNet, allowing users to reuse parts created by other ones also means that they have to be able to use third-party programmed components without having to touch to the code. When we say that the Human Interface has to be easy to use, we want to say that an electronic engineer working on assembling his own robots from different parts has to be able to create an intelligent and working network without having to understand the code inside the components programmed by other people.

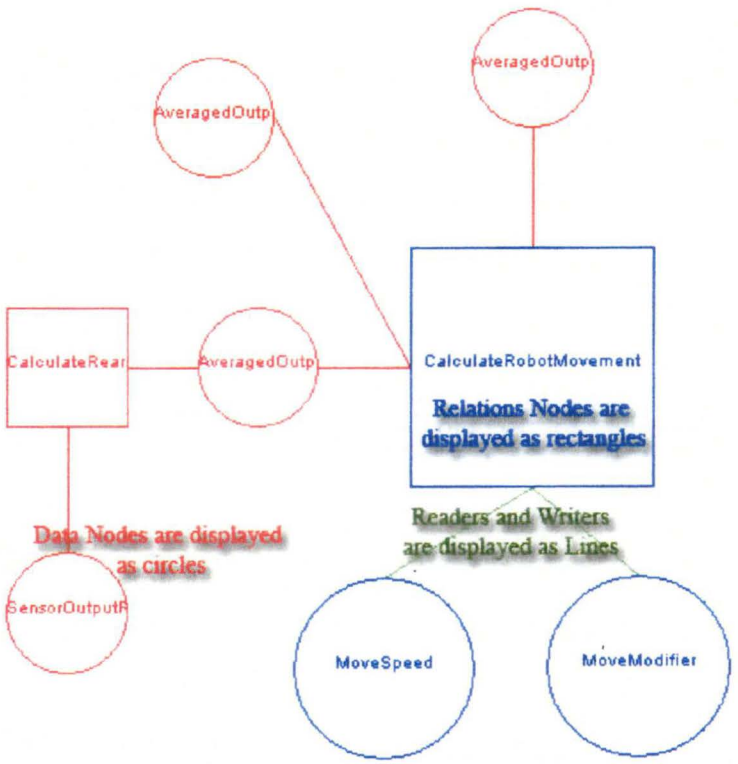
During our researches about what was needed by users, we found that:

- It must be possible to have a graphic overview of the FDNetwork created;
- To a certain extend, FDNetwork edition must be possible in a graphical way;
- Selecting entities inside this network in order to receive some general information about them is important to have a quick idea of what particular parts of the FDNetwork do;
- Having some abilities at disposal allowing a user to easily recognize FDNetworks parts would allow greater understanding in a shorter period of time;
- Dynamic capabilities have to be easily accessible.

The following pages will explain to you how each point has received an answer and how this answer has been implemented.

The graphic overview:

A complete graphic representation system has been implemented in order to make it possible for a user to visualize the FDNetwork he is editing. This representation displays every FDNetwork entity with a particular shape, upon which user can perform some tasks.



Nodes can be dragged and dropped anywhere in the Graphic Panel in order to allow the user to organize his Graphic Network as he wishes to. As they always connect Nodes, Connections cannot be moved by themselves but will, of course, follow the Nodes when they are moved.

This graphic ordering is saved at the same time as the FDNetwork edited itself (While FDNetworks will be saved in files bearing the .FDN extensions, Graphic Networks will be saved in .FDG files). Upon loading of an existing FDNetwork, the related Graphic Representation will be loaded too, if it exists. If not, all the entities will be drawn at a predefined place (the upper left corner of the Graphic Panel).

Graphically-driven FDNetwork edition:

Graphical representations of FDNetworks have advantages and drawbacks. It allows having a general view of the FDNetwork edited and gives an idea of what the FDNetworks do quite easily. But, on the other hand, it doesn't show complex information, or the way the information is displayed simply doesn't allow the user to see specific constructions. This is why we decided to limit Graphic's representation's editing capabilities.

But, even if some edition functions are not allowed due to their inadequacy with the Graphic Representation doesn't mean that no edition is possible at all.

While in graphic edition mode, it is possible to:

- Add Nodes and Connections through the use of the Add Entities Action Panel.
- Edit Nodes by Double-clicking on them directly on the Graphic Representation.
- To use the Merge and Replace functions by giving the same name to two Nodes of the same kind.

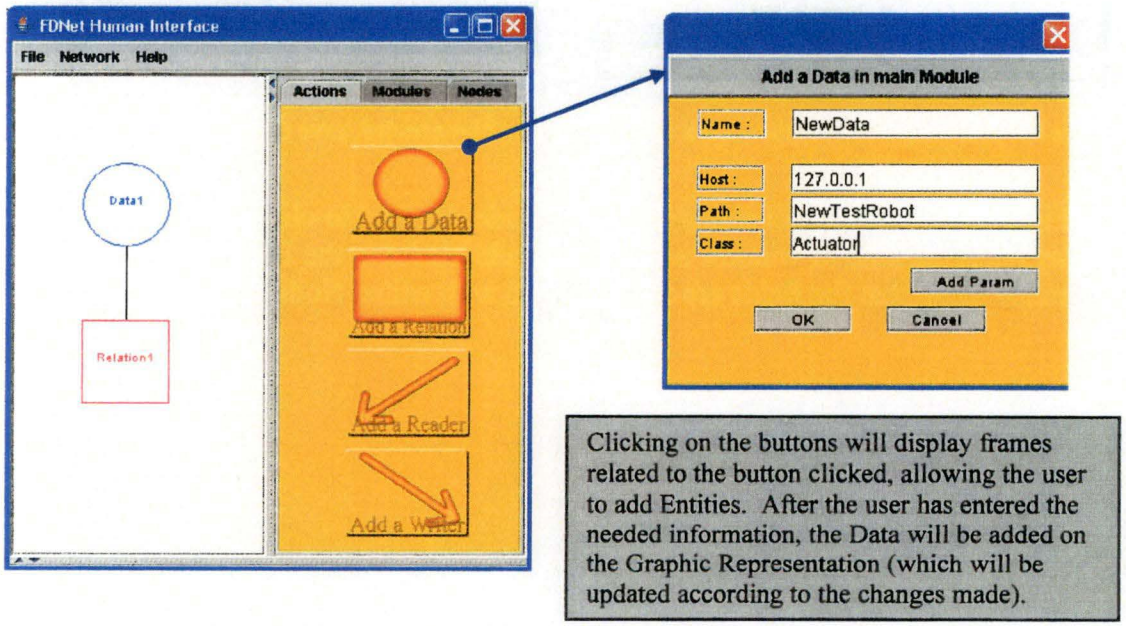


Figure 35: Human Interface Graphically-driven FDNetwork edition

The Human Interface also allows you, through the use of the Module Selection Action Panel, to select the Module in which you will add the Entities.

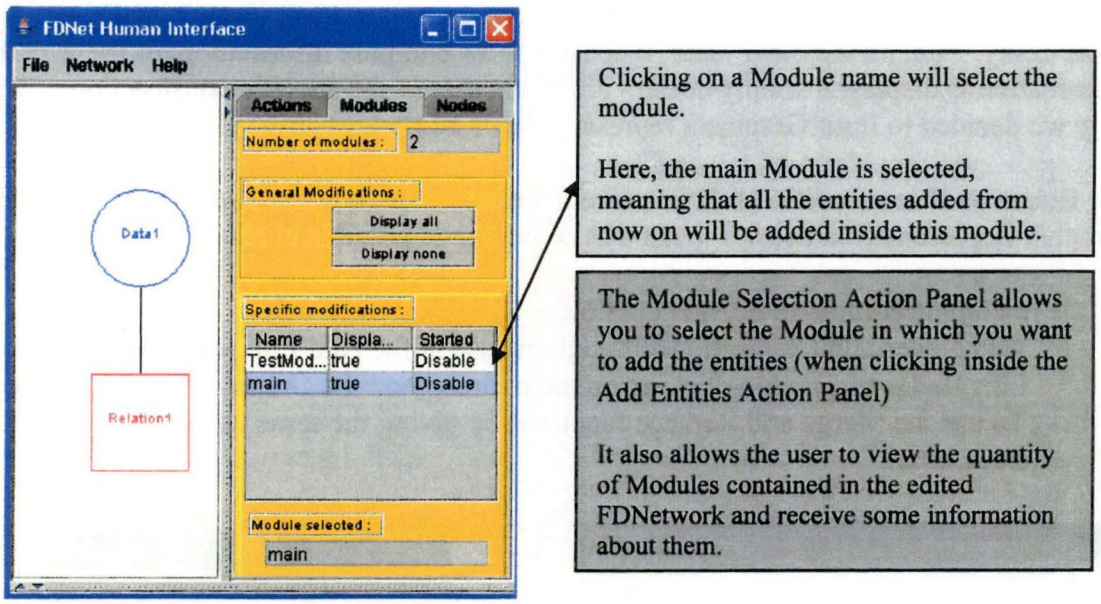


Figure 36: Selecting the Module where the network entities created have to be added into.

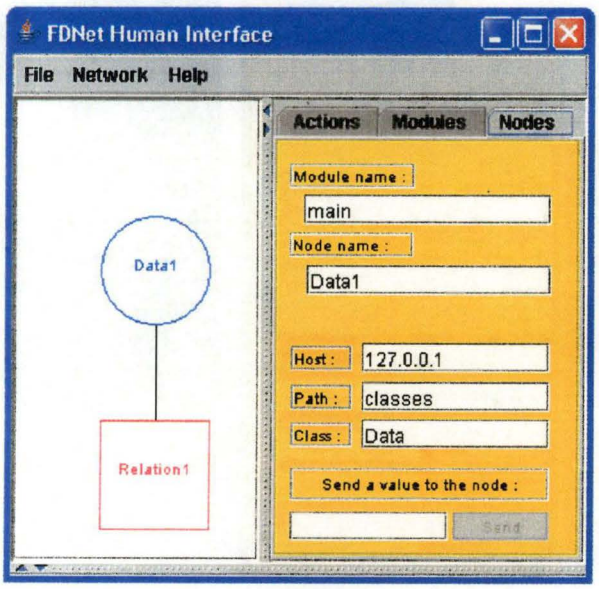
It is also possible to edit already existing Nodes by double-clicking on them. This will display the same kind of window as when you add Nodes in the FDNetwork, the only difference being that the fields are already filled with edited Node's information.

Display Entities information:

Human Interface's Graphic Display also allows the user to receive some information about Nodes[1]. Upon clicking on Nodes inside the Graphic Representation, the Node Information Action Panel will be updated with general information coming from this Node.

In the current version of the Human Interface, this panel allows to:

- View Node's name and the Module in which it resides;
- View where Node's implementation class is stored;
- Send a new value to this Node, while the FDNetwork is working (i.e. in real-time)



The user has clicked on "Data1" Data Node, thus updating the Node Information Action Panel with its information.

Nearly all information about Nodes can be seen inside this Panel.

