MASTER IN COMPUTER SCIENCE

FDNet: enhancing human interface with dynamic capabilities

Lambot, Nicolas

Award date:
2004

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

FDNet: Enhancing Human Interface with Dynamic Capabilities

Nicolas LAMBOT

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

FDNet: Enhancing Human Interface with Dynamic Capabilities

Nicolas LAMBOT

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

Résumé

L' I.R.S.I. « International Rescue System Institute » est une organisation Japonaise travaillant dans le domaine des robots, et plus particulièrement sur des robots sauveteurs. Le but de ces robots sauveteurs est d'aider les sauveteurs à retrouver des victimes dans des environnements dangereux résultants d'un désastre tel qu'un tremblement de terre.

Un des projets créés par l'I.R.S.I. est FDNet, diminutif de « Flat Distributed NETwork architecture ». Il fournit une architecture commune utilisable par toutes sortes de robots, leur permettant de coopérer les uns avec les autres.

La première partie de cette thèse expliquera en détail le projet FDNet tandis que les parties suivantes présenteront mes contributions à ce projet ; lesquelles étant l'ajout de capacités dynamiques à l'«Human Interface», une application facilitant la manipulation de FDNet par l'homme.

Abstract

The International Rescue System Institute [I.R.S.I.] is a Japanese organisation that works in the field of robots, and more particularly on rescue robots. The purpose of those rescue robots is to help rescuers finding victims in unsafe environments resulting of a disaster such as an earthquake.

One of the projects created by the I.R.S.I. is FDNet, acronym for "Flat Distributed NETwork architecture". It provides a common architecture usable by any kind of robot, allowing them to cooperate with each other.

The first part of this thesis will explain the FDNet project in details while the next parts will present my contributions to this project; which were to add Dynamic Capabilities to the Human Interface, an application facilitating the manipulation of FDNet by humans.

Acknowledgements

First of all, I would like to thank Mr. Schobbens for finding the place of the thesis and Mr. Tadokoro for its subject. Mrs. Tadokoro was also of an incredible help for all the administrative formalities. Without those three persons doing this thesis wouldn't have been possible.

Then I would like to thank Mr. Tokuda and his "RoQ" team for their warm welcome and their help. It has been a pleasure to work with them on FDNet during those four months.

To all the students I met in the laboratory, I would like to thank them for their support and for the great moments we had together.

A great thank to all the secretaries of the I.R.S.I. whom helped me to overcome any problem during my stay in Japan. More particularly, I would like to thank Tomoko, who has been a guide, a translator and moreover a very good friend to me.

Mr Youssef Achbany and Mr. Jérôme Jadoulle, two other students of my university and very good friends of mine, were also with me. I would like to thank them for their support, their help, and for all the amazing moments we have shared together. I could not have thought of better persons to do this adventure with me.

Lastly I would like to thank my family and all my friends for supporting me during all those years.

Table of Contents

Table of Figures

Glossary

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 or created by Relations.

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

FDNet's core: The host on which the FDNetwork will be working.

Human Interface: This is an application allowing users to easily edit an FDNetwork before and while it is working.

Logger: This is an application allowing to keep a trace of every event generated by a working FDNetwork.

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

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.

Swing: This is a graphics library for Java. It supersedes and extends the Java AWT library.

Viewer: Also called "Network State Viewer", this application allows to graphically display the values of a Data node in real time.

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

FDNet[1]

[1] Most of this chapter comes from the paper "Flat-distributed network architecture (FDNet) for rescue robots", [Yosihisa Koji 2002]

1 Introduction

The International Rescue System Institute [I.R.S.I.] is a Japanese organisation that works in the field of robots, and more particularly on rescue robots. The purpose of those rescue robots is to help rescuers finding victims in unsafe environments resulting of a disaster such as an earthquake. The world where the rescue robot operates is very complicated. Regarding this environment, the topography is unique in every place, every time. Unexpected situations might always arise while operating among the debris. Even an expected situation is complicated enough to be confusing. The first issue is how the rescue robot can cope with that complexity in order to act in such environment.

We cannot prepare the robot to fit each kind of environment, one after the other. Because of various trade-offs, the behaviour of an all-around robot can't be established. Therefore we need the rescue robot to have some sort of intelligence, a software architecture that combines the various fundamental technologies and new skills dynamically learnt in that specific place. That architecture is FDNet, a Flat Distributed Network architecture.

FDNet is especially based on ORiN, ORCA and Open-R, three previous architectures. These architectures have no structure to perform dynamic self-organization or dynamic reconfiguration, making them unusable for the rescue robots, which is the reason why FDnet took only the interesting characteristics from these architectures in order to create its specific and common architecture.

Like those architectures, FDNet is a flexible, extensible and generic architecture.

1.1 Flexibility

Given the complexity and variation of the environment, a rescue robot can't be entirely autonomous. Under these conditions, we must command it and watch over it. 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 feature half autonomy. It means that the robot must be able to perform complex operations given a simple order, but it have to wait for a new order when it falls into conditions where human judgment becomes necessary.

In this purpose, we need a common software architecture that shows flexibility regarding both software and hardware, our goal being to describe an intelligent system with that architecture.

1.2 Extensibility

FDNet architecture is divided into several layers, allowing to limit the impact of future changes or improvements to the layer concerned. Thus this extensibility allows by example to easily manage new robot sensors in the system.

1.3 Generic architecture

One of the main goals of FDNet is to provide a common protocol and a common architecture usable by all rescue robots. By doing this, it becomes possible for the rescue robots to exchange information with each other.

As an example a group of robots of different types like "crawler robots", "legged robots" and "flying robots" could be formed. Using the FDNet architecture, those robots may exchange information with each other. The flying robot could by example provide a global view of the environment to the other robots on the ground and it could ask them to search specific areas while the legged robots could ask the crawler robots to search specific areas too small for them to enter.

2 FDNet in more details

FDNet is a **F**lat **D**istributed **Net**work architecture based on the human imitation model. Its aim is to allow multiple rescue robots to interact with each other in order to help rescuers finding victims in case of disasters.

2.1 Flat Distributed Network architecture

FDNet uses a neural-like network to represent the intelligence of the robots.

The FDNetworks are composed of two components: The **Nodes** and the **Connections**. The Nodes are either raw information or processing objects, and the Connections are links between the Nodes, allowing the processing objects to access the information.

There are two kinds of Nodes:

1. **Data Node:** This kind of Node represents the raw information. This information can either come from the Network itself or from the environment of the robot, via its sensors. New Data can also be processed by the Relation Nodes.
   All information in the network is considered as a feature. In other words, any Data Node value is decided in the same way without any high-level or low-level distinction.
2. **Relation Node:** This kind of Node has the same meaning in FDNet than the neurons in neuronal networks. It materializes the relation among Data Nodes. By using some input Data Node, the Relation can compute new Data Node values. In this case, this Relation works with a "servant", an agent which has a specific function. A relation itself can only compute the values of directly linked Data Nodes. But through servants, it can access the whole structure of the network. Then the relation can perform dynamic self-organisation or dynamic reconfiguration of the network.

There are two kinds of Connection between the Data and Relation Nodes:

1. **Reader:** This kind of Connection linking a Data and a Relation Node allows the Relation Node to use the Data Node values.
2. **Writer:** This kind of Connection linking a Data and a Relation Node allows the Relation Node to compute new values for the Data Node.

You will find below an example of a basic representation of an FDNetwork.

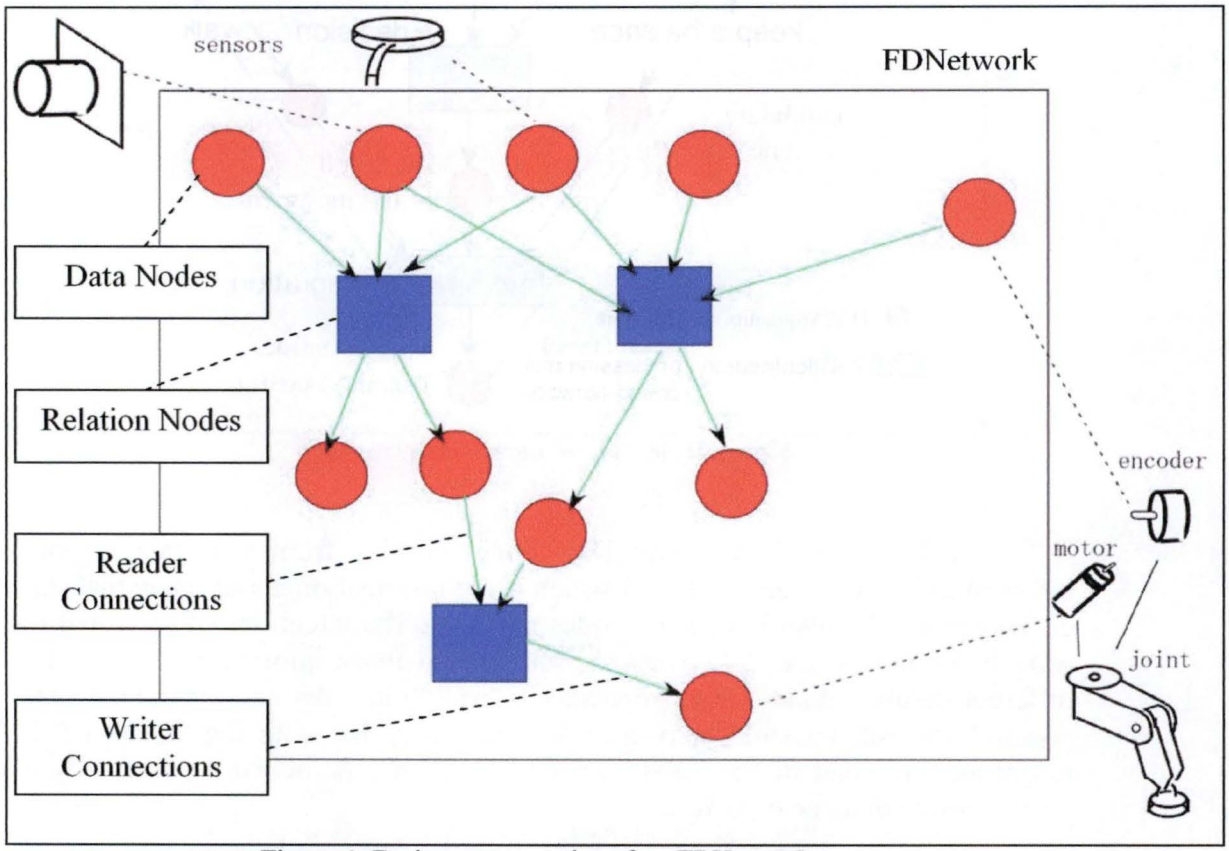


Figure 1: Basic representation of an FDNetwork

Below is another example showing a network allowing to manage the motion of a robot.

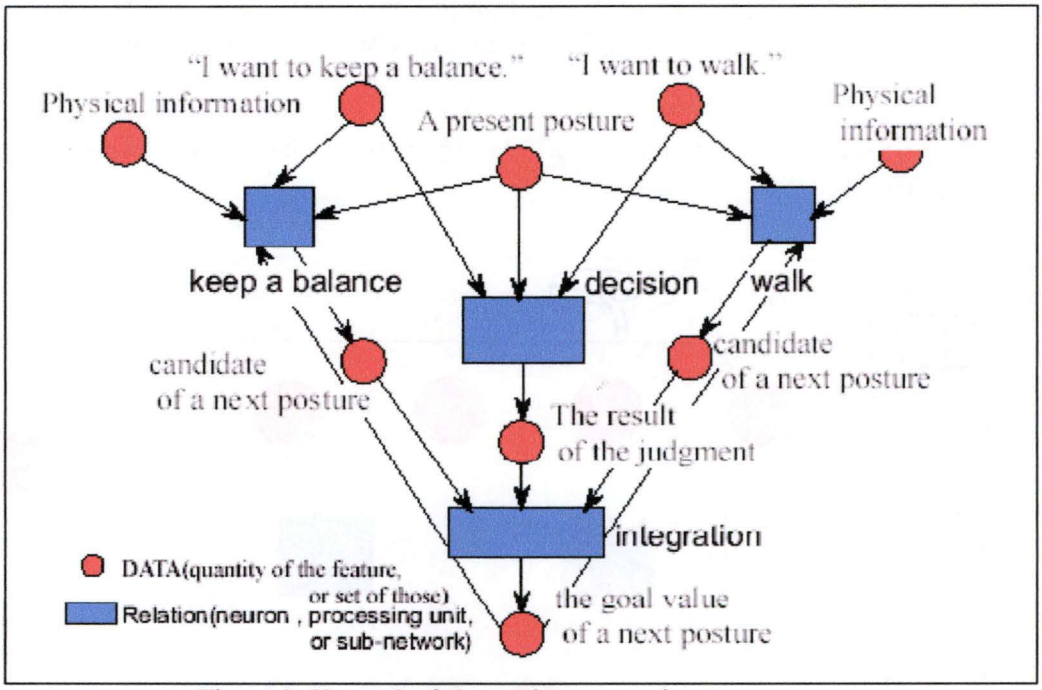


Figure 2: Network of the motion generation

In this network, we have Data nodes coming from sensors ("Physical information", "A present posture") which gives information about the actual state of the robot. We also have Data nodes providing the intentions of the robot ("I want to keep a balance", "I want to walk"). All those information are used in different Relation nodes ("keep a balance", "walk") in order to compute the "next posture" the robot should take in each case. Together with the "result of the judgment" provided by the Relation node "decision", the network will finally be able to compute its next posture.

There is also an important thing to notice in this example. The Data nodes "A present posture" and "the goal value of a next posture" are in fact the same Data node. This means that a new value of the Data node "A present posture" can be computed by one of its child Relation nodes. This is what a "Flat" network architecture means, as there are no restrictions on what Data node can be connected to what Relation node.

As for the "Distributed" network architecture, it has already been explained in "1.3 Generic architecture", as the FDNet architecture allows multiple robots to work with each other in a common FDNetwork.

2.2 The Human Imitation Model

Before specifying the FDNet architecture, the researchers have adopted a recognition model. In FDNet projects, the outline of human imitation is adopted for the rescue robot to solve most of the problems. There have been some researches in which the researchers transposed "features" from a human being to a robot and studied the application to the robot.

As an example, you can see in the figure below an imitation model for the groping motion.

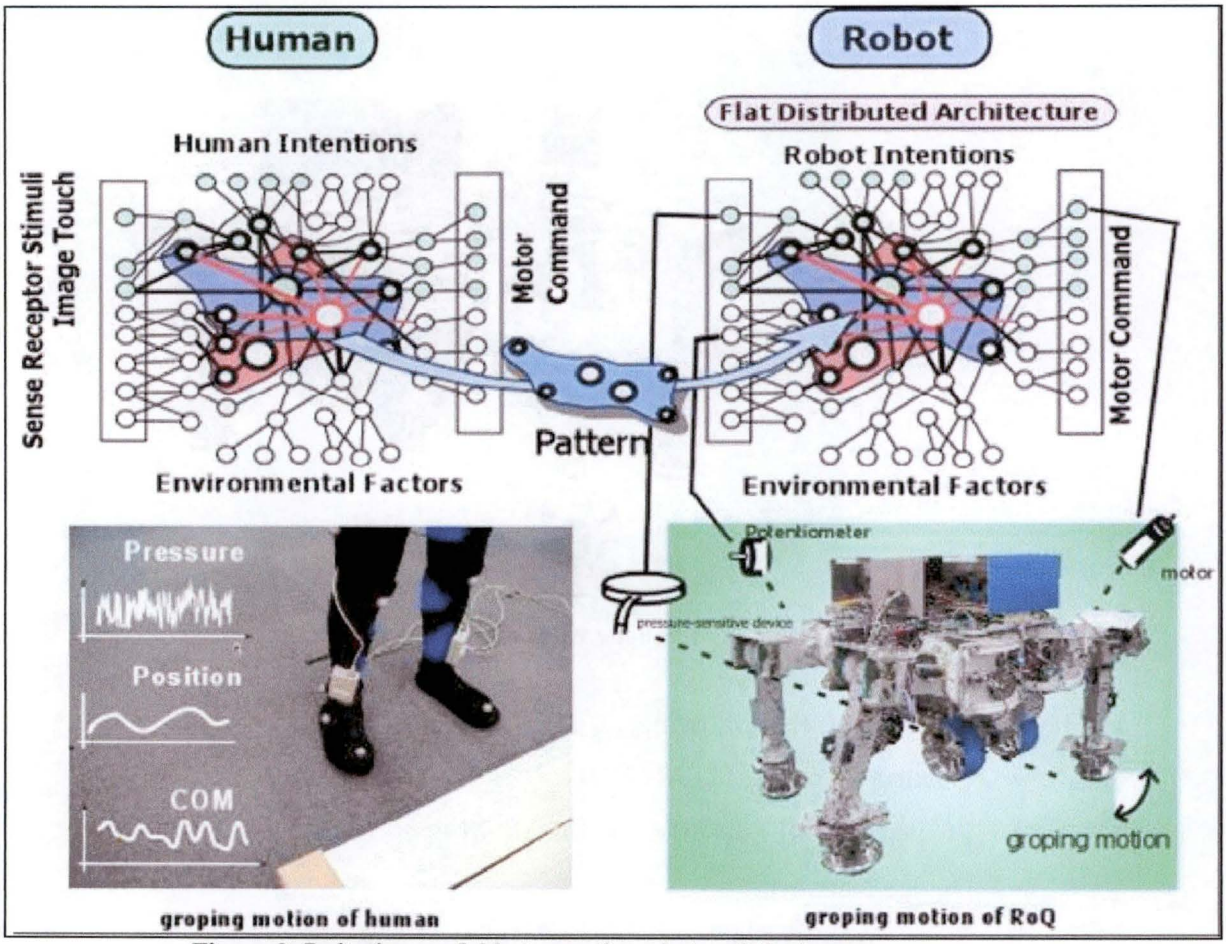


Figure 3: Imitation model between the robot and the human

The human imitation model is used in FDNet in order to capture high-level human tasks and, with the self-organisation of the network, to adapt them for the robot. In this example, the tactile sensing of the human will be realised by the robot by using a combination of movement information and sensor information. The robot will then be able to move its leg, tap the ground by using the leg movements and sensors to see if the new position is safe, and then decide to take the new position or not.

3 Current FDNet implementation

3.1 RoQ

RoQ (**Ro**botic Platform for Res**cue**) is a rescue robot still in development which is used to construct and test FDNet.

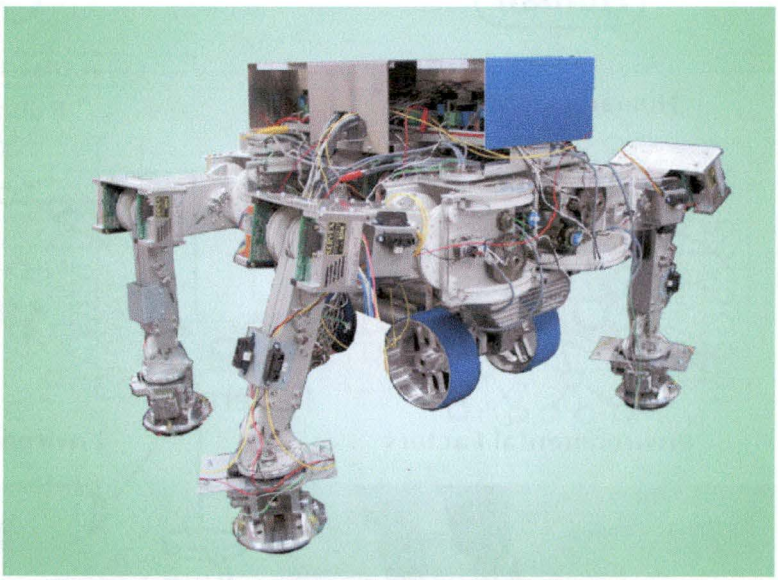


Figure 4: Robotic Platform for Rescue : RoQ

Its base hardware is the following:

- 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 mechanism
- Ankle mechanism

3.2 FDNet environment

- [Operating System] kernel Linux 2.4.4
- [Real time extensions] RTLinux 3.1
- [libc] GNU libc 2.2.5
- [java] Java2 SE 1.4.1_01
- [DBMS] postgreSQL-7.1.3
- [CORBA] OpenORB-1.2.0

3.3 FDNet's API

FDNet's API is defined in the following main layers:

- Network layer (CORBA): Allowing the Connections and the exchange of information.
- Program language layer (Java, C++): Allowing the implementation of Relation and Data nodes.

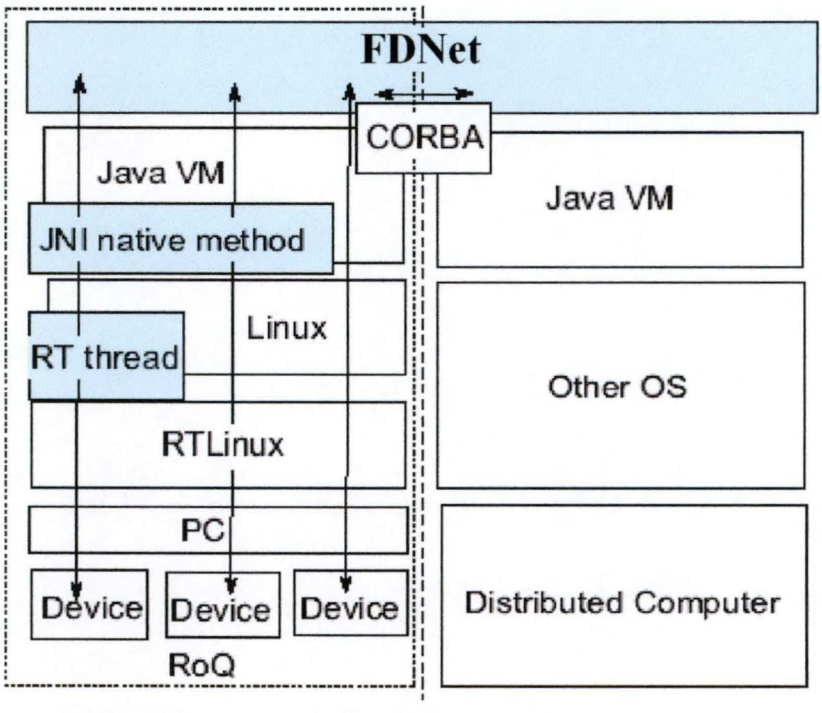


Figure 5: FDNet's API defined within several layers

The choice of Java/CORBA was for Java's portability feature and the choice of RTLinux was to control the parts of the robot that are time critical (i.e. a defined response time is expected). By example, the control of each joint of the RoQ robot is mounted as a real-time task of the RTLinux.

Retro Engineering[1]

[1] This part « Retro Engineering » comes from the thesis of Mr Achbany & Mr Jadoulle [Achbany & Jadoulle 2004], as the three of us did this work together.

1 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.

2 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.

3 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.

Human Interface's Dynamic Capabilities

1 Introduction

The aim of the Human Interface is to give a simple way for FDNet users to manage the robot(s) through an easy edition of an FDNetwork before and while the robot(s) are functioning.

My work was to provide Dynamic Capabilities to the Human Interface. First and foremost it was necessary to be able to see graphically the evolution of Data Node values in real time when the FDNetwork was working. For this I implemented the **Network State Viewer** (further called Viewer) which will be explained in "2 Network State Viewer".

Then I had to provide a way to allow to start an FDNetwork loaded in the Human Interface and to be able to interact with the FDNetwork while it was working, and for this I had to implement the **connection between the Human Interface and FDNet's core** which will be explained in "3 Connection between Human Interface and FDNet's core".

The Human Interface was mainly programmed by Mr. Jadoulle and Mr. Achbany, so if you want to learn more about it apart from the Dynamic parts that will be explained in this chapter, I invite you to read their thesis[1].

[1] See [Achbany & Jadoulle 2004]

In the figure below you can see a global view of the Human Interface.

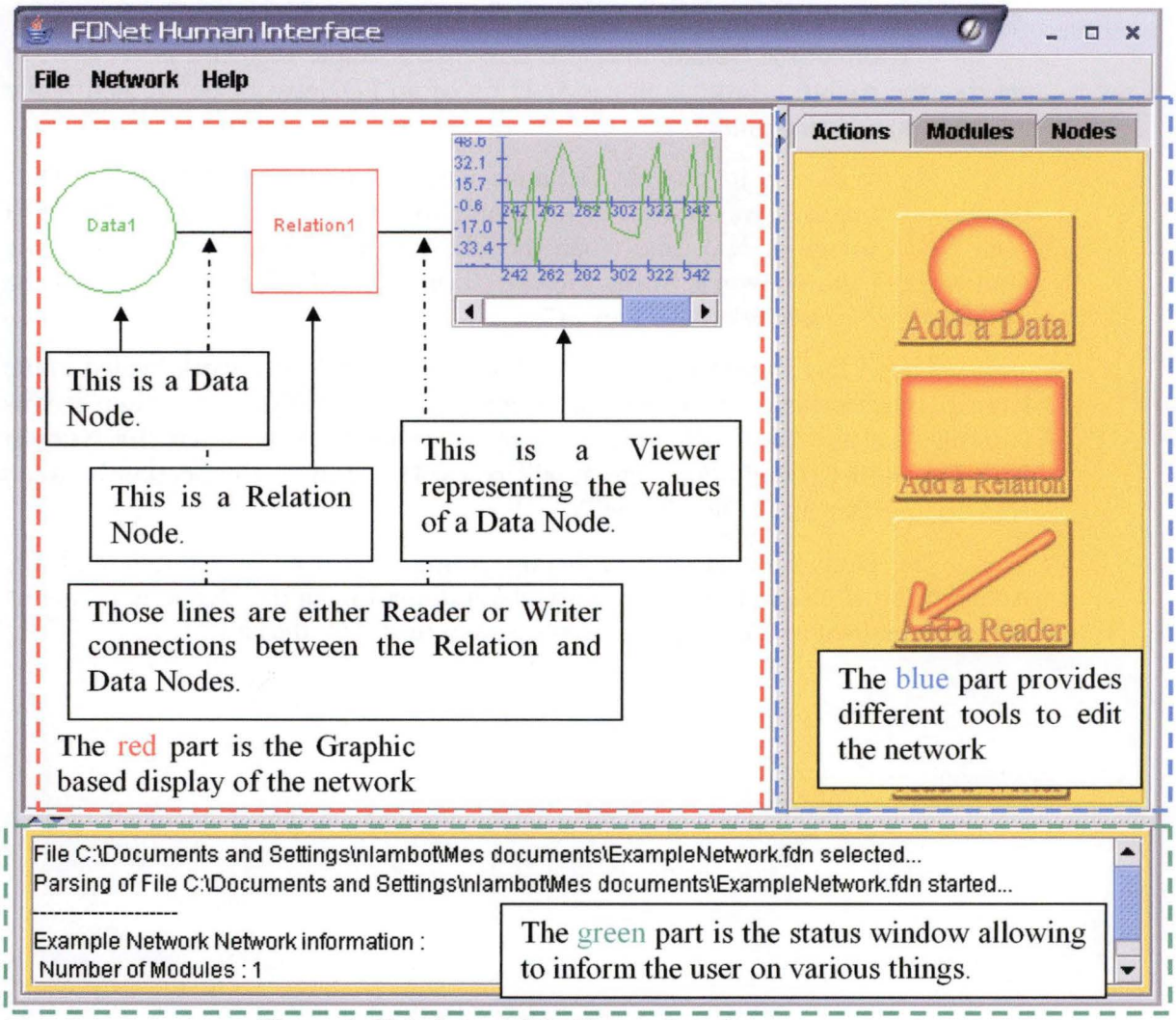


Figure 6: Global view of the Human Interface

2 Network State Viewer

2.1 Introduction

The FDNet Network is composed of nodes which can be Datas or Relations and connections between those nodes. The aim of the Network State Viewer is to display graphically the values of a specific Data node over time and it has to do this in real time, thus displaying the data values as soon as they are generated.

At the beginning, the Network State Viewer was a separate application that has been programmed in C# by a former French research student in Kobe Laboratory. That application was not finished and not fully functional.

Therefore the first thing I had to do was to improve that application with some functionalities that were required for a presentation of FDNet.

In order to do these improvements, I had to get familiar with the C# programming language and with the source code of that application.