"Recognizing" abilities:

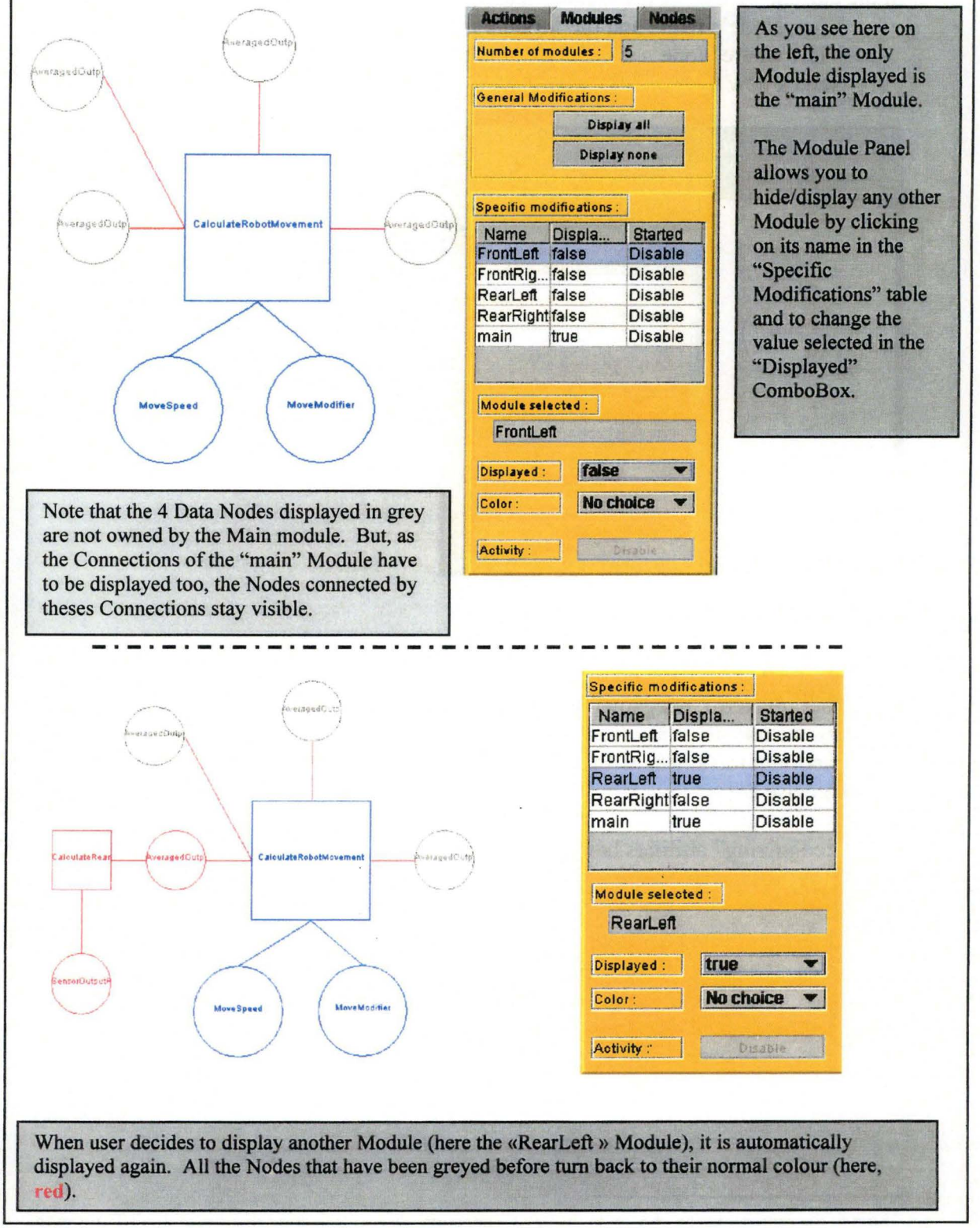
Under this term, we gather all abilities implemented in the Graphically-based display allowing the user to easily recognize parts of his FDNetworks while he is editing it. This is very important too since, as FDNetworks can become very big, it can become very difficult to find a precise entity.

The "recognizing" abilities have been implemented at two levels: the Module level and the Entity level.

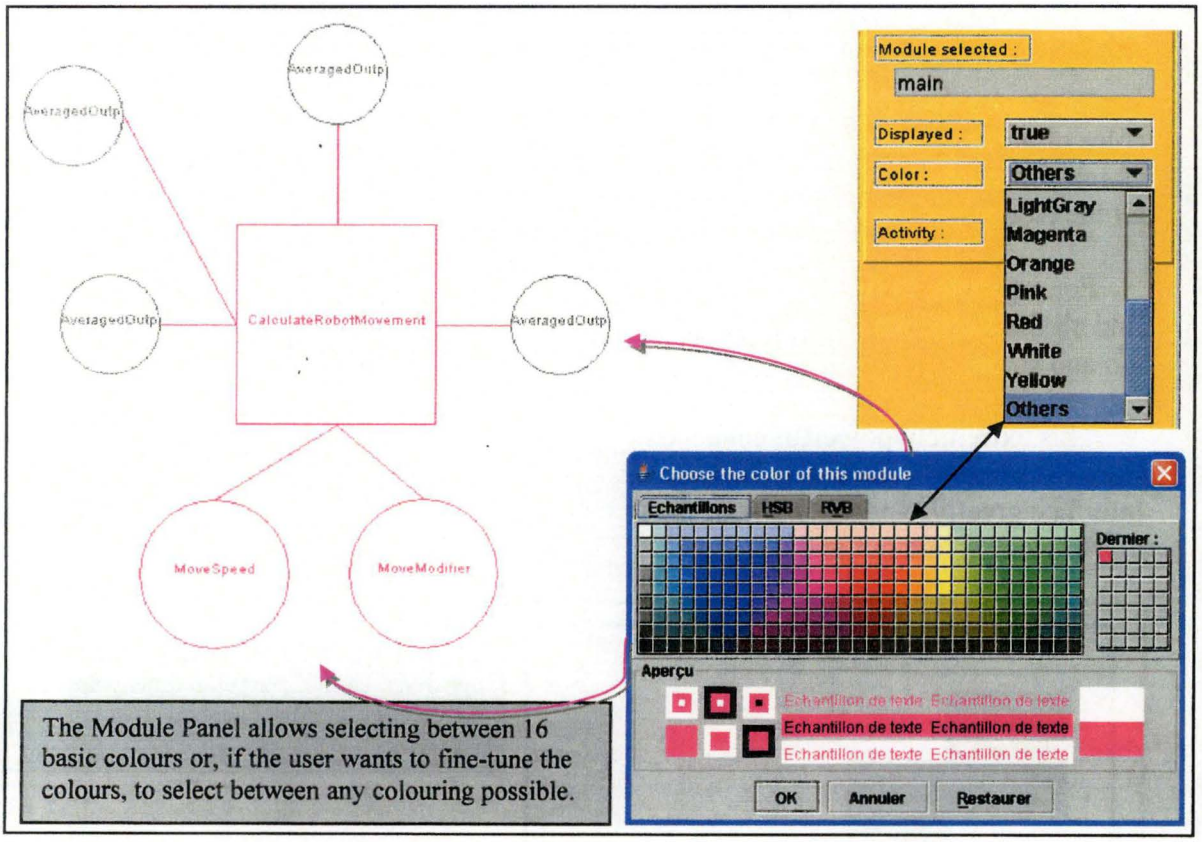
[1] Due to the representation chosen to display Connections (lines), selecting them is quite difficult in the Graphic Representation. This is the reason why it is currently not possible to receive any information about Connections in the Graphic Representation.

1: Enhancements at Module level:

It is possible to hide and display any module in one mouse click. Doing this, the user can focus on what is interesting him, instead of having the screen filled with entities which have nothing to do with the current work the user is achieving.



It is also possible to change the color of a whole Module in the same Module Panel. Just by selecting the color wanted, all the entities contained in the Module will be displayed in that color.



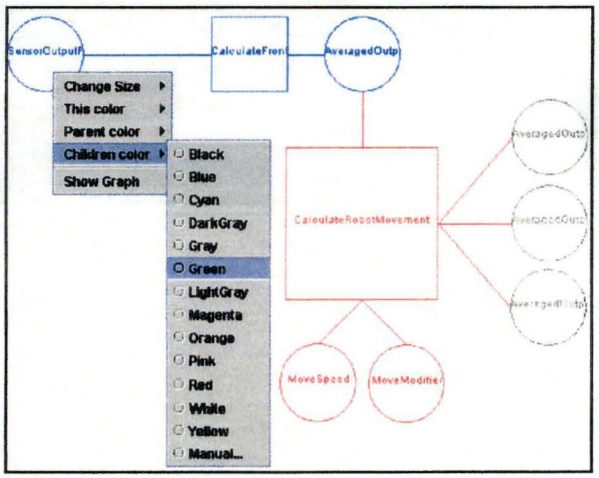
2: enhancements at the Entity level:

Entities themselves have been granted with some very powerful visual enhancements. As said before, it is possible to change the size of the graphic representation of a node. It is also possible to change the color of any Node, using the same system as for the Modules.

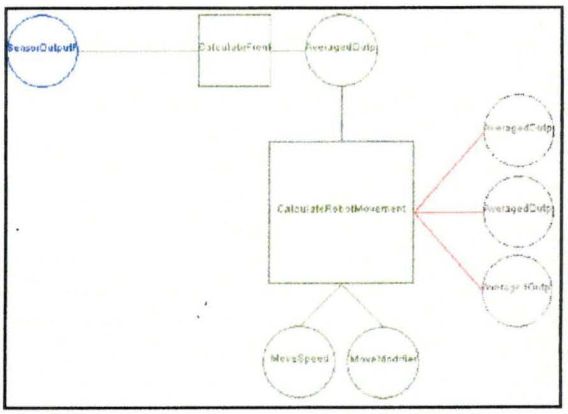
But, more than this, it is possible, for any Node in the edited FDNetwork, to change the color of all its parents and/or all its children. The parents (ancestors) are all the Nodes and the Connections whose work have an impact of the Node selected. The children (successors) are all the Nodes and the Connections depending on the Node selected.

For example, to find the children of a Data Node, we select all its Reader Connections to find all the Relations reading this Data. All the Writers of theses Relations will be selected too and so on.

This extremely powerful enhancement allows the user to know where any Node comes from or where it is used very simply. This capability enhances the utility of the Human Interface greatly. Indeed, most of the time, the user wants to have a quick idea of the way its FDNetwork is laid out and this is particularly the aim of this enhancement.



Upon right-clicking on the Data Node at the upper-left corner, a scrolling menu appeared and the user selected the "change children colour" submenu.



As you can see on the right, all the children of the selected node have been coloured in the selected colour (here green)

Accessibility of Dynamic Capabilities:

The last main requirement to fulfill was to allow FDNetworks edition at work-time. As said while explaining Human Interface's Dynamic Capabilities, the user of an FDNetwork has to be able to edit it even when the network is loaded in memory and working.

In fact, this requirement has been implemented in a very smooth way since, even when the edited FDNetwork is loaded in memory, nothing changes for the user: Every action available at creation time is still available at work time.

The Human Interface is aware of edited network's state. When it is loaded in memory and working, all user orders will first be sent to the FDNetwork before being applied inside the Graphic representation. Should an order be refused by the working FDNetwork, the Interface would display an error message telling that the update cannot be done. This way, Human Interface's Network and working Network always stay consistent with each other.

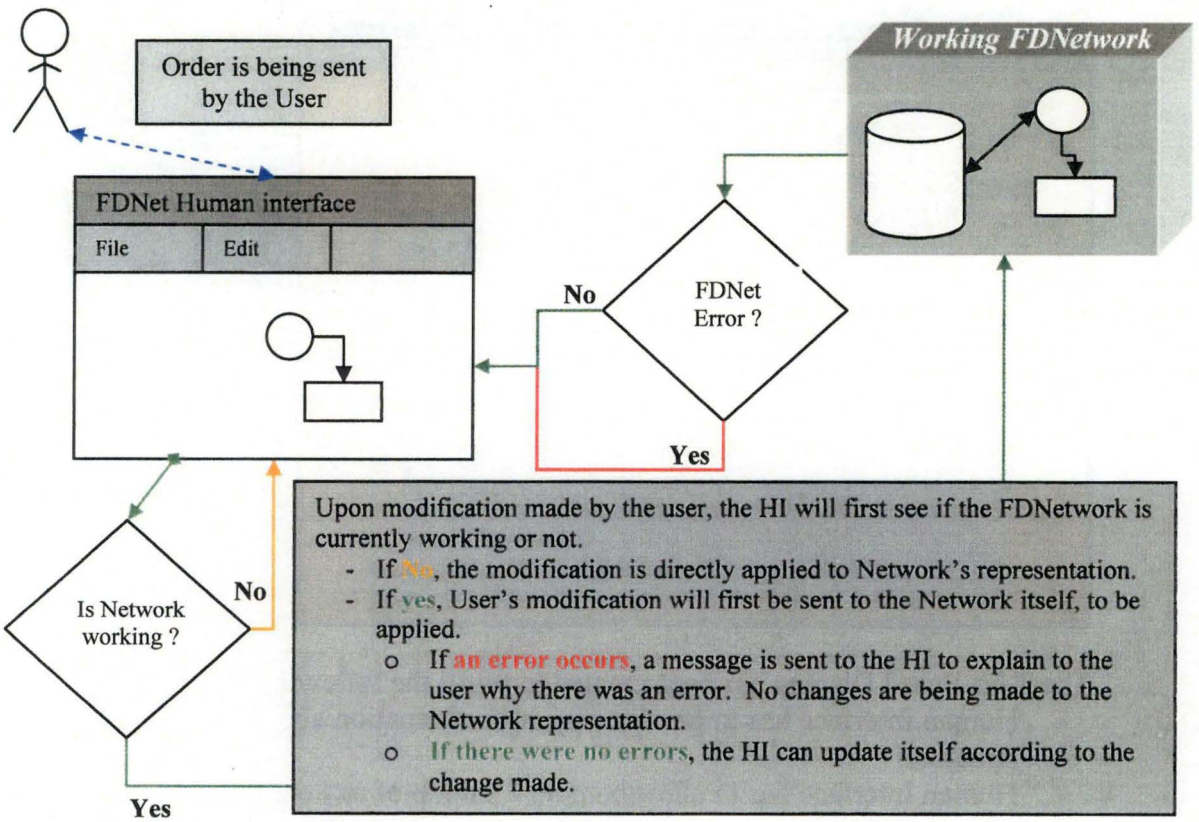
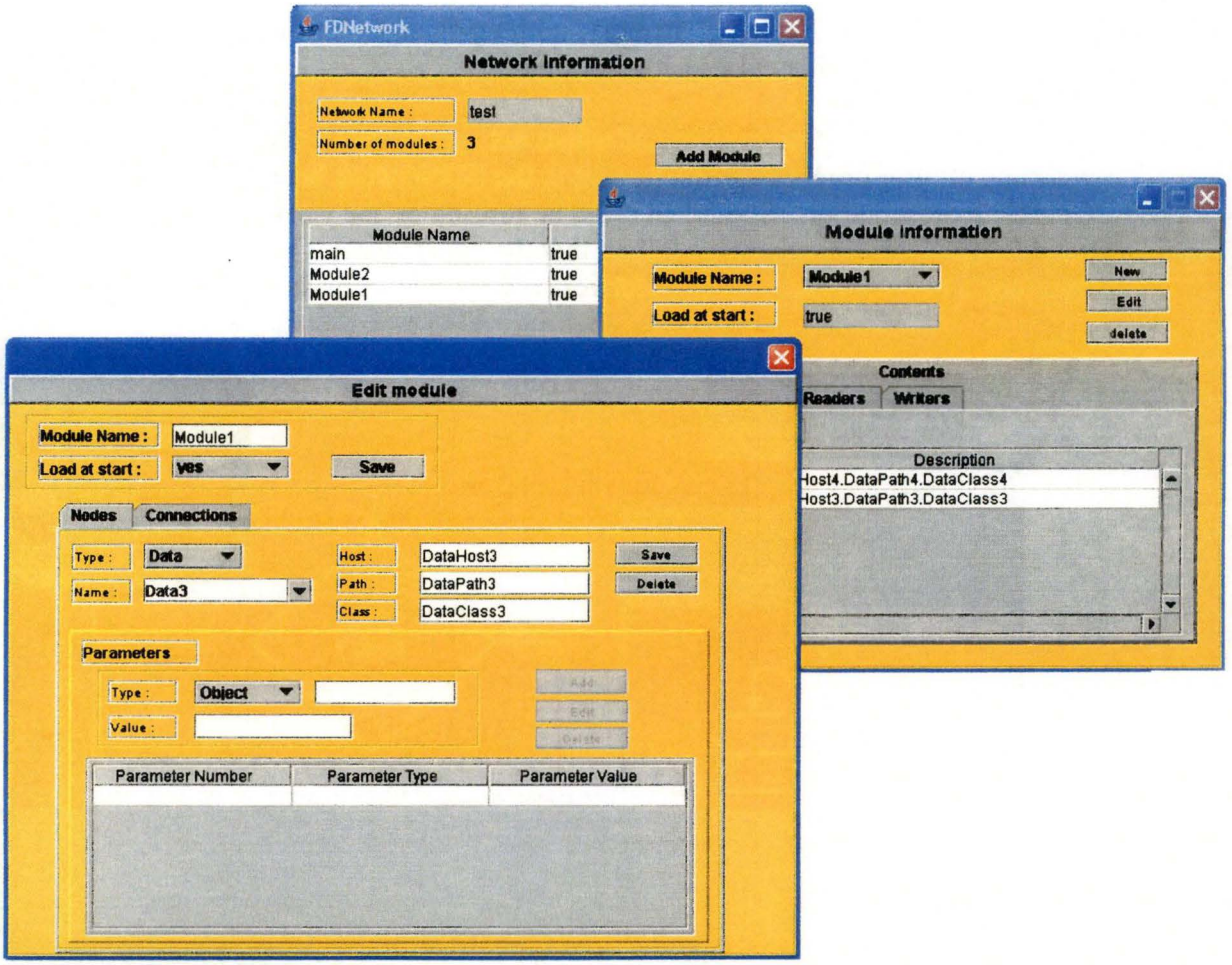


Figure 37: Human Interface's reaction upon user order.

B. The Text-Based Display:

While the first part of the Human Interface allows the user to create and edit his FDNetworks by working with their graphic representation, this part will allow him to do it through the use of forms.



The Text-Based Display has been created to fulfill the following requirements:

- Human Interface has to provide numeric information about edited FDNetwork's structure;
- Human Interface has to allow complete edition of any entity contained inside the FDNetworks;
- Human Interface is also simply another edition way provided to the User.

Following will be the presentation of each of the abilities created in order to respond to theses requirements.

FDNetworks in numbers:

The different frames created will give the user numeric information about the FDNetworks edited. How many Modules are contained in the FDNetwork? How many Data Nodes are there in the "main" Module? And so on.

This kind of information can be interesting for users wanting to see their networks from another point of view. For example, possessing this information will allow a user to know if some of the Modules he created are too big (which will cause them to load more slowly) or too tiny (It would then be interesting to merge some of them to reduce their overall quantity in the FDNetwork).

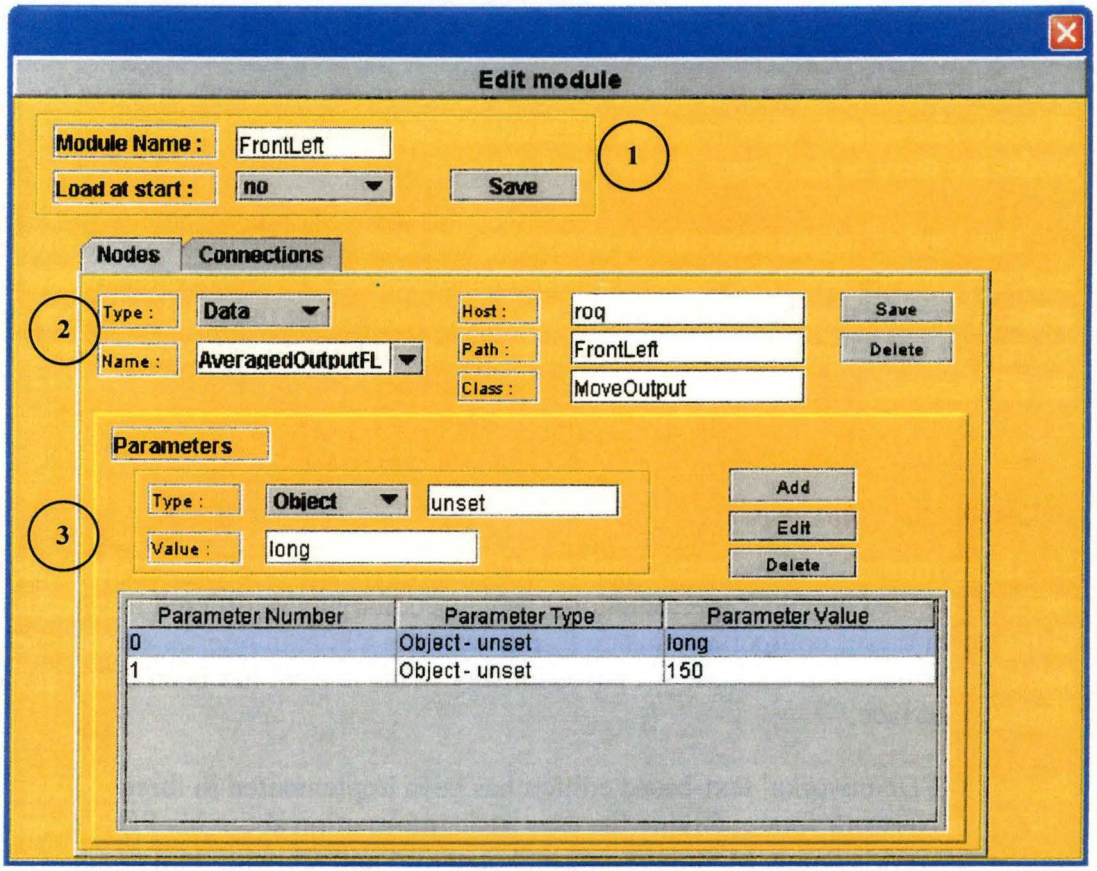
FDNetworks complete edition:

Using frames to allow users to edit their FDNetworks brings power impossible to obtain while using the Graphic Representation. Tables, combo boxes, ordered lists and so on… Using all of theses components inside forms will help a human interface creator to display lots of information in a simple yet powerful way. This is what has been done on FDNet's Human Interface.

Basically, FDNetworks' text-based edition has been implemented in three frames:

- The Network frame: Giving the user basic information about his FDNetwork (Network's name, Modules name and quantity) and allowing him to add Modules inside this FDNetwork.
- The Basic Module frame: Giving general information about each Module (Name, quantity and names of Nodes and Connections) and allowing the user to edit and delete each of them.
- The Advanced Module frame: This frame is the most important one of the three. While the two first frames focus on giving structural information about FDNetworks, this frame will allow the user to work on the Entity level.

Instead of explaining how each function of each frame works, we will directly focus on the most important frame, the Advanced Module frame, and explain the idea around its creation.



This frame allows editing anything about a Module, from its name to the parameter of one of its Data Node. It is divided in three parts:

1. Module information: Allows changing Module's name and Load Modifier.
2. Contained Entities Information: Allows creation and edition of Nodes and Connections.
3. Contained Entities Parameters Information (only for Nodes): Allows creation and edition of Nodes parameters.