After that, as the Network State Viewer had to be used in the Human Interface designed by Mr. Achbany & Mr. Jadoulle in Java, we commonly decided to make it again from scratch in Java and in a way that it could be easily imported into the Human Interface.

2.2 C# Version of the Viewer

I've been asked to improve a first version of the Network State Viewer, programmed in C#, that Mr Alain Pujol has made; And to get it working for a presentation 3 weeks later.

So I had to read some documentation about C#, to read all the source code of Mr Pujol, and then to see how I could implement the modifications required for the presentation:

- Being able to see arrows for readers/writers connections
- Allowing to scroll the Graph
- Generating datas (data values over time) to be used in the presentation

As I said before, this application was first programmed by a former research student in Kobe Laboratory. The figure below shows a screenshot of the original application displaying a really simple FDNet network.

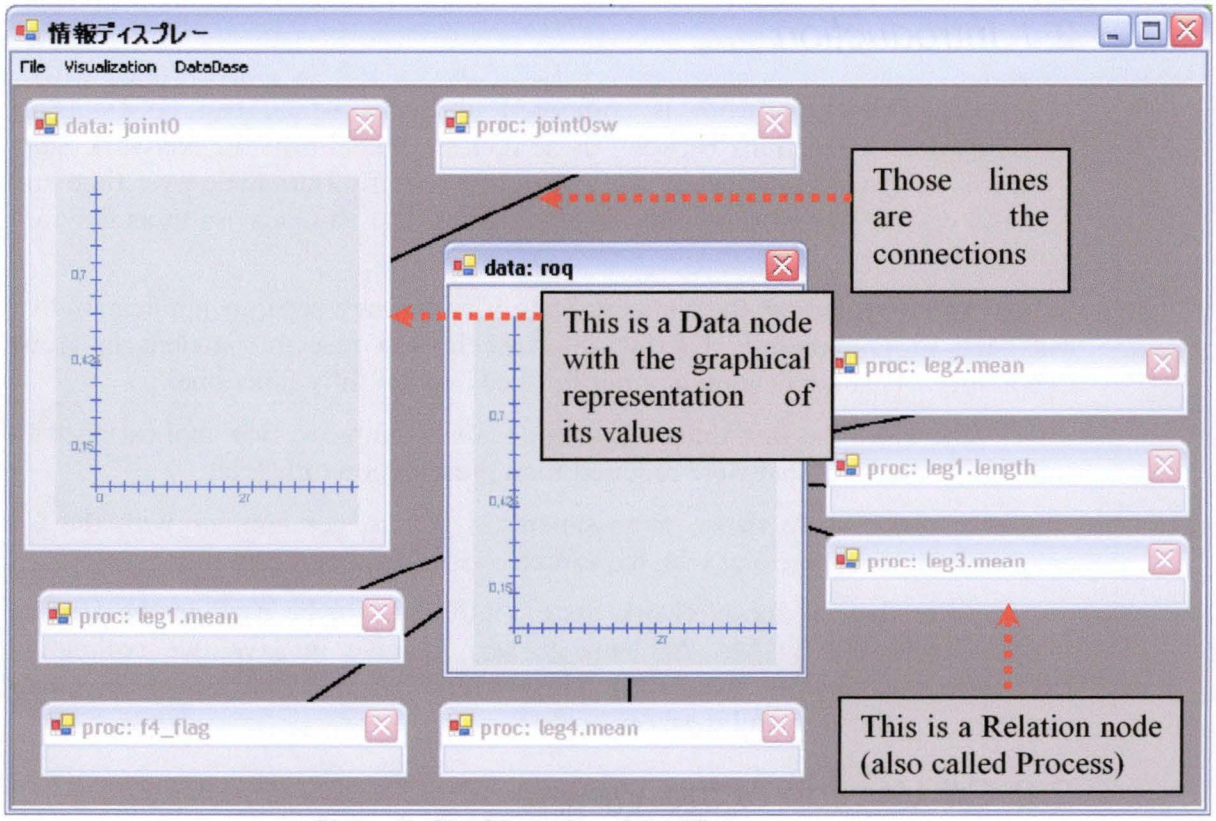


Figure 7: Old C# version of the Viewer

As you can see, this application was more than just a Network State Viewer, as it represented a network with its nodes (process or relation and data) and connections between them. But this version was not sufficient as the graph for the data values was limited (we could only see the first 30 seconds of value) and the connections were not enough detailed (it wasn't possible to know if it was a reader or writer connection).

To review those problems I was asked to improve this application so that it could be used for a presentation of FDNet.

In the picture below, you can see a screen capture of the application after that the changes have been made. Notice the arrows on the connections, allowing to know whether the connection is a reader or a writer. And the scrollbar below the graph of the data values, allowing the user to scroll when there are more values than we can display.

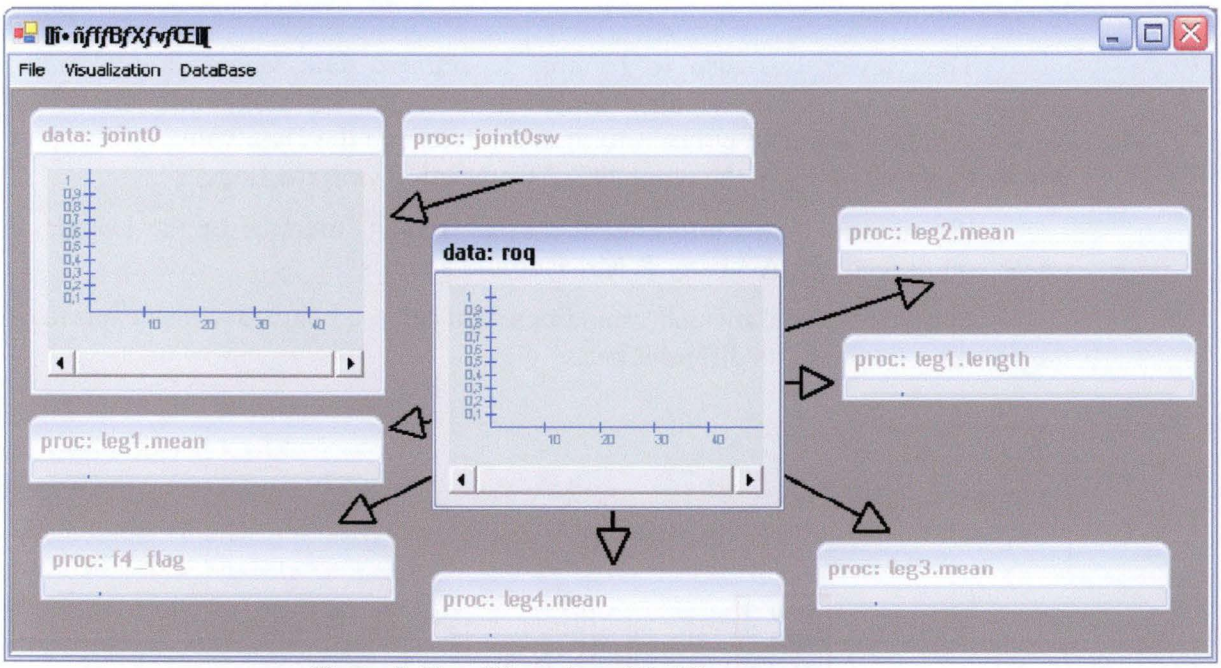


Figure 8: New C# version of the Viewer

2.2.1 Limitations of the C# Version

This version of the Network State viewer was very limited. It could only display values between 0 and 1 and it was not possible to make it work in a real situation by getting data values in real time from FDNet.

For those reasons, and also because the Human Interface in which the Network State Viewer would be later included was being programmed in Java, I had to create a completely new version of the Network State Viewer in Java. This version would only focus on the display of data values, as the other functionalities would be provided by the Human Interface.

2.3 Java Version of the Viewer

As the Network State Viewer would be later included in the **Human Interface** that Mr Jadoulle has created, it had to be easy to import it into other applications. This is the reason why I implemented it as a Standalone Graphical Java Component.

The viewer had also to be able to retrieve data values in real time; And for this, Mr Achbany implemented a server on his **Logger** to allow the Network State Viewer to connect to it; and to retrieve the Data values. This part will be explained later in the point "2.3.2 Connection with the Logger".

If you want more information on the Human Interface or the Logger, I invite you to read Mr. Achbany & Mr. Jadoulle's thesis[1].

In order to avoid misunderstandings, you can see below a screen capture of the viewer explaining its different parts.

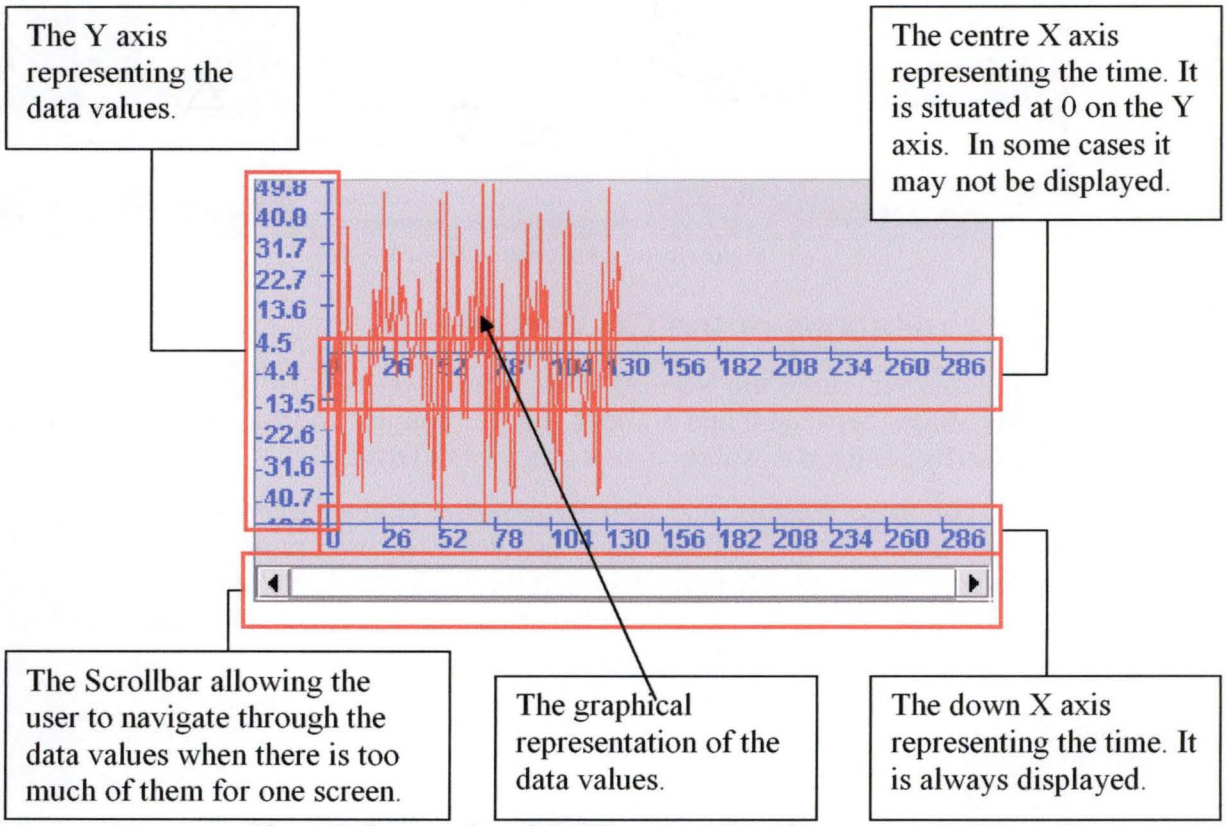


Figure 9: Java version of the Viewer - explanation

[1] [Achbany & Jadoulle 2004]

2.3.1 Standalone Graphical Java Component

The aim of this Graphical Java Component is to Stock, manage and display data values. It has been developed as a single Java package providing a Swing graphical component class that can be used in any Java application using the Java Development Kit version 1.2 and older.

2.3.1.1 Stocking Data values

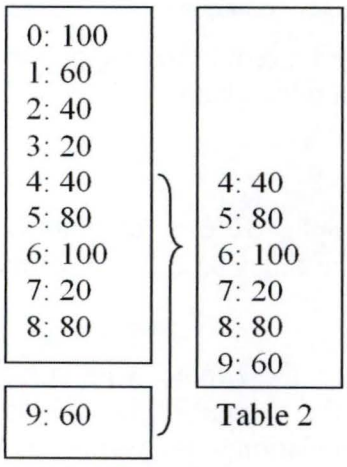
For each Data node of an FDNet Network, we have values that can be any figures and that could change at maximum once every microsecond. The viewer also needs to display the values in real time and it was essential to keep all the values directly in memory, thus avoiding possible lags of using slower non volatile memory such as hard drive. Consequently the memory consumption for displaying and memorizing the values of a specific Data may become really high.

Therefore, I had to limit the amount of values kept in memory.

2.3.1.2 Management of Data values

LIMITING THE AMOUNT OF DATA VALUES IN MEMORY

In order to limit the amount of memory used by the data values, I implemented a limitation of the number of data values kept in memory. This was done by using a maximum amount of data values that, once reached, led to the suppression of old data values, in order to reach an average amount of data values.



In this example, we have:
Maximum amount of data values = 10
Average amount of data values = 6

When we get the 10th value in *table 1*, we suppress the old values in order to keep only the 6 newest ones, as shown in *table 2*.

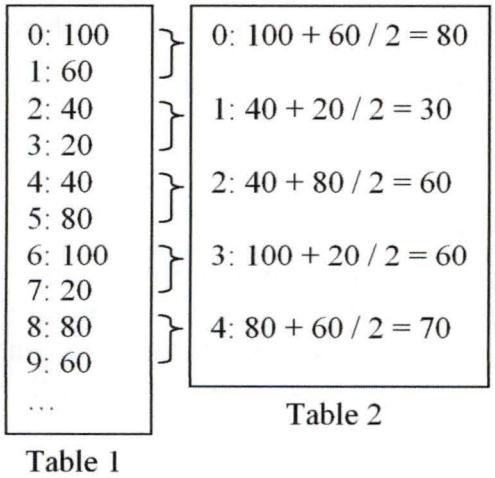
Figure 10: Limiting the amount of data values in memory - example

Another way to reach this aim could be to simply have a maximum amount of data values that, once reached, led to the suppression of one old data value for each new one. But as the viewer used already a lot of processing power and because such process would have been called every time we received a new data value, it would have been a bad idea as the viewer has to work in real time.