This frame helps the user a lot in his edition work since it carries out all the problems that could be encountered while editing FDNetworks. Here below are explained some examples about what this frame is capable of doing:

- All frame's components (buttons, text fields, tables,…) are evaluated in real-time to be sure their state is consistent. For example, if the user types in the name of an already existing Data Node while trying to create a new one, this Data's information (Host, path, class and all parameters) will be automatically loaded and "Save & Delete" buttons will change their behavior in consequence.
- While creating Connections, only existing Nodes will be selectable to connect. This ensures the user will not create Connections connecting nothing (which leads to errors).
- Upon deletion of Nodes, all the related Connections are deleted too.

All the relations between FDNetworks entities are handled by the NetworkInfo architecture. This means that the NetworkInfo architecture handles all the constraints applying to the FDNetworks (like, for example "Connections can only exist if their two connected Nodes exist too").

As said just before, the frame will change its component states automatically, upon user changes. This is done to ensure the user will benefit of the best experience possible but how did we do this?

In fact, this was one of the main problems encountered while creating the Human Interface. This frame contained so many components (27 actually!) that programming all the interactions between all of them was nearly impossible. To cope with this problem, we had to find another way to program Human Interfaces. We called it "the SpeedyDesign technique" and it was a total success.

X. The SpeedyDesign technique:

Human Interface programming is quite simple to understand. A frame only consists of other components such as text fields, buttons combo boxes and so on. All theses components can be interacted with through the use of events, which are called by the system.

For example, in Java, to react to a click on a button, you will just have to tell to the button itself which method call upon mouse click and put all the needed code inside this method. This way of doing things is really simple to understand although quite powerful. It seems to allow the programmer to maintain its code easily since all the code related to clicking on the button will be found in the same method.

But the fact is that, programming like this will lead to incoherence in user code, which will not be maintainable anymore. The problem is that, by programming this way, components' state is modified by other components. As more and more components are added to the frame and as more and more interactions have to be created, this ultimately leads to impossible to predict component behavior.

While programming the Advanced Module frame shown here above, the point of collapse, as we called it, was reached and forced us to stop development to find a way to correct the problem. The point of collapse is a point where it is impossible to add a new component to your frame. This component will lead to so many changes in the frame that it will become too hard to code.

After analyzing the problem we found out that its base lies in the fact that a component can have its state modified by lots of other components, leading to the spreading of code related to it.

The key to having a maintainable code is then to have it written in a single place. This is the base of the SpeedyDesign technique. Since a component will always have to be able to modify its own state, the only way to have only one entry point is to force a component state to be modifiable only by itself.

Although it might seem very restrictive, this constraint is not a big deal. In fact, instead of directly changing other components' state, a component will simply ask them to update themselves by telling them that he has been modified.

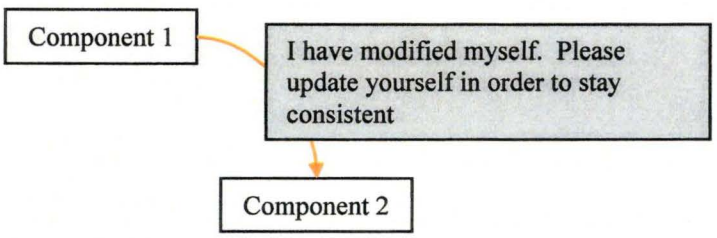


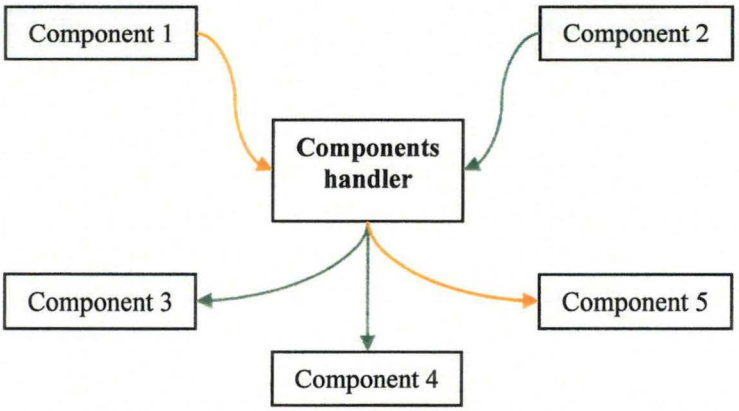
Figure 38: SpeedyDesign technique – Asking components to update themselves

And this solves the problem! There is now only one place where the programmer will have to code in order to update a component's state. If the state of one component depends on the state of another one, it will just have to look how this component is behaving in order to know how to be updated.

As we found out that this technique was quite useful, we tried to see if it was not possible to further enhance our way of coding to make it even better. And it was possible.

In fact, there is still one problem here. Every component has to know every other component in order to ask them to update themselves if necessary. It means that, each time a component is added to the frame, every other component have to be reviewed in order to ensure consistency.

To correct this problem, we introduced the component handler. The component handler's aim is to centralize every update calls in one single point. When a component updates itself, instead of asking to every other component to update themselves, it will just tell to the component handler that it has updated itself. The component handler will transfer all the update calls himself.



When component 1 is updated (user click), it tells it to the component handler which knows that the component 5 is related to component 1 and will then ask it to update itself. The same system can be applied to every component.

This way, all the interactions between components are handled in one single place. Upon addition of a new component, a single modification in the component handler will have to be done.

Note that, if the state of the new component is important for other components too, theses components will have to modify their update method too. But modifications will have to be done in one method only.

Figure 39: SpeedyDesign technique – Role of the components handler

While programming the Advanced Module frame and trying to find all the relations between all the components in order to program the component handler correctly, we found that there were still more enhancement to perform.

This enhancement concerned the quantity of times that the "update" methods are called by the component handler. Up until now, if a component has to be updated by n components, its update method will not be called n times but a number of times between n and n! times. This is due to the fact that, at its current state, the SpeedyDesign works with "ancestors – descendant" relations where it should only work with "parent – child" ones.

To understand this enhancement, the following definitions have to be introduced:

Ancestor: A component is said to be the ancestor of another one if a modification in his state will have impact on the state of the other component.

Descendant: A component is said to be the descendant of another one its state can be updated through updates to the other component.

Parent: A component is said to be the parent of another one if knowing its state is essential for the computation of this other component.

Child: A component is said to be the child of another one if knowing the state of this other component is essential for the computation of the state of this component.

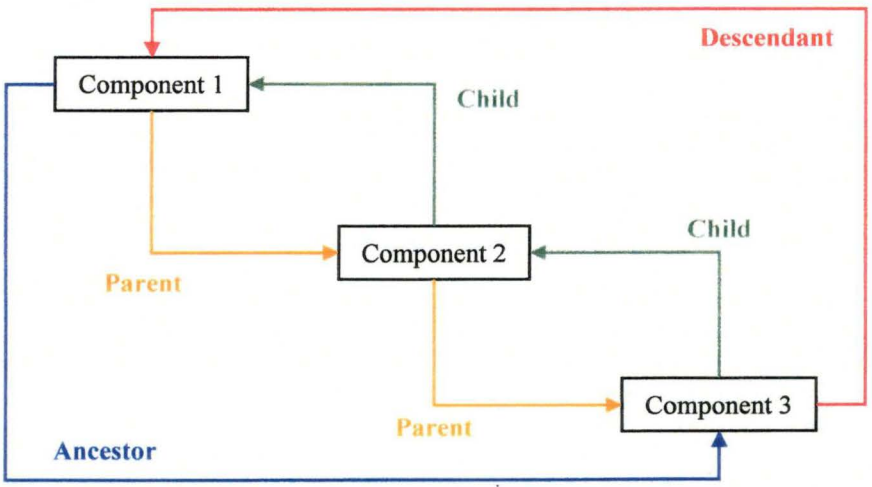


Figure 40: SpeedyDesign technique – The existence of hierarchy between components

As you see in the Figure 40, "Component 1" is the parent of "Component 2" and the ancestor of "component 3". Upon update of the state of "Component 1" the component handler will call the update method of "Component 2" and the update of "Component 3". But the problem is that, when "Component 2" will be updated, "Component 3"'s update method will be called again.

The resolution of this problem lies in the definitions themselves: Upon update of a component, the component handler has to call the update methods of its children only and not is descendant.

To find descendants and the children is not difficult. It only requires the programmer to create the functional dependencies graph for the components and to sort them in levels by putting all the components that have no children in the same level and start again with the graph in which the components put in level have been removed.

Note that it is not possible to have a loop (a Component which is, at the same time, the ancestor and the descendant of another one). Indeed, Human Interfaces always follow leveled information patterns. And in case the programmer has loops in his interface, rearranging his frame in order to remove the loop would still be possible.

The scheme below summarizes the SpeedyDesign programming technique:

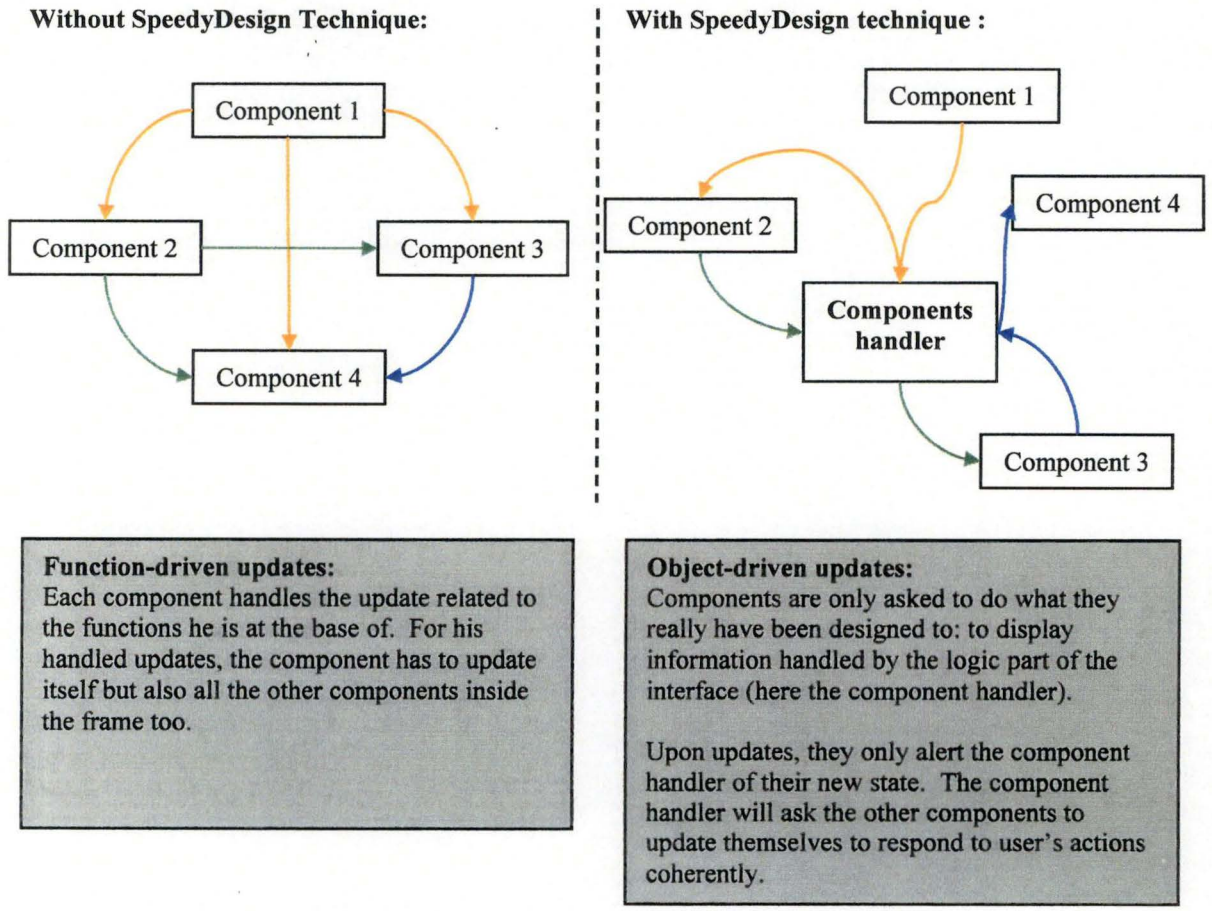


Figure 41: A summary of the SpeedyDesign programming technique

XI. Human Interface's current limitations:

The Human Interface we programmed, although already usable, is still a work in progress. To program a project as big as this one in the time imparted is not possible. All the abilities presented here in the thesis are implemented and already working but they could be enhanced a lot.

FDNet architecture itself being still a work in progress, FDNetworks are still complicated to create. Due to this fact, the Human Interface has not really been used on real FDNetworks and a lot of testing still has to be done in order to make it fully respond to FDNet researchers.

This fact set apart, we would like here to introduce you to the enhancements that we think would be interesting to work on. Theses enhancements would allow coming nearer to the utility and usability factors that we said would be achieved through Human Interface's creation.

When all the enhancements listed here below and all the testing related to them will be done, the Human Interface will truly become a powerful addition to FDNet project.

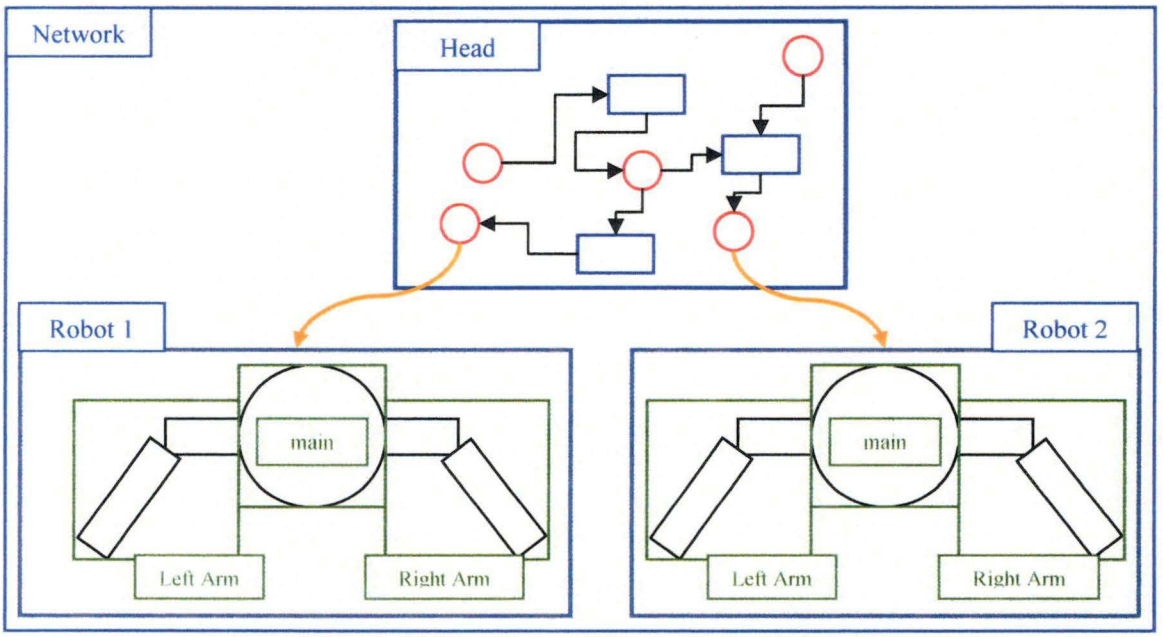
A. Enhancements concerning the NetworkInfo structure:

Giving a definite type to parameters:

In FDNet's current specification, Nodes parameters don't have a definite type. Any information is stored in "string" format and only the Node containing the parameters knows what he has to do with them. If parameters were directly saved with a type, we could check it in the Human Interface to ensure that created FDNetworks are more reliable.

Multi-leveled Module subdivision:

Concerning Modules, only one level is allowed. It is not possible to have Modules contained in other Modules. We think that Modules will play a very important part later in FDNet development and are sure that this ability - having Modules contained in other ones – would be extremely useful. It would allow a more complete FDNetwork subdivision that would benefit to everybody. The scheme below introduces you an example of what theses sub-modules could allow if they were implemented:



Through the use of more than one Module level, it would really be possible to achieve one of FDNet's real aim easily: To have a single "head" handling a pool of robots all working together to rescue people.

Using more than one Module level allows the user to divide his Network a lot more correctly. In the example above, you can see that the two robots are included in one module but, at the same time, each robot is divided in more parts. Through the use of Human Interface's editing abilities, this division would allow an user to focus more easily on the work he wants to achieve.

Figure 42: Introduction to multi-levels Module subdivision

B. Enhancements concerning the FDNetworks' Graphical Representation:

Connection Entities representation:

For the time being, through the use of the Graphical Representation, it is only possible to interact with Nodes. Due to current Connection representation (a simple line), mouse interactions are not possible. Enhancing Connections' representation (An arrow with the name of the connection in the middle of this arrow for example) in order to allow the user to receive more information about them and to edit them would increase Graphical Representation's utility.

Intelligent placement:

When an already existing FDNetwork is loaded in the Human Interface for the first time, all the entities are displayed one above the other in the upper left corner of the Graphical Representation. It is clear that a better placement system has to be implemented.

This problem also appears each time new Entities are added to FDNetworks. Newly added Network Components are not placed in an intelligent way (they are placed at the top left until the user drags them to another place). Creating an intelligent system placing Nodes in consistent places based upon the Module they exist in and the Connections they possess would allow more easiness of use.

Of course, this is not an easy-to-implement enhancement and it is the reason why it has not been done. But, if the Human Interface is to be used in real situations, this will have to be done.

C. Enhancements concerning Human Interface's integrity:

Representation system harmonization:

FDNet's Human Interface could have its overall integrity enhanced. Indeed, the two different representation systems (Graphical Representation and Text-based Representation) do not provide the same functionalities even if they are based on the same information. Of course, there will always be some differences between the two representation systems but it is clear that in the current Human Interface, some differences are not justified and abilities could be implemented in the same way. Users would have it easier to master the Human Interface and would be able to do better work in less time.

Errors and exceptions handling:

Even if all the mechanisms needed already allow it, handling errors and exceptions in a better way has to be done. The Human Interface contains a Status Bar showing important information to the user. But this Status Bar has been neglected until now and must be improved to handle more errors and exceptions.

Interface Revisions:

Until now, the interface is still in beta version. It is clear that some options are missing, some of them are not useful, bugs have to be found and corrected and so on. This kind of errors/enhancements can only be found with extensive use.

XII. Conclusion:

FDNet is a complex system that one cannot easily understand. It requires the user the have some knowledge of robots, of cognitive science and of computer science just to grasp its concepts. Even when concepts are understood, using FDNet to produce something really useful is quite complicated.

When this kind of problem arises, there are usually two main ways to correct them: The first one is just to make the system simpler, by removing parts that are not useful and by using tricks to simplify the important ones.

The second one has to be used for systems that are known to be complex no matter how you use them. For theses systems, the technique is to create adaptive tools that will help users to apprehend system's complexity over time while providing more and more complex abilities as users improve.

The Human Interface can be seen as one tool of this kind. It will, at the beginning, give the user a straightforward and easy to use graphical representation of the networks he tries to create, while at the same time, provide functions to make Networks' edition simpler.

At a second time, it will offer more advanced functions and another edition mode: the Text-based edition mode. This one will give users more power over the FDNetworks they are creating through providing them with complete sets of edition abilities.

Even if our Human Interface is far from being perfect, it is already usable to a point that it can really enhance FDNet users' work. All FDNet computer aspects have been hidden and it is now possible to create FDNetworks while focusing only on the work to be done.

Of course, a lot of enhancements will have to be done in the future if FDNet is to be used but Human Interface's current implementation already shows the power it can unleash. We are confident that this part of the project is a milestone for FDNet users and that FDNet's success will depend on tools like this one.

Chapter 6: Conclusions

I. Conclusion about FDNet:

The aim of our work was to enhance FDNet's Utility and Usability through the creation of a Logger and a Human Interface.

In term of Utility, the Logger allows saving the state of the working FDNetworks at different moments, thus allowing the users to analyze them in order to enhance their understanding. This is the reason why the Logger shares this information between the different Viewers. The Human Interface introduced the concept of Module, allowing users to create FDNetworks divided in such a way that they become a lot more powerful.

But it is in term of Usability that FDNet has been improved the most. Before our arrival, I.R.S.I researchers had focused their work on developing a network architecture allowing them to control a pool of rescue robots. But, even if their work was useful concerning the problem to resolve, it was not very usable.

The Human Interface is the incarnation of FDNet usability. Indeed, it allows to easily create, edit, update and delete a network configuration. Also, starting and stopping an FDNetwork can be done in just one or two mouse clicks. Analyzing the network evolution is done by using the Viewers[1] integrated to the Human Interface. Through the use of the Human Interface, FDNet can be used by people coming from diverse discipline (computer science, electronic engineering …).

Nevertheless, the Human Interface would not be usable if the Logger was not providing it with working FDNetworks' information. It has been done in such a way that this information can nearly be given in real-time to numbers of different clients (Human Interfaces, viewers…). The Logger was also granted a Human Interface allowing its administrator to configure it and start it in an easy way.

All this work done allows us to say that we think to have fulfilled our mission of making FDNet a lot more useful and usable and hope that this work will influence positively the project in the future.

[1] Refer to Mr Nicolas Lambot's work to have more information on this subject.

II. Personal conclusion:

At first it seemed very difficult to work along with the Japanese researchers at the International Rescue System Institute. To have a discussion about the work to be done was quite complicated because of the differences in the languages spoken. To have an explanation about FDNet itself - the base of our work - was not even possible.

But we tried hard and, by using retro engineering methods on FDNet's source code, we could understand more and more about it everyday. By that time, our relations with the Japanese people had become far better since the fear of speaking English and the shyness we were all facing at the beginning had disappeared.

Having to read on all the source code in order to understand FDNet, although it was an extremely difficult task, appeared to be one of the best choices we made. Indeed, having all the code in our minds, we could think of the best architecture possible for our own work: The Logger and the Human Interface.

More than just producing a good Human Interface and a good Logger, we could introduce very interesting enhancements to FDNet itself. Enhancements that will, we hope, be continued by other researchers in the years to come.

But it would have been impossible to think about theses enhancements if FDNet was not an interesting project to work on at the beginning. We are now sure that it holds a great potential in term of helping rescuers to saving lives and are honored to have had the chance to participate to its creation, as little as our work may seem to be in the future.