FUNCTION FOR MANAGING THE DATA VALUES

Another thing I had to implement was a function that allowed getting a list of data values between two given times and with a given time slice. This function was required in order to allow the modification of the X axis scale for the representation of data values, as it will be explained in "2.3.1.3 Displaying Data values : Allowing to change the scale of the X axis".

You can see in the example below a little explanation of how this function works.

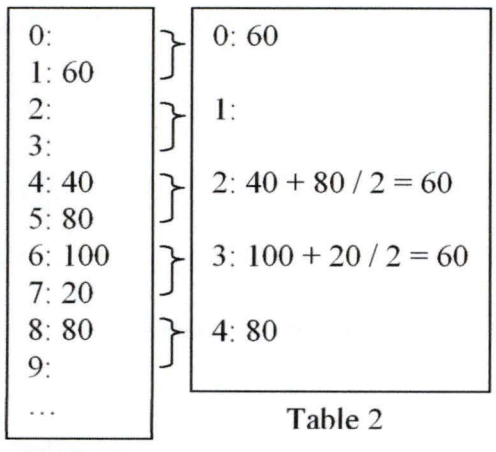


In this example, the *Table 1* would represent the list of data values as they are in memory, with the line number representing the time of appearance of the value in microseconds.

The *Table 2* would represent the list of data values given by the function with a start time of 0, an end time of 9, and a time slice of 2 microseconds. As you can see each value in *Table 2* is an average of the values contained in its time slice.

Figure 11: Function for managing the data values - example 1

But as the data values can occur at any time, it could happen that we receive values in a less ordered manner, in which case we have to be careful, as shown in the example below.



In this example, we can see that we don't have values for the times 0, 2, 3 and 9 microsecond in the *Table 1*.

Consequently, the *Table 2* will have no value for line 1 and the values for lines 0 and 4 doesn't need the calculation of an average as there is only 1 value in those time slices.

Figure 12: Function for managing the data values - example 2

2.3.1.3 Displaying Data values

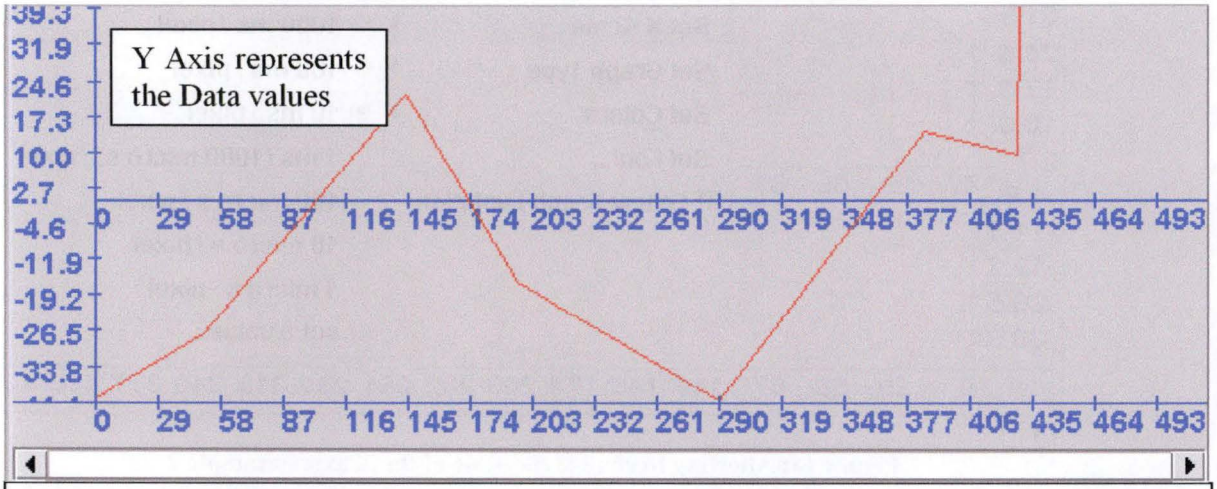
An essential functionality of the viewer is to display the data values. And that is the reason why I spent a lot of time on this part of the application.

Please note that the data values used in the different examples of this section does not have any meaning, as they were randomly generated.

ALLOWING TO CHANGE THE SCALE OF THE X AXIS

Allowing to change the scale of the X axis (representing the time axis) of the graphical representation of the Data values was an essential feature, as it is possible that we receive one new value each microsecond, thus being impossible for the user to cope with the speed at which the values would be displayed. This was also necessary to allow the users to choose between global or precise views of the data values.

In the figures below you will find some screen captures allowing you to better understand how it works.



The two X Axes represent the time of occurrence of the value in the scale chosen for the viewer (here the scale is 1 millisecond per pixel or 1000 microsecond per pixel).
Thus here you can see that the X scale goes from 0 to about 500 milliseconds, meaning that about the first 0.5 seconds of this data's values are displayed.

Figure 13: Allowing to change the scale of the X axis - example 1

And as you can see in the following screen captures, the user can change the X axis scale at any moment by using the contextual menu provided when right clicking on the graphical component with the mouse.

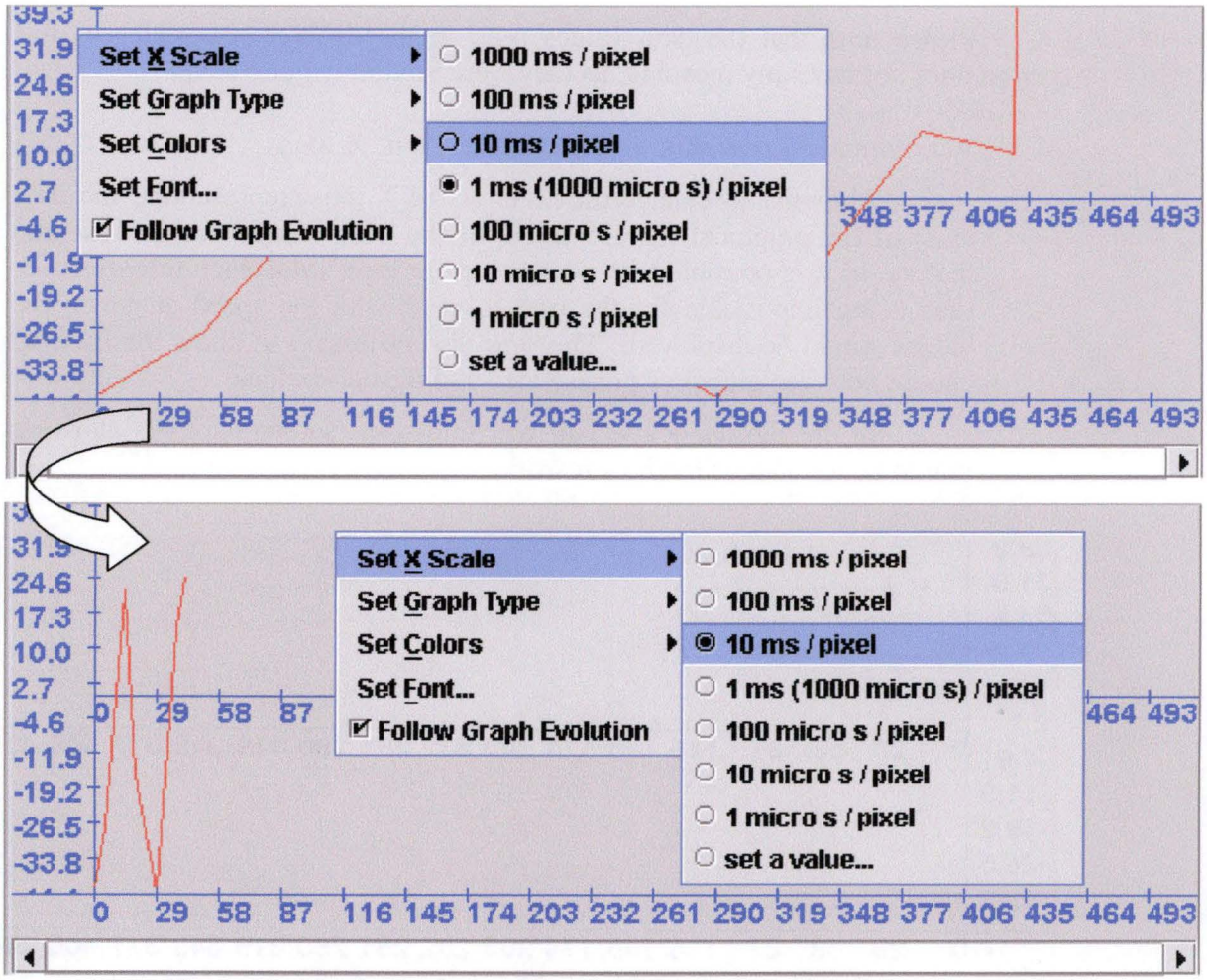


Figure 14: Allowing to change the scale of the X axis - example 2

In the example, the X axis scale has been changed and the graphical representation has been simplified a little, as some data values were merged together. This is explained in "2.3.1.2 Management of Data values : Function for managing the data values"[1] (where the *time slice* is represented by the *X axis scale* in microseconds), as I use this function to get the values that need to be displayed at each given moment.

Allowing to change the Graph type

In order to clearly see the Data values (or average values) and their positions, it was necessary to allow the representation of the data values as *points* in addition to the *line presentation.*

[1] See at page 42

As you can see in the example below, like the other features, this one can be accessed by using the contextual menu provided when right clicking on the graphical component with the mouse.

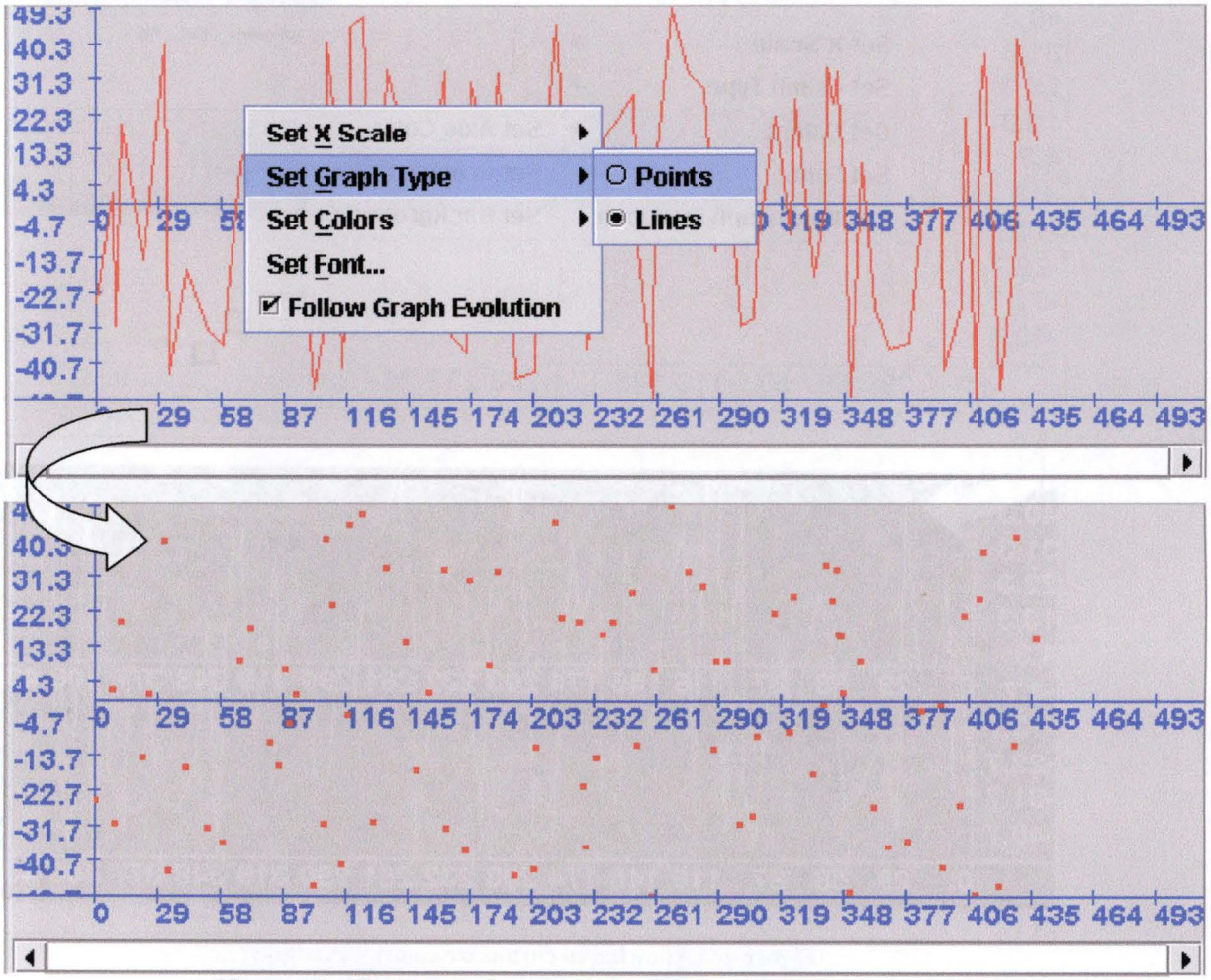


Figure 15: Allowing to change the graph type - example

Allowing to customize Colors

The customization of colors was also an important feature, because once integrated into the *Human Interface* there could be several instances of the viewer running at the same time. And for this reason I allowed the change of colors for the Axes (axes lines, graduations and figures), the Graphic (points or lines) and the Background.

In the example below you can see a modification of the Axes, Graphic and Background colors.