Youssef and Jérôme

Bibliography

Fumio Ozaki, "Open Robot Controller Architecture (ORCA)", http://staff.aist.go.jp/t.kotoku/fyi/AIM2003.html#TANIE (20 July 2003) (Date of access May 2004)

JARA, "Specification of Orin", http://www.jara.jp/E_ORiN/En_ORiN.htm (1999)(Date of access May 2004)

Lambot Nicolas, "FDNet: Enhancing Human Interface with Dynamic Capabilities" 2003

Masahiro Fujia and Koji Kageyama, "An Open Architecture for Robot Entertainment", *Proceedings of the First International Conference on Autonomous Agents*, ACM Press, 1997

Toshiba Corporation, "Toshiba to Bring its New "ApriAlpha" Concept Model Robot to ROBODEX 2003", http://www.toshiba.co.jp/about/press/2003_03/pr2001.htm (20 March 2003)(Date of access May 2004)

Toshihiro Inukai, "ORiN: A common object model for robotic systems", http://staff.aist.go.jp/t.kotoku/fyi/AIM2003.html#TANIE (20 July 2003) (Date of access May 2004)

Yosihisa Koji, "Flat-distributed network architecture (FDNet) for rescue robots", 2002

Appendices

Appendix 1: Retro engineering on FDNet

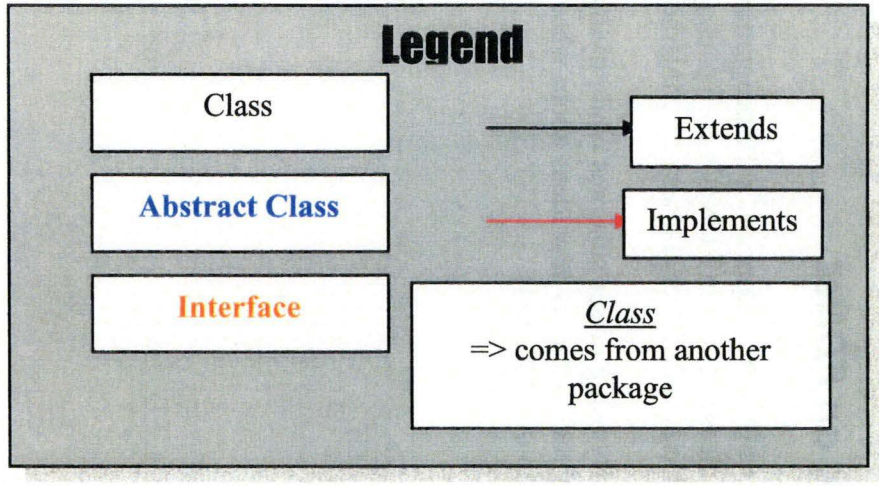
This annex consists of the schemes created while retro engineering the programming work done on FDNet. The schemes represent all the classes found in FDNet's core packages. Doing this kind of work allowed us to obtain a general view of FDNet classes static interactions.

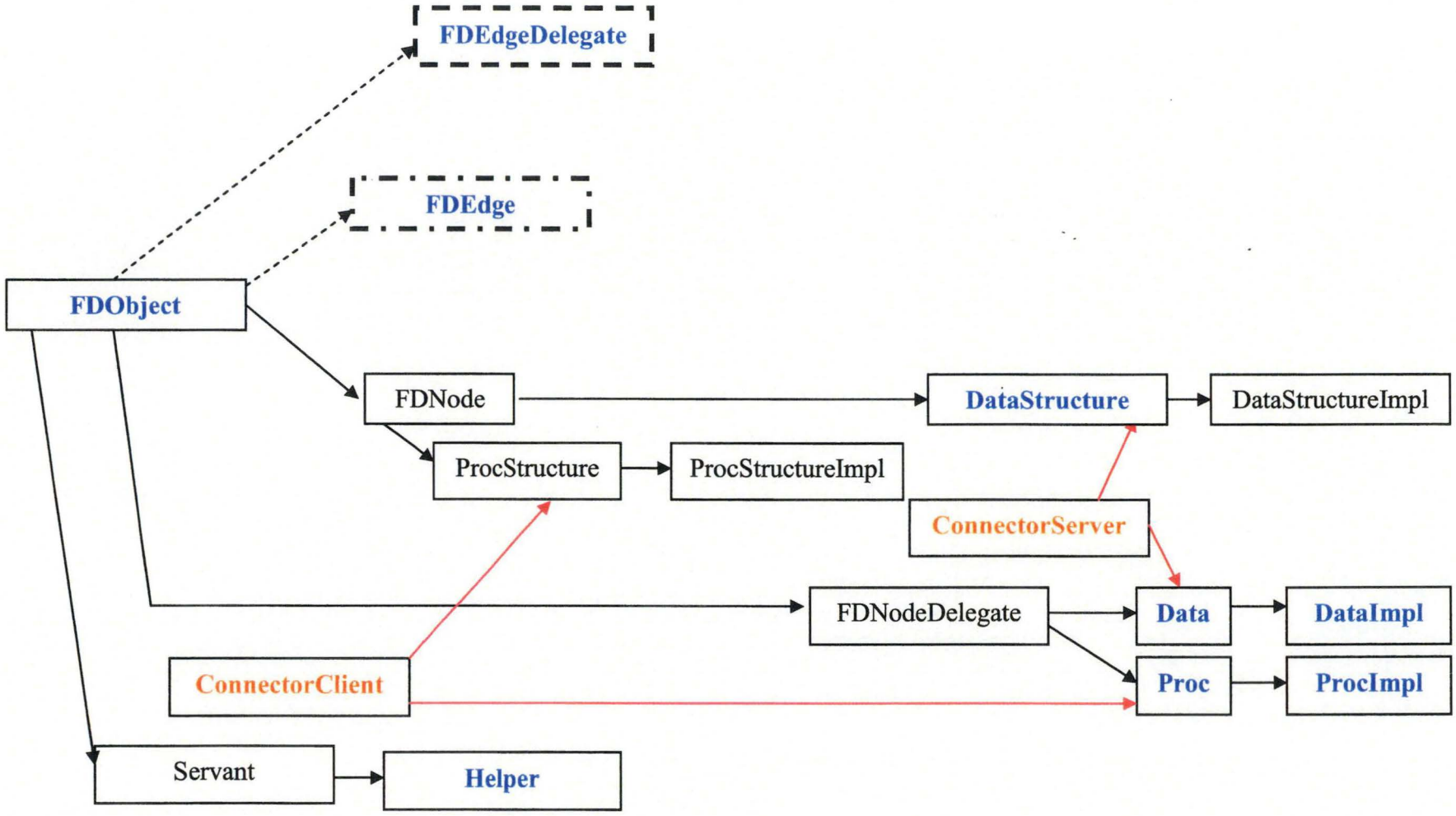
It also gives an idea of how FDNet works. By using theses schemes while trying to understand the code written by the I.R.S.I researchers gave us new ideas about what the code was doing and where to look to find an answer to our questions.

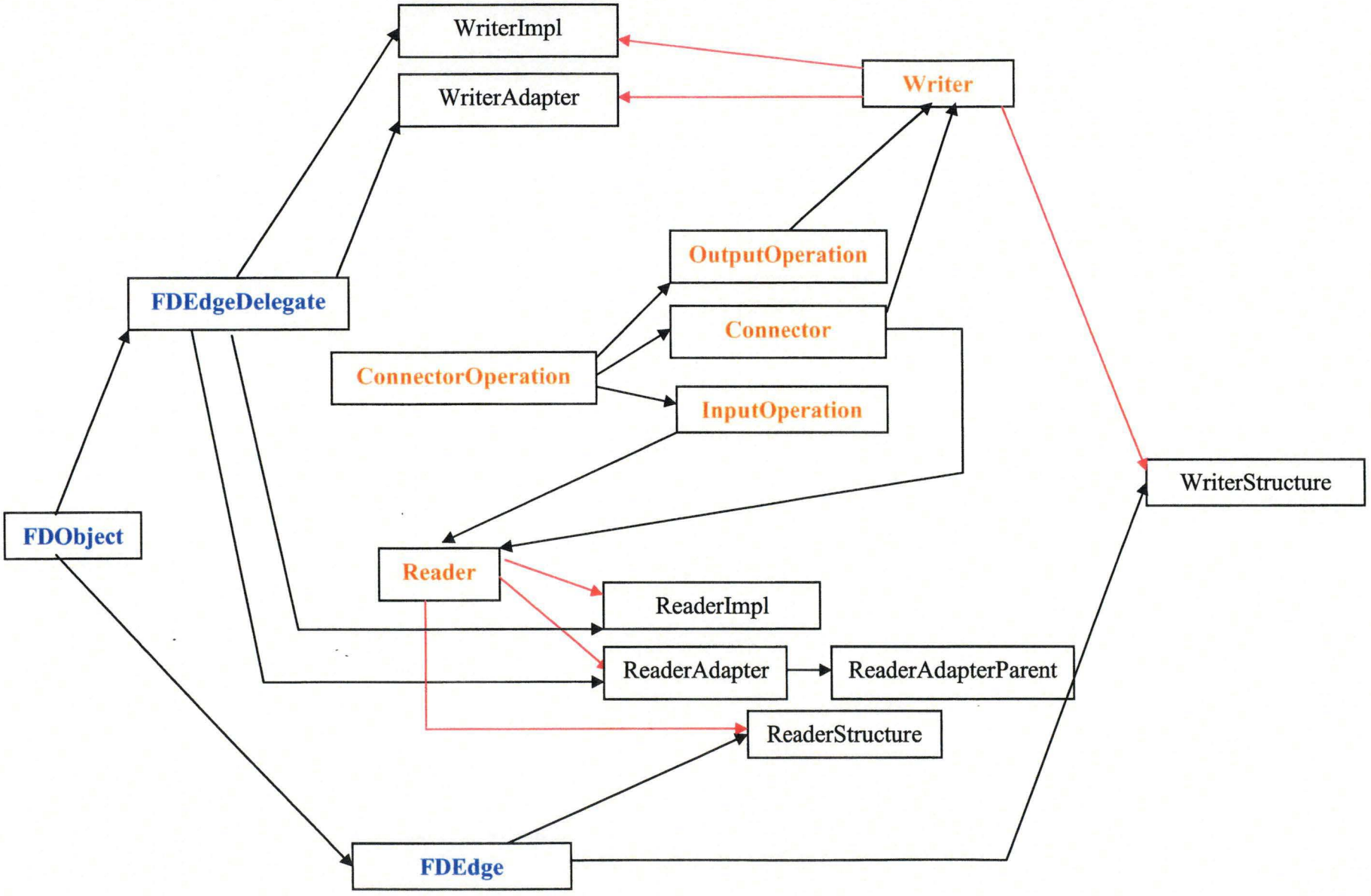
This is the classes hierarchy of the "cnet.core." package of FDNet project.

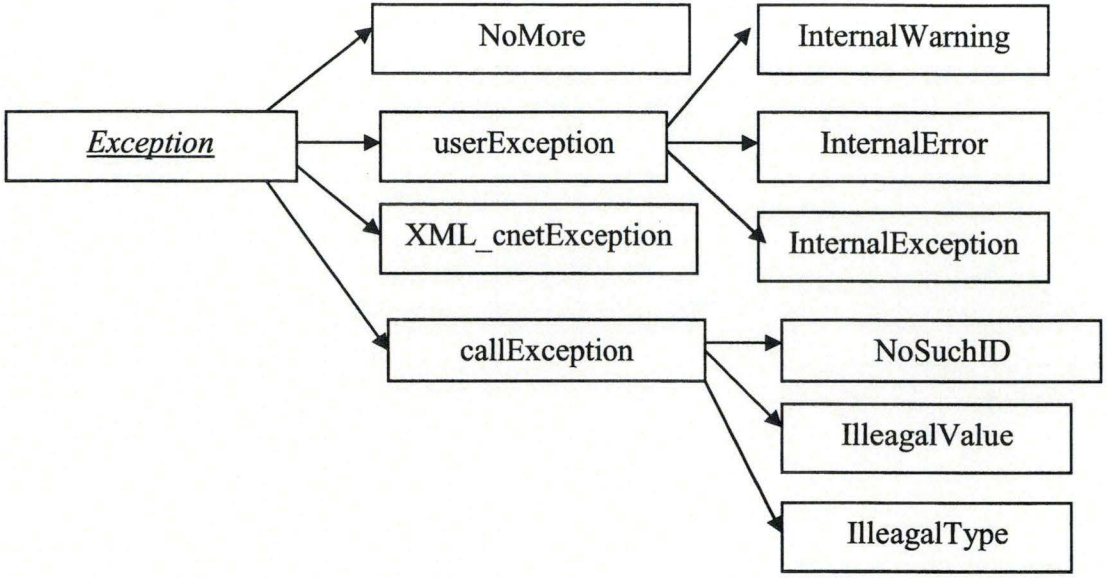
As written just right in the legend :

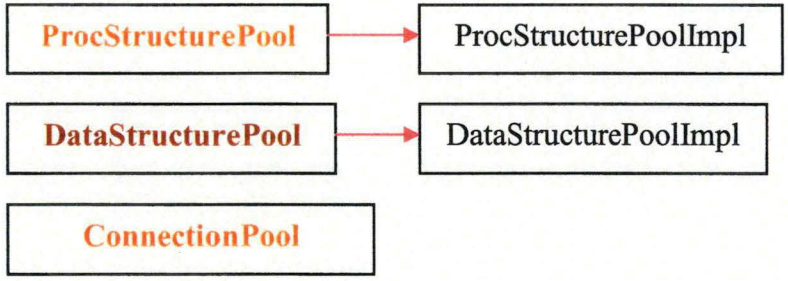
- the classes written in blue are abstract;
- the interfaces are written in orange;
- a class/interface extending another one will have a black arrow arriving to itself.
- a class/interface implementing another one will have a red arrow arriving to itself
- underlined and italic text means that the class comes from another package;

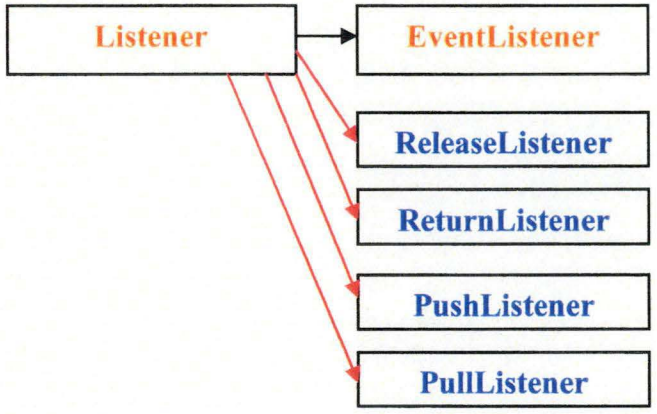


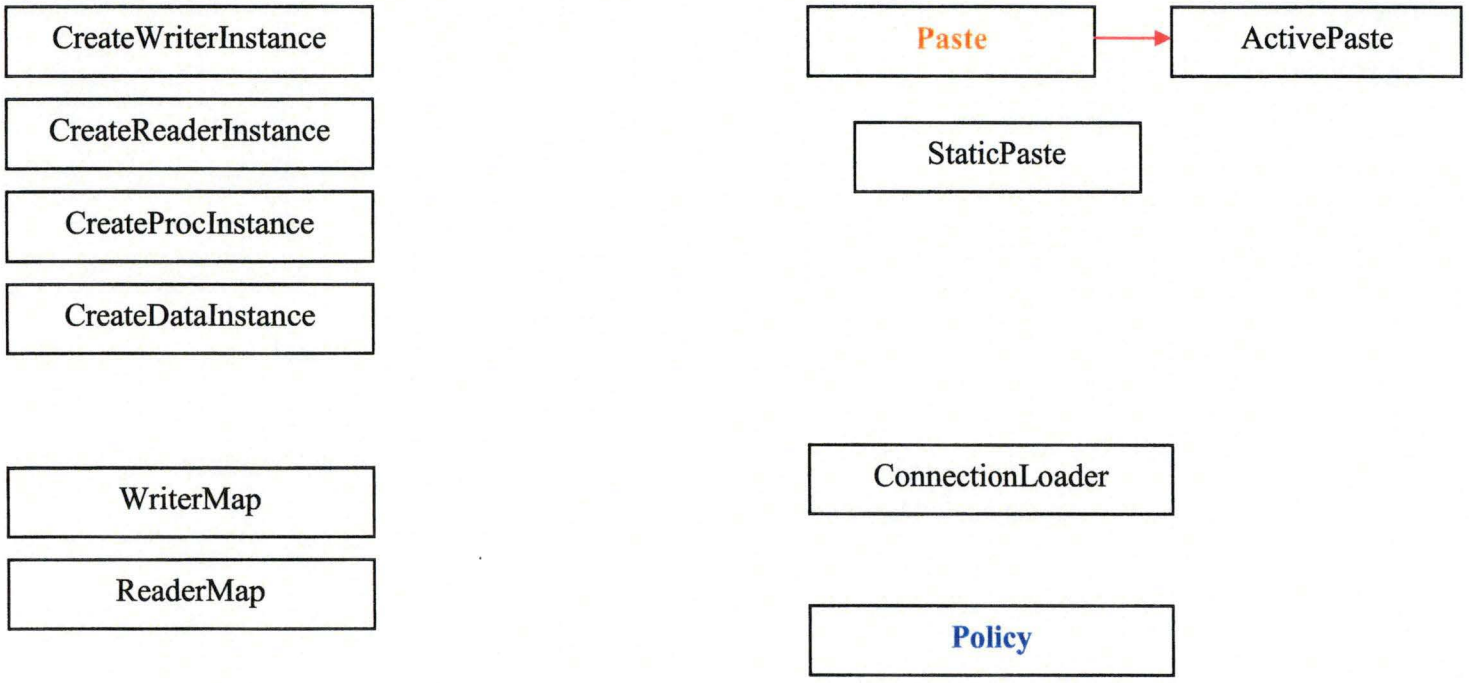

FDEdgeDelegate
FDEdge
FDObject
FDNode
DataStructure
DataStructureImpl
ProcStructure
ProcStructureImpl
ConnectorServer
FDNodeDelegate
Data
DataImpl
ConnectorClient
Proc
ProcImpl
Servant
Helper


WriterImpl
WriterAdapter
Writer
OutputOperation
FDEdgeDelegate
Connector
ConnectorOperation
InputOperation
WriterStructure
FDObject
Reader
ReaderImpl
ReaderAdapter
ReaderAdapterParent
ReaderStructure
FDEdge


Exception
NoMore
userException
XML_cnetException
callException
InternalWarning
InternalError
InternalException
NoSuchID
IlleagalValue
IlleagalType


ProcStructurePool
ProcStructurePoolImpl
DataStructurePool
DataStructurePoolImpl
ConnectionPool


Listener
EventListener
ReleaseListener
ReturnListener
PushListener
PullListener


CreateWriterInstance
CreateReaderInstance
CreateProcInstance
CreateDataInstance
WriterMap
ReaderMap
Paste
ActivePaste
StaticPaste
ConnectionLoader
Policy

Appendix 2: Retro engineering documents

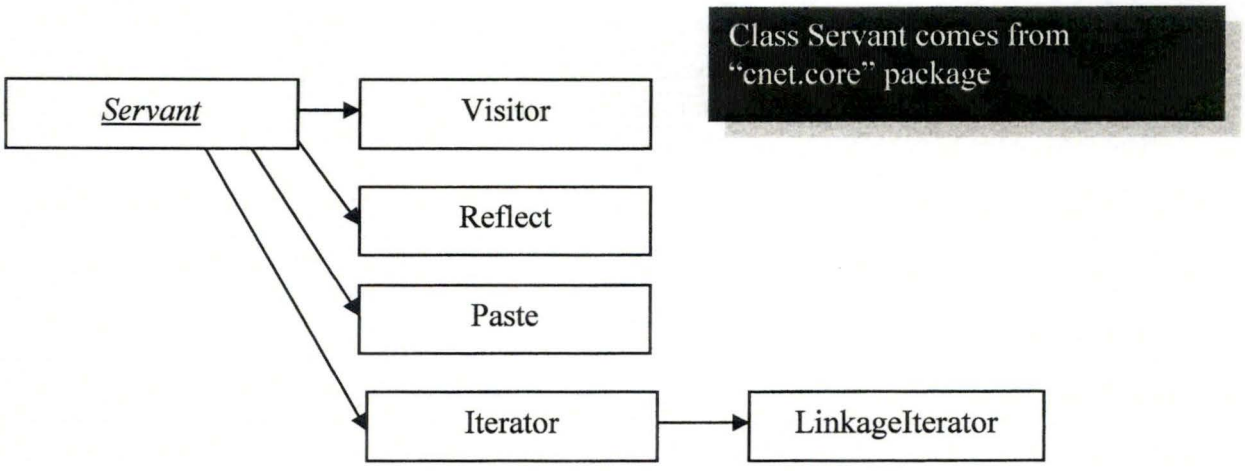
The following documents show you how our retro engineering process was performed. You can see them as a snapshot of our understanding of one part of FDNet's core packages (here, cnet.core.Servant).

As a snapshot, the information it contains is not especially true. Most of what is written comes from the understanding we could have of FDNet's way of working. It thus means that it can still contain errors or information that is too vague to have a real meaning.

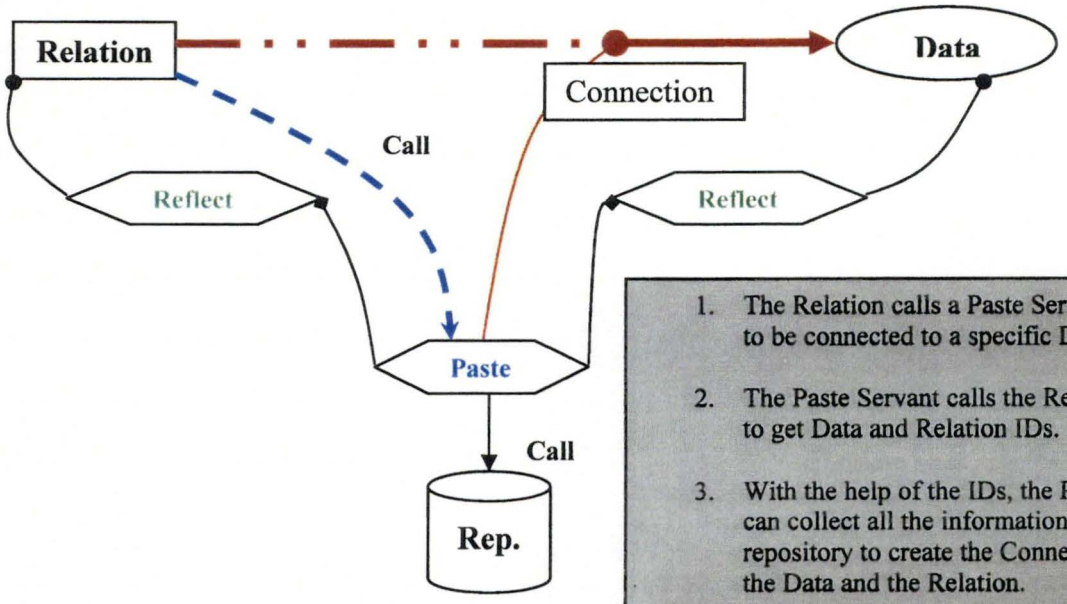
It also contains all the questions we were asking to ourselves at this time. This means that none of the questions asked here had found any answer at that moment.

Basing on theses documents, we could try to speak with FDNet researchers in order to try to understand them and to enhance our own understanding of FDNet.

Study of the classes in package cnet.core.Servant :



The scheme here below explains how the paste and the Reflect servants work:



1. The Relation calls a Paste Servant to ask it to be connected to a specific Data.
2. The Paste Servant calls the Reflect Servants to get Data and Relation IDs.
3. With the help of the IDs, the Paste Servant can collect all the information needed in the repository to create the Connection between the Data and the Relation.
4. The Connection (reader/writer) is created by the Servant.