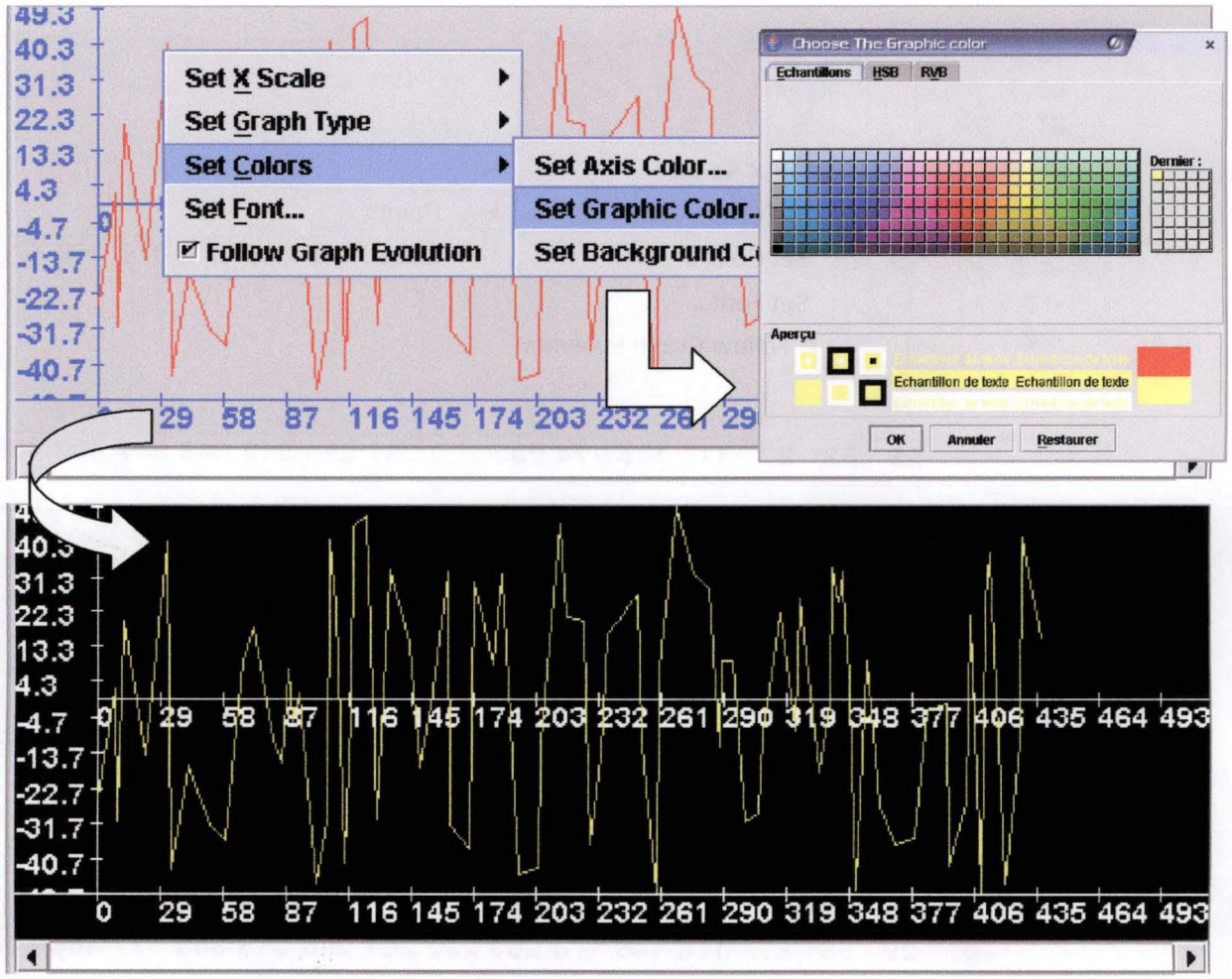


Figure 16: Allowing to customize colors - example

Allowing to customize Text

This feature allows the user to modify the police of character used to display the figures on the axes, thus with this the viewer can be used without problems in either small or big screen resolutions, as the text can be adjusted in consequence.

You can see in the example below the window allowing to choose the font. As this feature was not provided by Java (unlike the Color chooser in the point above), I had to search on the internet to see if such thing hadn't already been programmed, and in the process, I found the project *Ozten's JFontChoose* which I used. This project is under the GNU LESSER GENERAL PUBLIC LICENSE but it has since then been discontinued.

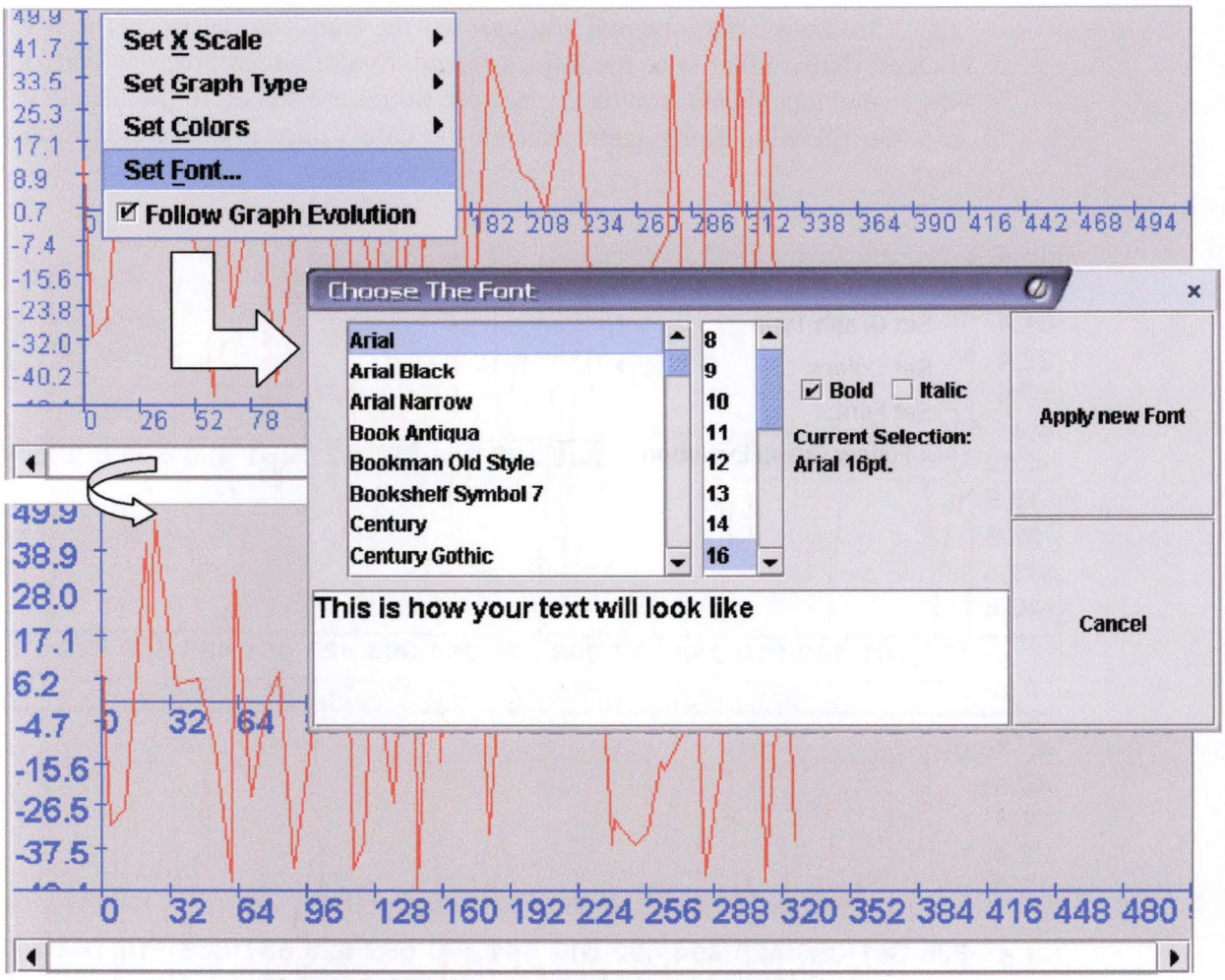


Figure 17: Allowing to customize text - example

You can also notice that the viewer have adapted itself to work correctly with the new (bigger) font size. More space is given for the figures of the Y axis and the X (down) axis, also the size of the graduations becomes bigger in order to compensate for the size taken by the figures.

The graduations are created in such a way that it will try to display the most graduations possible, without having figures overlapping each other.

Allowing to follow the Graph evolution

This feature allows the user to decide whether he wants to see the new data values as soon as they arrive or stay at the same position to look more closely.

Like the other features, this one can be accessed by using the contextual menu provided when right clicking on the graphical component with the mouse. But this time the choice is provided by a checkbox, as there were only two possible choices.

In the example below, you can see the transition between the two choices. Notice that once the *Follow Graph Evolution* feature is activated, when new data values arrives, the newest values are shown to the user (the scrollbar allowing to navigate through the data values is positioned to its end).

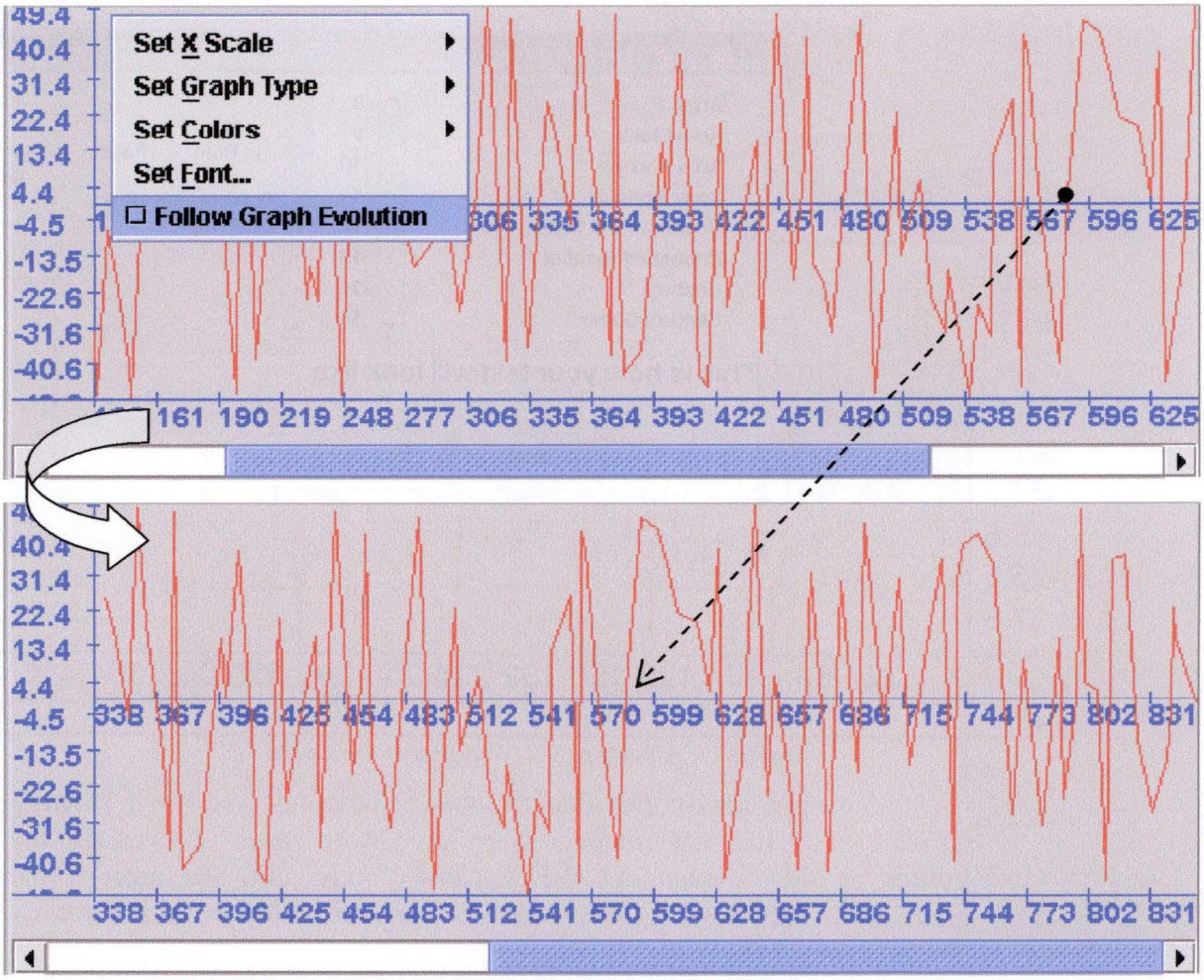


Figure 18: Allowing to follow the graph evolution - example

AUTOMATICALLY ADJUSTING THE SIZE OF THE GRAPH

Because the Data values can be any figures, it was necessary to automatically adapt the Y axis whenever new bigger or smaller values occurred. Also, it was necessary to implement the second X (down) axis which is displayed continuously in the case of data values never going to 0.

Below are some examples.

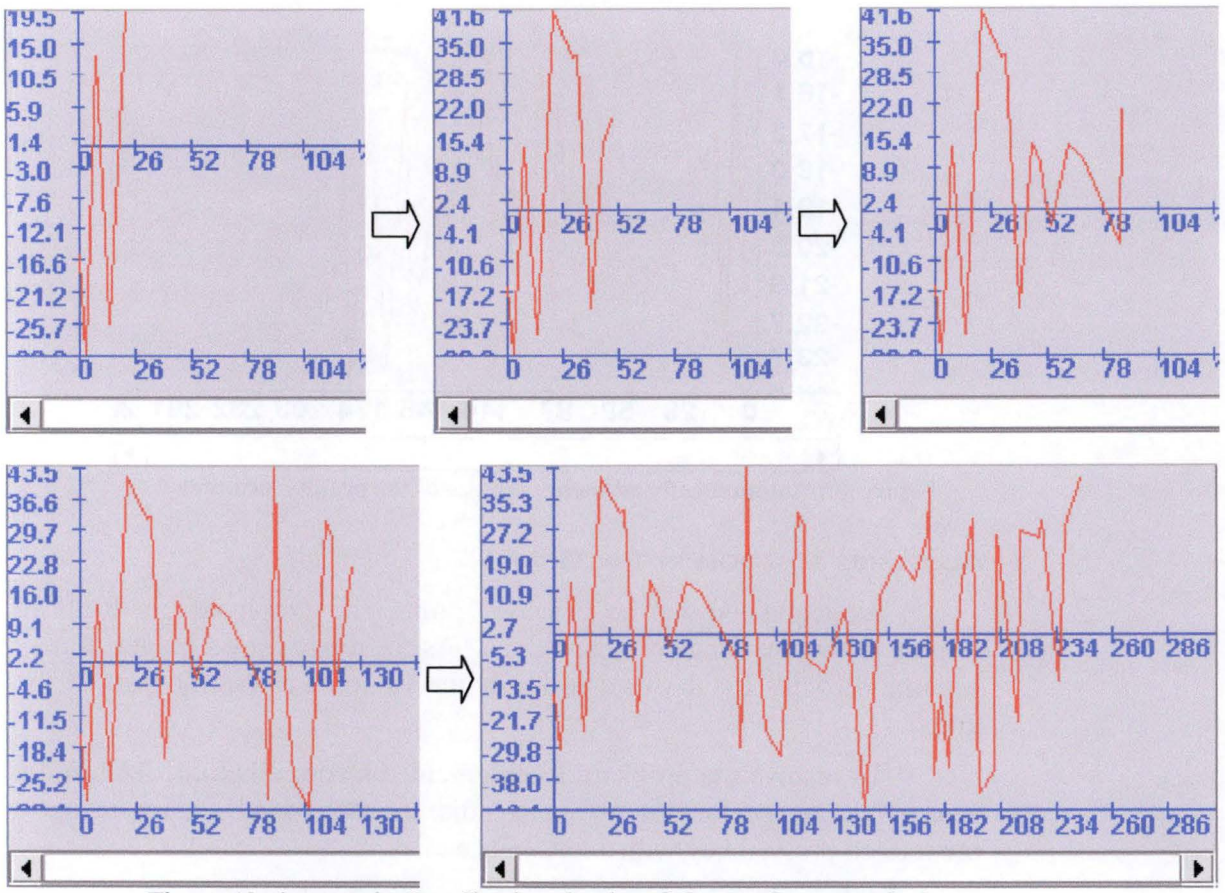


Figure 19: Automatically adjusting the size of the graph – example 1

In this example, the Y axis, ranging in the beginning from about -30 to 19.5 has automatically adjusted itself to come to about -46 to 43.5.