Class Name	cnet.core.Servant
Extends	FDObject
Implements	null
Aim of the class	This class is handling the Servant system. A servant is a mechanism that allows an information (an FDObject of a "cnet.util.TYPE" type) to be shared on the network. The servant object is an API used by a RELATION to access the network. Relation can only access their directly connected Datas but, through the use of servants, the whole network is reachable. A servant is not linked to any relation at all. Servants are indeed shared objects called by Relations to execute some specific work. Servants have tree mechanisms. It means that there are root servants, parentServants, ChildServants and so on. Servants that are fathers of other ones contain the Class definition of all their ChildServants and can then instanciate any ChildServant on demand.
Comments	A lot of the methods whose aims are to handle the different HashTable are declared Static. It means that all these methods can be called from anywhere in the code and exist only once. It also means that theses static methods are shared among all the instances of Servant Class.

Property name	static private HashMap ServantClasses
Property use	Static HashMap. It contains all the ServantClasses used at a certain time in the program
Comments	HashTable of the classes having the role of Servant ?

Method name	static void setServantClass(String servant_name, String class_name)
Method use	Adding a Servant class to the Servant HashMap
Comments	The Servants added like this will be reachable from any other instance of Servant class.

Method name	static protected Class getServantClass(String servant_name)
Method use	Returns the class associated with the String passed in parameter.
Comments	By having this class, it will be possible to create new instances of the Servant (or classes extending it) and to use it on the network.

Method name	static protected boolean containsServantClass(String servant_name)
Method use	Allows Servants to know if a specific servant, whose class name is given in parameter, is already added in the HashMap ServantClasses
Comments	

Property name	static private Server server;
Property use	
Comments	See cnet.Server for more information Question : What is a Server?

Method name	static void setServer(Server s)
Method use	Sets a new server for this Servant
Comments	

Method name	static Server getServer()
Method use	Get the server currently used by this servant
Comments	

Property name	static private HashMap instances
Property use	A hashMap containing the instances of the Servant Classes
Comments	As it is a HashMap only one instance can be associated with a Servant(Class) Name.

Property name	private long instance_counter;
Property use	Allows to count the number of instances currently held inside HashMap instances
Comments	

Method name	static private String createName()
Method use	Creates a new instance name for the Servant Class to add it in the HashMap instances. The first free position will be used to create the name.
Comments	Names are of the format : %servantName:[NumberOfCurrentInstance]

Method name	Static void setServantInstance(String servant_name, Servant servant_instance) throws ClassNotFoundException
Method use	Adds in the HashTable instances a new instance of the Servant class
Comments	

Method name	static protected Servant getServantInstance(String servant_name)
Method use	Allows to get a servant already contained in the instances HashMap
Comments	

Method name	static protected void releaseServantInstance(String servant_name)
Method use	Delete a Servant from the list of the existing instances
Comments	

Property name	private HashMap children;
Property use	Allows the Servant to creates a tree where he can finds all his child and parent servants
Comments	A servant has only one Parent. This property is not static ⇔ every Servant has it's own children HashMap.

Method name	public final Servant getParent()
Method use	Returns the Parent Servant of this Servant.
Comments	

Method name	protected final void setParent(Servant p)
Method use	Allows to add the Parent Servant of this servant in the children HashMap
Comments	

Method name	final protected void setChildServant(String name, Servant p)
Method use	Add a children of this Servant in the children HashMap
Comments	

Method name	final protected Servant getChildServant(String name)
Method use	gives a specific child Servant with the help of its name
Comments	

Method name	final protected void releaseChildServant(String name)
Method use	Delete a servant from the children HashMap. It means that the servant deleted will not be a child if oursefl anymore
Comments	

Property name	private FDObject object;
Property use	The object that this Servant is sharing.
Comments	A servant only serves one FDObject.

Property name	private Type[] type;
Property use	Contains information about FDObject's data type.
Comments	Defined in the "cnet.util" package

Method name	protected FDObject getFDObject()
Method use	returns Servant's FDObject
Comments	

Method name	protected cnet.core.Servants.Node getObjectData()
Method use	Returns the node associated to the FDObject contained in the Servant
Comments	

Method name	protected Type[] getObjectType()
Method use	gives the type of the Object of the Servant
Comments	

Property name	private HashMap returnListener;
Property use	
Comments	The ReturnListener is a callBack mechanism that allows a servant to tell to relation that has called it that its work is done

Method name	ReturnListener getListener(String name)
Method use	Gets the ReturnListener whose name correspond to the one passed in parameter
Comments	

Method name	protected void setListener(String name, ReturnListener listener)
Method use	Add a ReturnListener to the returnListener HashMap
Comments	

Method name	protected void releaseListener(String name)
Method use	remove the ReturnListener whose name is passed as parameter from the returnListener HashMap
Comments	

Property name	String servant_name;
Property use	Give a name to the current Servant
Comments	What is this name? The Class's name? Something with a structure or something without any structure at all?

Method name	void setName(String s)
Method use	Sets the name of the Servant
Comments	

Method name	protected String getName()
Method use	Gives the Servant's name
Comments	

Method name	void setID(ID[ ] s)
Method use	Set Servant's ID. ID[ ] is a property inherited from FDObject
Comments	The ID of each object in FDNet is a unique attribute given by the system at runtime. The ID allows finding a specific object in the System and is used as identifying information for everybody. This mechanism is not yet implemented

Method name	protected ID[ ] getID()
Method use	Gives Servant's ID
Comments	This mechanism is not yet implemented

Method name	protected Servant()
Method use	Basic Servant constructor
Comments	No code => this does nothing, not even any initialisation The Constructor is protected. What is the use of specifying a protected constructor? What is the meaning? What do they want to achieve by doing this?

Method name	protected Servant(FDObject obj,Type[] t)
Method use	We initialise a servant by initialising the FDObject that is bound to it and by associating a type to this FDObject.
Comments	It appears that a Servant serves one and only one FDObject (to verify). What kind of object can be passed as parameter? relations only?

Method name	protected Servant(FDObject obj,Type[] t)
Method use	This is not an FDObject that we receive anymore to initialise the servant but another servant. This servant, passed in parameter will be known by the currently constructed Servant as it's parent.
Comments	this is here that the HashMap "children" is used.

Method name	public void init(String name, Servant p)
Method use	Initialisation of a Servant by using another one as this Servant's father
Comments	Rem : The first Parameter (String name) is never used in the method. It's not use passing it...

Method name	protected void finalize()
Method use	This method is called when the Servant (ourself) wants to destroy itself. The aim is to free memory and to destroy the links we have with our Servants Parents. The Servant Parent will receive the order to remove us from it's children HashTable.
Comments	There seems to be an error in this method's code. Look in the code to find more explanation about it.

Method name	public Object sendMessage(String serverClass, String message, Object[] args) throws IllegalArgumentException, IllegalAccessException, InstantiationException, InvocationTargetException, ClassNotFoundException
Method use	This methods calls a specific method of a specific class by passing it args parameters
Comments	All is done dynamically => Creation of class, instanciation and so on...

Method name	public Object sendMessage(String serverClass, String message, Object[] args, ReturnListener l) throws IllegalArgumentException, IllegalAccessException, InstantiationException, InvocationTargetException, ClassNotFoundException
Method use	This methods calls a specific method of a specific class by passing it args parameters
Comments	Same as above but we have now a listener mechanism.

Method name	public Servant getRootServant()
Method use	Returns the servant that is the father of all other ones
Comments	Uses the "children" HashMap to getthe information

Method name	public Servant getServant()
Method use	Returns our self (this Servant)
Comments	

Method name	public Servant getServant(String servant_name) throws InstantiationException, IllegalAccessException, ClassNotFoundException
Method use	Returns a specific Servant object corresponding with the "servant_name" parameter
Comments	

Class Name	cnet.core.Servants.Visitor
Extends	Servant in cnet.core
Implements	null
Aim of the class	The aim of this servant is to search for data in the datapool.
Comments	

Class Name	cnet.core.Servants.Reflect
Extends	Servant in cnet.core
Implements	null
Aim of the class	The reflect servant is called by Paste servants and its aim is to fetch information about datas (and give it back to the Paste servants to let them connect the datas with the calling relation). See the schema about this at the beginning of this study (Tengo First group)
Comments	How and when do we connect the reflect servants with the nodes? Does each node (Relation/Data) have it's reflect servant?

Method name	public String getName()
Method use	
Comments	Returns the name of the node contained in the FDObject associated to the Reflect

Method name	public ID[] getID()
Method use	Returns the ID of the FDObject contained in the Reflect.
Comments	

Method name	public ID[] getClassID()
Method use	
Comments	What is this ClassID? When is it used?

Class Name	cnet.core.Servants.Paste
Extends	Servant in cnet.core
Implements	null
Aim of the class	This is the paste Servant used to connect a data object with a relation. The aim of a Paste Servant is to connect data to relations. It gets reference of the data node to connect it to the relation. It is the paste servant that is able to connect data to relation.
Comments	How the things work : 1. Paste Servant get a Reader object from the Data object with the "getReader" method. At this time, the Reader is created by the Data object . 2. The servant passes the Reader to the Relation by calling Relation's setReader() method. The Relation is now connected to the data 3. The Paste Servant is not refered anymore and can die (go back to a servant pool).

Method name	protected void raw_paste(Connection the_connection)
Method use	Pastes the connection to the Relation
Comments	This method currently has no implementation!

Method name	public void create(Connection new_connection)
Method use	Creates the connection to the data to pass it to the Relation
Comments	This method currently has no implementation!

Method name	public void activate(Connection the_connection)
Method use	Activating a connection means creating the link between the data and the relation that needs it.
Comments	

Method name	public void paste(Connection new_connection)
Method use	Creates a connection to connect the data to the relation and pastes it so that the 2 are connected.
Comments	

Class Name	cnet.core.Servants.Iterator
Extends	Servant in cnet.core
Implements	null
Aim of the class	The aim of this servant is to trace the network (follow). The aim is to know another relation's id
Comments	We currently have no information about this class. There is no implementation… We need more complete information about this servant.

Class Name	cnet.core.Servants.LinkageIterator
Extends	Iterator
Implements	null
Aim of the class	The aim of this servant is to trace the network (follow). The aim is to know another relation's id
Comments	We currently have no information about this class. There is no implementation… We need more complete information about this servant.

Appendix 3: FDNetworks Structure definition file

This is the DTD file used to ensure that the XML network definition file is valid:
The Module extension is written in **green.**

```
<?xml version="1.0" encoding="EUC-JP"?>

<!ELEMENT path          (#PCDATA)>
<!ELEMENT host          (#PCDATA)>
<!ELEMENT class         (#PCDATA)>
<!ELEMENT param         (#PCDATA)>

<!ELEMENT name     (#PCDATA)>

<!ELEMENT pstructure     (name,host,path,class,param*)>
<!ELEMENT dstructure     (name,host,path,class,param*)>

<!ELEMENT proc     (#PCDATA)>
<!ELEMENT data     (#PCDATA)>
<!ELEMENT exist EMPTY>
<!ATTLIST exist
        value (yes|no) "yes">

<!ELEMENT read          (name?,(data|dstructure),(proc|pstructure),exist?)>
<!ELEMENT write         (name?,data,proc,exist?)>

<!ELEMENT connection  (read|write)*>

<!ELEMENT module ((pstructure|dstructure)*,connection*)>
<!ATTLIST module
      name CDATA #REQUIRED
      loadatstart CDATA value (yes|no) "no">


<!ELEMENT cnet          ((pstructure|dstructure)*,connection*,module*)>
<!ATTLIST cnet
        name CDATA #IMPLIED>
```