Note that the X axis scale of this example is of 10 milliseconds per pixel, and thus it can happen that the last line of a graph changes after some time because all the values for that time slice may not be available yet.

Below is an example showing the importance of the X (down) axis when all the values of a data are far away from 0.

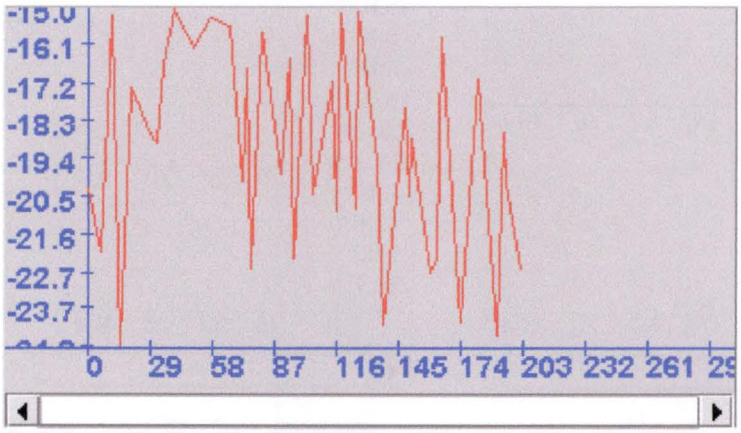


Figure 20: Automatically adjusting the size of the graph – example 2

Allowing to zoom in the Graph

As explained in the previous point, the graph adapts itself to display the whole range of data values. As the range grows bigger, it may become difficult for the user to know the value of a specific part of the graph.

To resolve the problem, I implemented a zoom feature. This zoom can either be performed by using the mouse wheel or by using a predefined pressed key with a left mouse click.

Using Mouse wheel

The zoom behaves the same way as in picture editing programs.

Turning the mouse wheel up when the mouse cursor is on the graph will zoom in on that part of the graph and centre the view on it.

Turning the mouse wheel down when the mouse cursor is on the graph will zoom out and try to centre the view on that part of the graph.

You can see an example below.

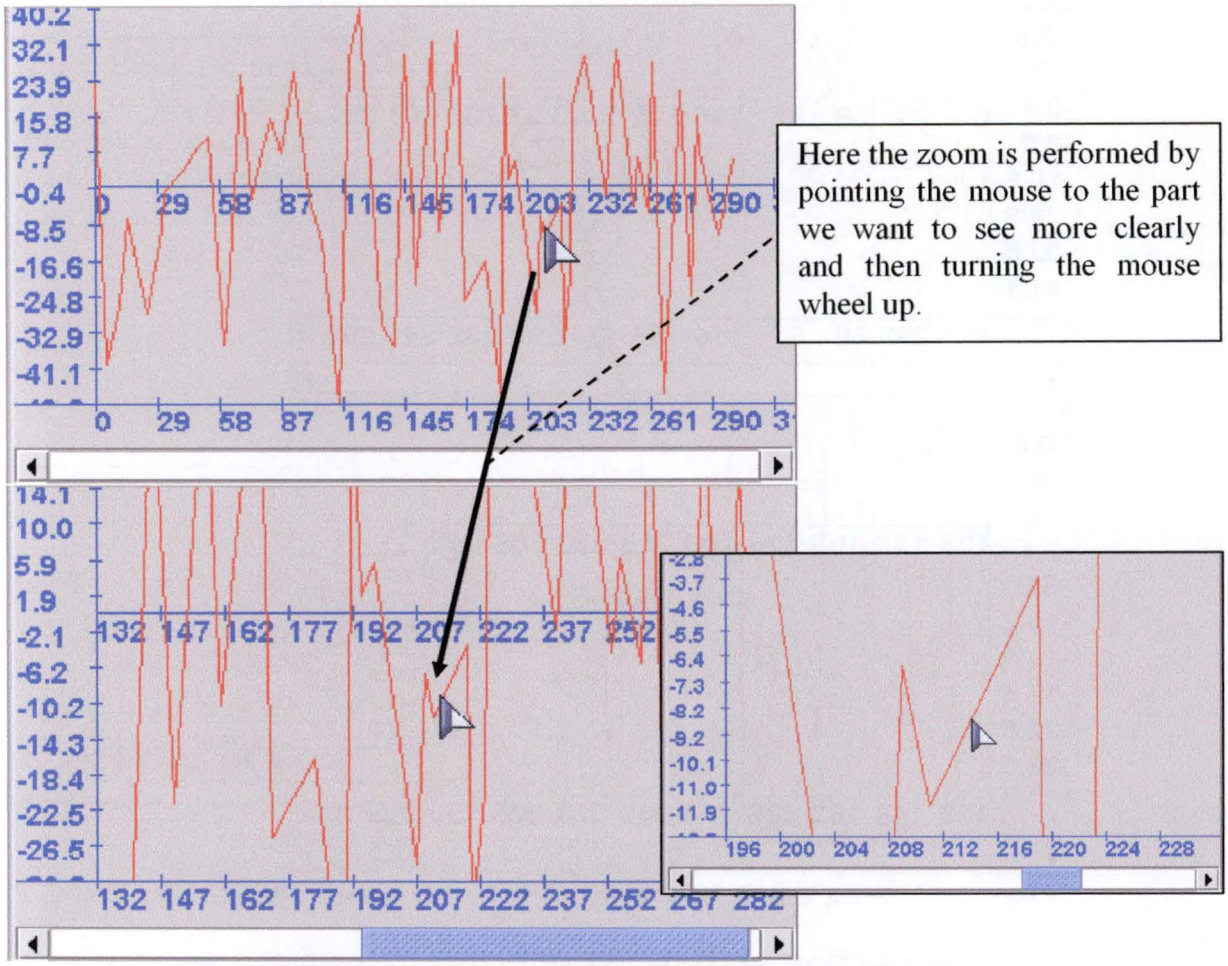


Figure 21: Allowing to zoom in the graph with the mouse wheel – example

Using Pressed key and Mouse click

This feature behaves exactly like when using the mouse wheel, except that it requires the user to press a specific key and perform a left mouse click on the graph in order to zoom in or out, as explained below.

This mouse cursor is displayed when the user is pressing and holding the **SHIFT** key while being over the graph. When this cursor is displayed, if the user does a left mouse click on the graph, it will zoom in.

This mouse cursor is displayed when the user is pressing and holding the **CTRL** key while being over the graph. When this cursor is displayed, if the user does a left mouse click on the graph, it will zoom out.

Figure 22: Allowing to zoom in the graph with a pressed key and a mouse click – figure 1

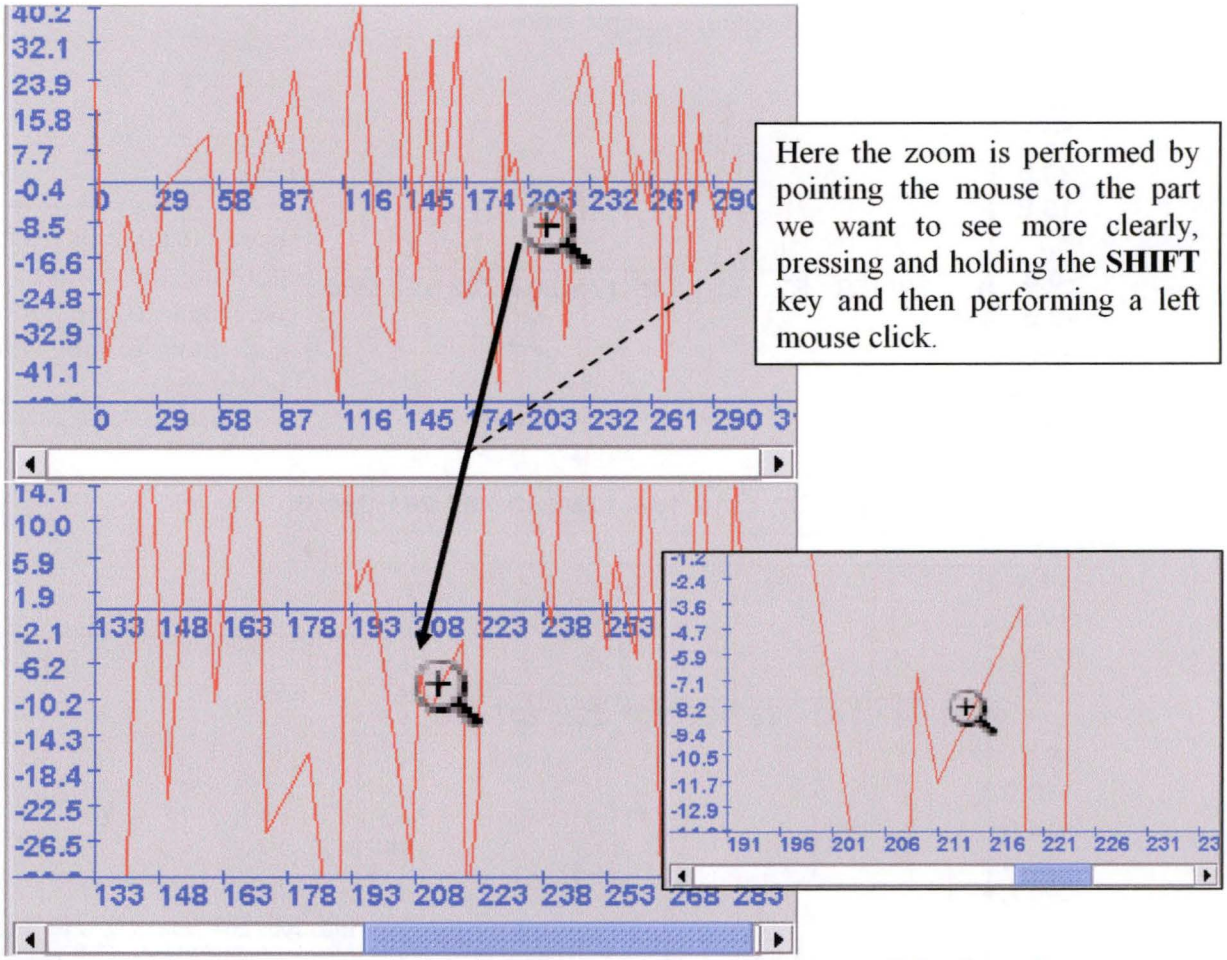


Figure 23: Allowing to zoom in the graph with a pressed key and a mouse click – figure 2

2.3.2 Connection with the Logger

As I said before, the data values are provided by the **Logger** realized by Mr. Achbany. For a better understanding I will start by explaining quickly the purpose of the Logger and what it does for the viewer. If you would like to have more information on the Logger, I invite you to read Mr Achbany & Mr Jadoulle's thesis[1].

2.3.2.1 The Logger

The main purpose of the Logger is to behave like a buffer between the FDNet Core and the Database. In fact, when the FDNetwork is running, it sends all the information generated to the logger in order to keep them for later reviewing. And among those information are the data nodes values.

[1] [Achbany & Jadoulle 2004]

Below you can see the general architecture of the logger.

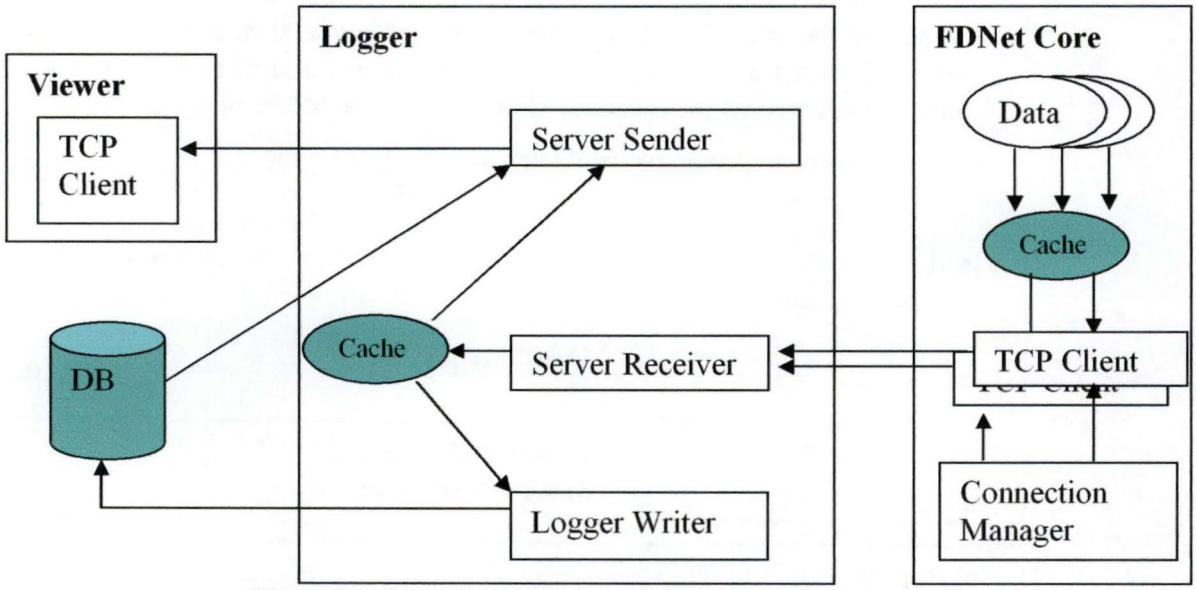


Figure 24: General architecture of the Logger

The part of interest in this figure is the one working with the viewer, namely the "Server Sender".

2.3.2.2 The Viewer

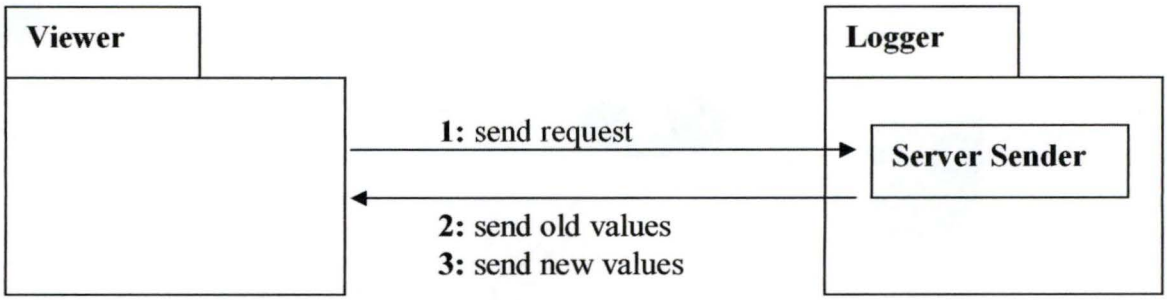
For the connection itself, we have to provide the viewer with some information. As the protocol used to communicate is TCP/IP, we need the IP address of the Logger and the port number used by its "Server Sender". When the viewer is correctly configured and when the logger is running, we can initiate the connection.

There are three possibilities for the connection: we can ask to Start, Pause or Stop the connection with the logger.

STARTING THE CONNECTION

When we start the connection, we will ask the logger to send us values of the specific data displayed by the viewer, then it will take all the values it has for this data and send them to us, and after this, whenever new values arrives for this data, they will be sent to the viewer.

You can see an example below.



Comments:
1: The Viewer sends a request to the logger and provides the identifier of the Data Node it is displaying.
2: The Logger retrieves all the values it has for this Data (in its cache and/or in the database) and sends them to the Viewer.
3: Whenever new values arrive in the Logger, they are directly sent to the Viewer.

Figure 25: Starting the connection between the viewer and the logger - example

PAUSING THE CONNECTION

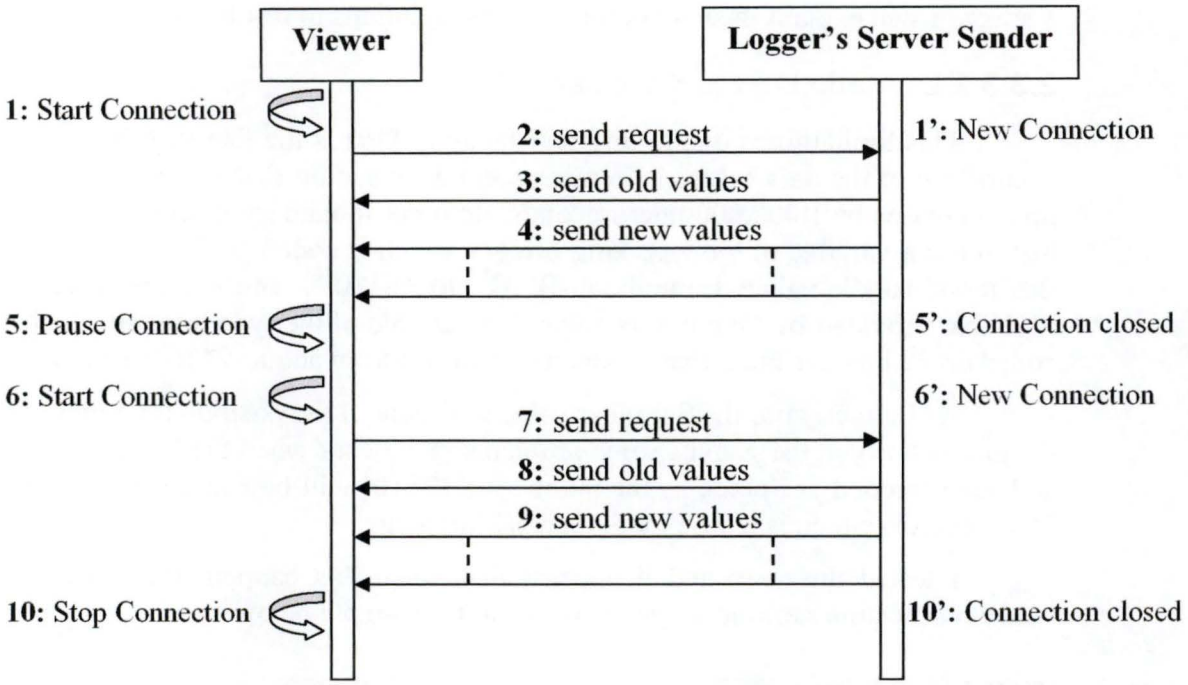
When we pause the connection, the physical connection with the logger is stopped and all the data values already in the viewer's memory are kept. Being able to "Pause" the connection was a good feature to implement because it allows the user to later restart the connection without having to wait to get all the old data values before getting the new ones, as only the newer values that are not yet in the viewer's memory will be sent.

STOPPING THE CONNECTION

When we stop the connection, the physical connection with the logger is stopped and all the data values already in the viewer's memory are erased. Thus if the user restart the connection later, all the data values will have to be sent again.

EXAMPLE

Below is an example showing all the states of the connection.



Comments:
1: The user decides to start the connection between the Viewer and the Logger.
2: The Viewer sends a request to the logger and provides the identifier of the Data Node it is displaying.
3: The Logger retrieves all the values it has for this Data (in its cache and/or in the database) and sends them to the Viewer.
4: Whenever new values arrive in the Logger, they are directly sent to the Viewer.
5: The user decides to pause the connection and all the data values already in the viewer's memory are kept.
6: The user decides to (re)start the connection
7: The Viewer sends a request to the logger and provides the identifier of the Data Node it is displaying **and** the time index of the last data value it has in memory.
8: The Logger retrieves all the values of this Data newer than the time index it has received (in its cache and/or in the database) and sends them to the Viewer.
9: Whenever new values arrive in the Logger, they are directly sent to the Viewer.
10: The user decides to stop the connection and all the data values in the viewer's memory are erased.

Figure 26: Connection between the Viewer and the Logger - example

2.3.3 Limitations of the Java Version

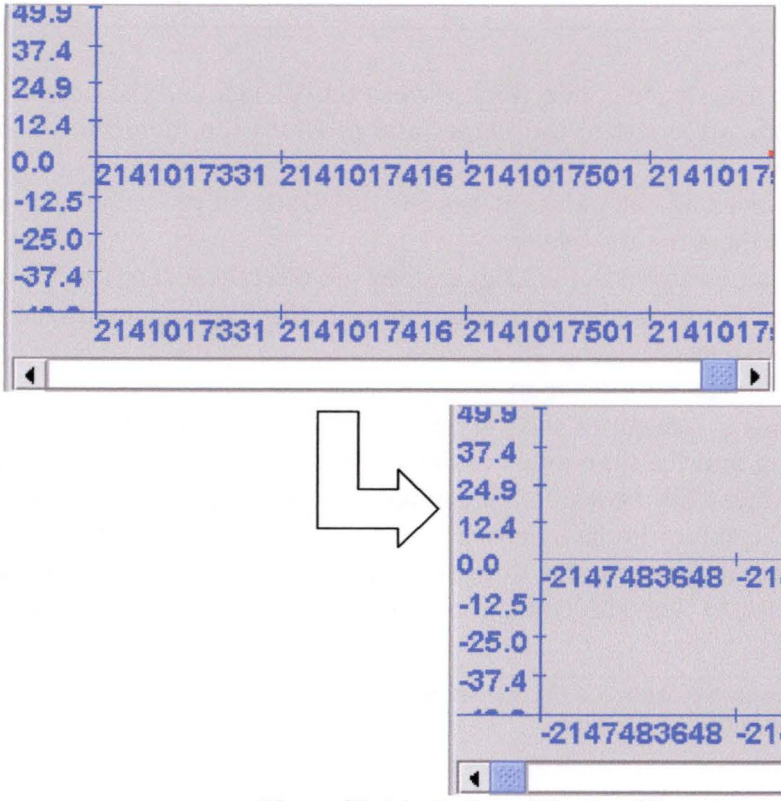
There are still some limitations to this version of the Network State Viewer. I will explain them and propose some solutions to resolve them.

2.3.3.1 Limitation of the Scrollbar

This limitation is due to different factors. First is the fact that the time of occurrence of the data values is in microseconds, meaning that in one second the time increases by 1.000.000 microseconds. In order to manage such big figures, I had to use a variable of the type **long integer** which is coded on 64 bits, meaning that it can handle values from about $-9*10^{18}$ to $+9*10^{18}$. In the other hand, the Scrollbar provided by Java has its value in a variable of the type **integer** which is coded on 32 bits, meaning that it can handle values from about $-2*10^{9}$ to $+2*10^{9}$.

As I directly use the Scrollbar value to decide of the position (and thus also the graduation) on the X axis, some problems may occur when the X axis scale is at 1 microsecond per pixel, as the number of $+2*10^{9}$ will be exceeded after only 2000 seconds which is a little more than half an hour.

I tested this case and it appears that when this happen, the scrollbar's behaviour becomes erratic, as you can see in the example below.



In the first picture, we are approaching the limit of the **integer** variable.

In the second picture you can see that we have looped to the smallest value of an **integer**.

Figure 27: Limitation of the Scrollbar - example

To resolve that problem, the easiest way would be to find an implementation of the Scrollbar that uses a variable of the type **long integer** for its value; that way we would never have this kind of problem again, or only after about 290.000 years.

2.3.3.2 Problems with Lines representation

This problem is due to 2 factors. First of all, as I mentioned in "2.3.1.2 Management of Data values : Function for managing the data values"[1], I use this function to get the data values I have to display at each given moment, meaning that I only ask for data values that are currently displayed on the graph. The second part is that, to display Lines, we need to have two points, and when the X axis scale is very small, we don't have usually many data values to display at the same time.

Thus it can occur that in some cases, when in Line representation, there is a data value that cannot be displayed, because it is the only data value in the displayed part of the graph.

[1] See at page 42

You can see in the example below that with two data values in the displayed part of the graph we can see where are those values with the line representation, but if we move the graph a little to the left, when there is only one data value, nothing is displayed in the line representation.

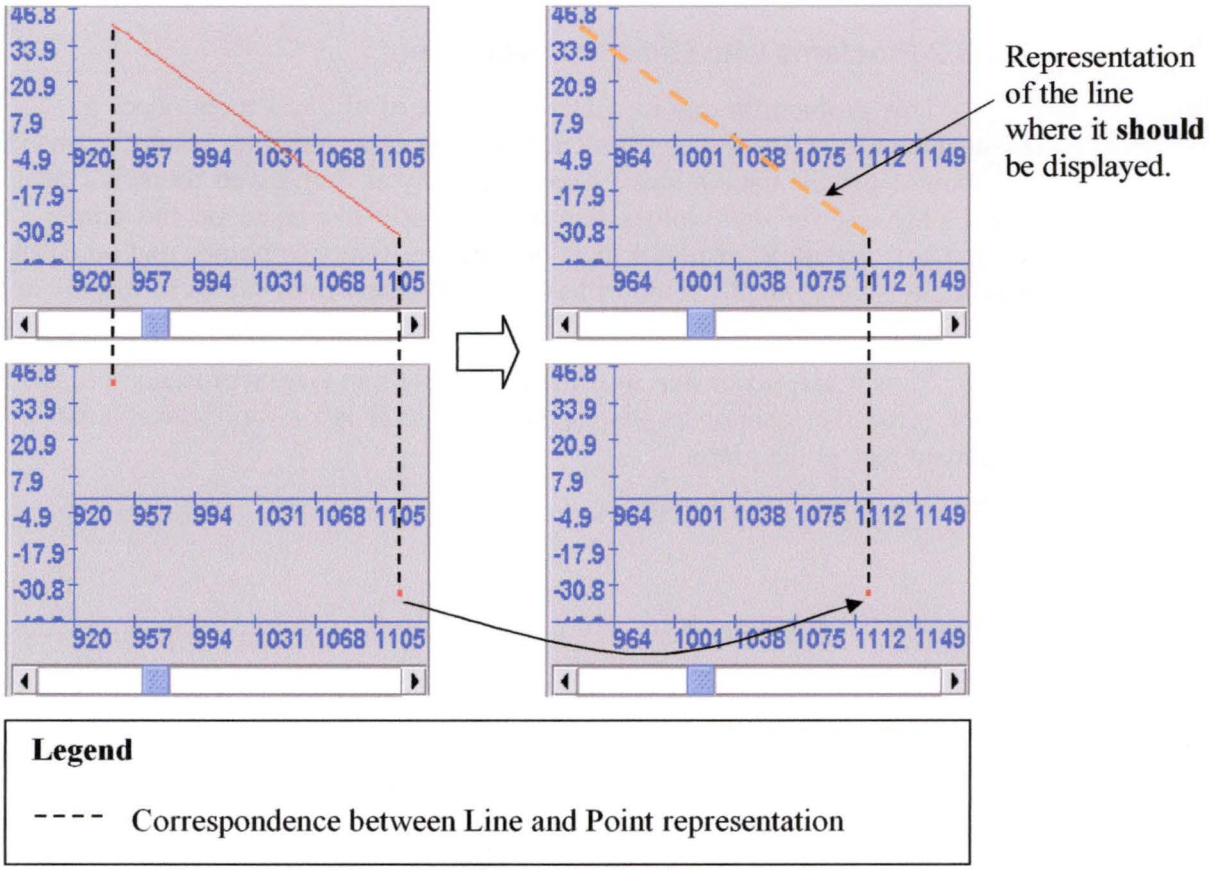


Figure 28: Problems with Lines representation – example

As a first solution, the "Function for managing the data values" could be enhanced to deliver one data value before and one data value after the displayed part of the graph, and that way, we would never have this problem again. But doing this could also increase the processing power needed to run the viewer, as providing one data value before and one after could be time consuming when those values are distant from one another.

Another way could be not to use the "Function for managing the data values" but to have one whole set of data values in memory for each possible scale of the graph. That way it would be very easy to get the data values needed for the display, but on the other hand, it would require additional memory and some processing power in order to maintain the sets of data values for each scale whenever new data values arrives.

2.3.3.3 Limitation of the connection with the Logger

As you may have noticed in "2.3.2 Connection with the Logger", whenever we start a new connection (without having paused the connection before) between the viewer and the logger, the entirety of the values of this viewer's Data node are sent from the logger to the viewer. That means that even if the viewer's memory capacity is too small to contain all the data values, the data values will be sent anyway. In some cases, this may lead to an important delay before the user is able to see the new data values arriving in real time.

To resolve that problem we should allow to specify one more parameter when connecting to the logger. This parameter would be a number representing the maximum quantity of old data values that we want to receive from the logger. Thus the logger would only retrieve and send that quantity of recent data values before sending the new data values.

2.3.4 Integration into the Human Interface

Because the Viewer was programmed as a standalone graphical Java component, the integration into the Human Interface was pretty easy. The only functionalities that had to be added into the human interface were to allow configuring the connection with the Logger, to allow showing / hiding the Viewer for each Data node and to allow starting / pausing / stopping the connection between the Viewer and the Logger.

In the following figure, you can see the window allowing to configure the connection with the Logger.

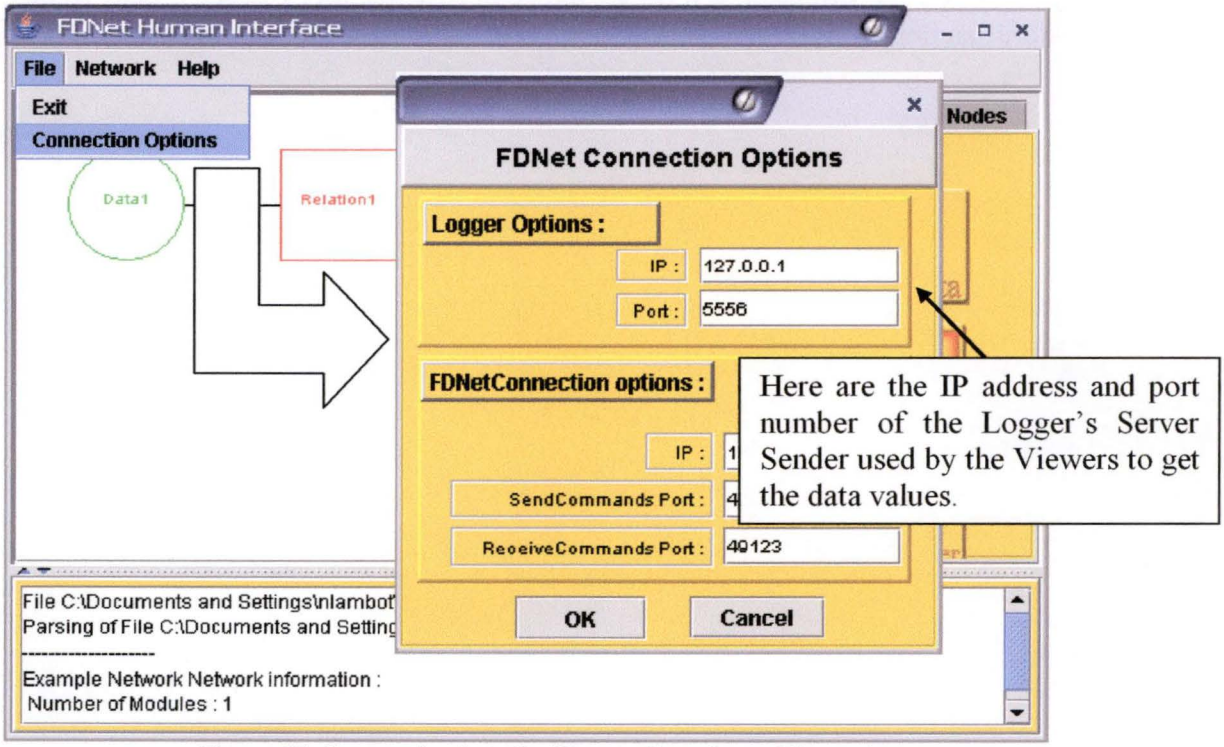


Figure 29: Integration into the Human Interface – Figure 1

In the following figure, you can see the graphical part of the Human Interface with the contextual menus accessible on the Data nodes and on their Viewer's representation.

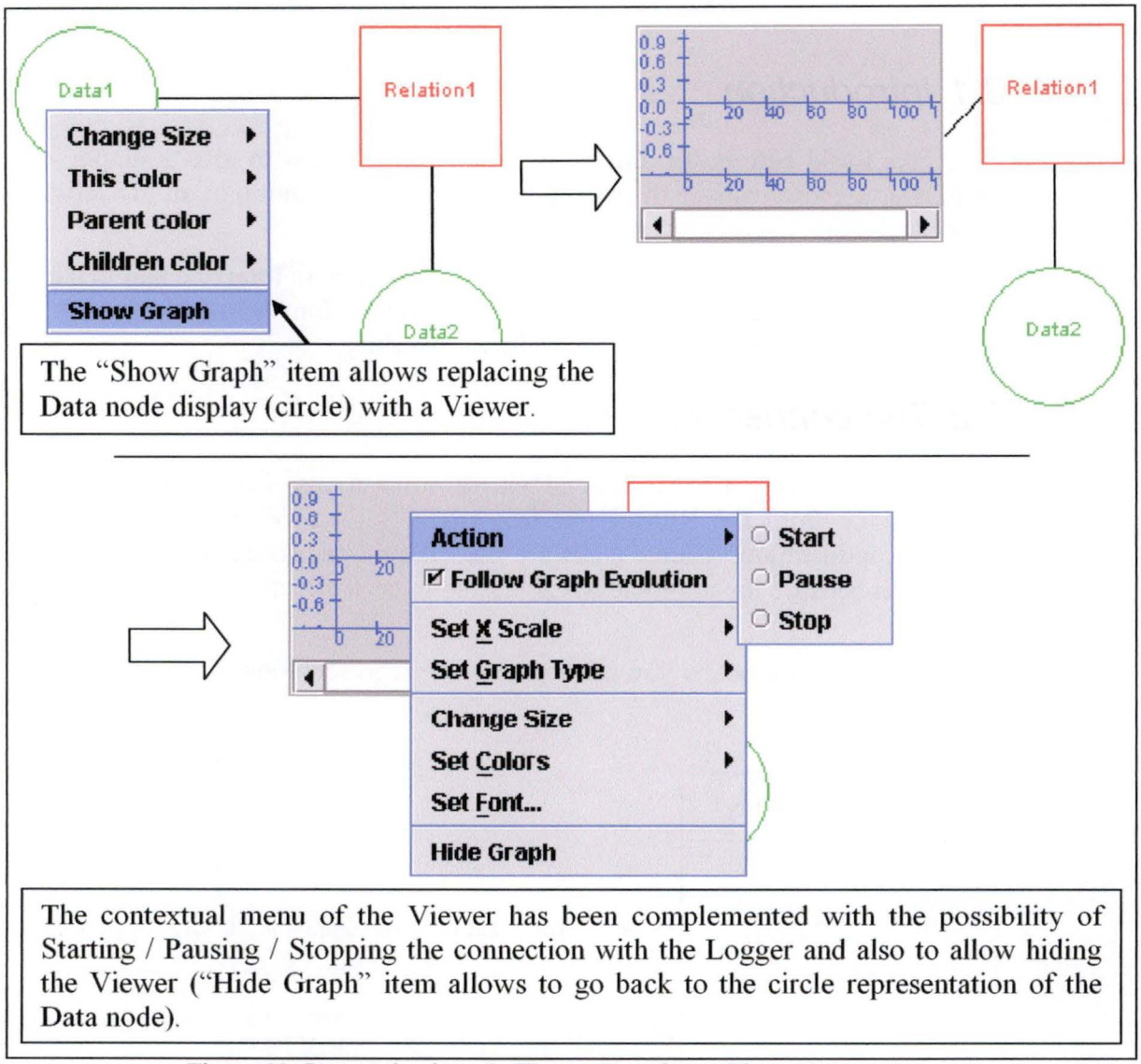


Figure 30: Integration into the Human Interface – Figure 2

3 Connection between Human Interface and FDNet's core

3.1 Introduction

As I said before, the aim of the Human Interface is to give a simple way for FDNet users to manage the robot(s) through an easy edition of an FDNetwork before and while the robot(s) are functioning.

Thus the aim of the connection between the Human Interface and FDNet's core is to allow to start an FDNetwork loaded in the Human Interface and to be able to interact with the FDNetwork while it is working.

3.2 The connection

The connection is subdivided into 2 connections; one to send network modifications from the Human Interface to FDNet's core and one to receive network modifications from FDNet's core. It was necessary to subdivide the connection because the information exchanged in each direction is not of the same type.

In the figure below you can see a simple representation of the architecture of the connection.

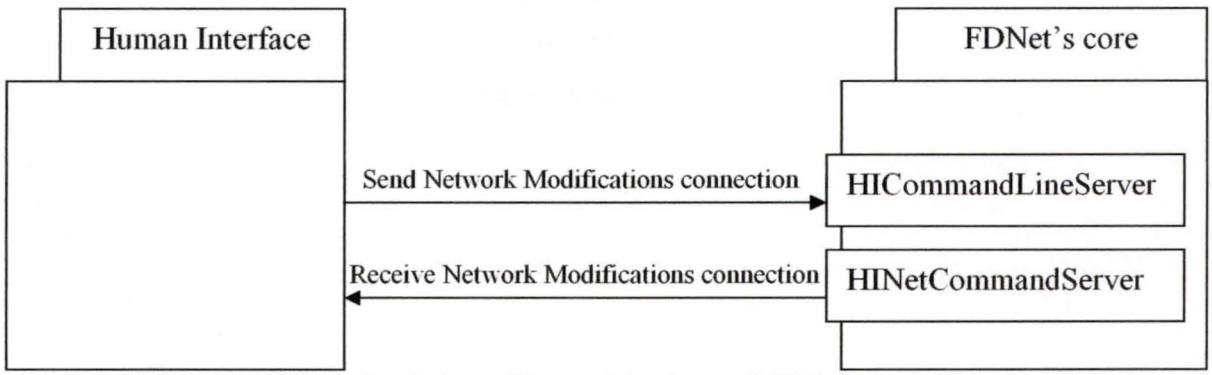


Figure 31: Connection between Human Interface and FDNet's core

3.2.1 Send Network Modifications connection

This connection is based on text command lines and it allows the Human Interface to send network modifications to FDNet's core. This connection is used to send the FDNetwork when we start the main connection and to send network modifications made by the user when the FDNetwork is working (and thus when the main connection is active).

The commands allowed are very simple as you can see in the figure below.

addData Name Host Path Class [arg1 [arg2 [...]]] Adds a Data node to the network.
addRelation Name Host Path Class [arg1 [arg2 [...]]] Adds a Relation node to the network.
delData Name Deletes a Data node from the network.
delRelation Name Deletes a Relation node from the network.
connect Name DataName RelationName Type Adds a connection between a Data node and a Relation node. The "Type" parameter allows specifying if it is a Reader ("R") or a Writer ("W") connection.
disconnect Name RelationName Type Deletes a connection between a Data node and a Relation node (but the name of the Data node is not needed to identify a connection). The "Type" parameter allows specifying if it is a Reader ("R") or a Writer ("W") connection.
set DataName DataValue Sets a manual value for a Data node.

Figure 32: Send Network Modifications – command lines

3.2.2 Receive Network Modifications connection

This connection is based on NetCommandEvent objects that are directly generated by FDNet's core and it allows the Human Interface to receive network modifications made by FDNet's core.

These NetCommandEvent objects can provide the same commands as in "Figure 32: Send Network Modifications – command lines" except that it does not provides Data node values, as those are only sent to the Logger.

3.3 Translating FDNetwork modifications

As there are some differences between the network of the Human Interface and the network of FDNet's core, I had to make a set of methods to allow translating Network modifications made in the Human Interface into something that FDNet's core could understand. In fact, the command lines shown in "Figure 32: Send Network Modifications – command lines" are the only commands that can be used to send the network modifications.

For the simple modifications like adding or deleting a Data or a Relation node, the translation is very simple, but for some advanced functionalities provided by the Human Interface, it becomes more complex.

I will now go through some of those differences and explain how I managed them. If you would like more information on the functionalities of the Human Interface I will introduce in the following points, I invite you to read the "Human Interface" part of Mr. Achbany & Mr. Jadoulle's thesis[1].

3.3.1 The Module extension

The module extension is one of the main functionalities provided by the Human Interface. It allows the user to subdivide an FDNetwork in different parts (named modules). Those modules are generally composed of Data nodes, Relation nodes and connections (Readers or Writers) that have a common goal; by example allowing the management of a specific sensor or a specific part of a robot.

For each module (except the default "main" module) the user can choose if the module has to be loaded when starting the FDNetwork (when connecting to FDNet's core), and once the FDNetwork is started, the user can activate or deactivate a module whenever he wants.

As the module extension only exists in the Human Interface, we have to translate the activation or deactivation of a module into something that FDNet's core can understand and thus, we can only use the commands provided in "Figure 32: Send Network Modifications – command lines".

By example to activate a module, we have to do the following things:

- Add all the Data and Relation nodes of the module
- Add all the Reader and Writer connections of the module
- Check for all the connections this module may have with the loaded part of the FDNetwork, in order to add them if necessary.

[1] [Achbany & Jadoulle 2004]

3.3.1.1 Example

You can see below a simple FDNetwork composed of two modules: the "main" module and another module named "Sensor1".

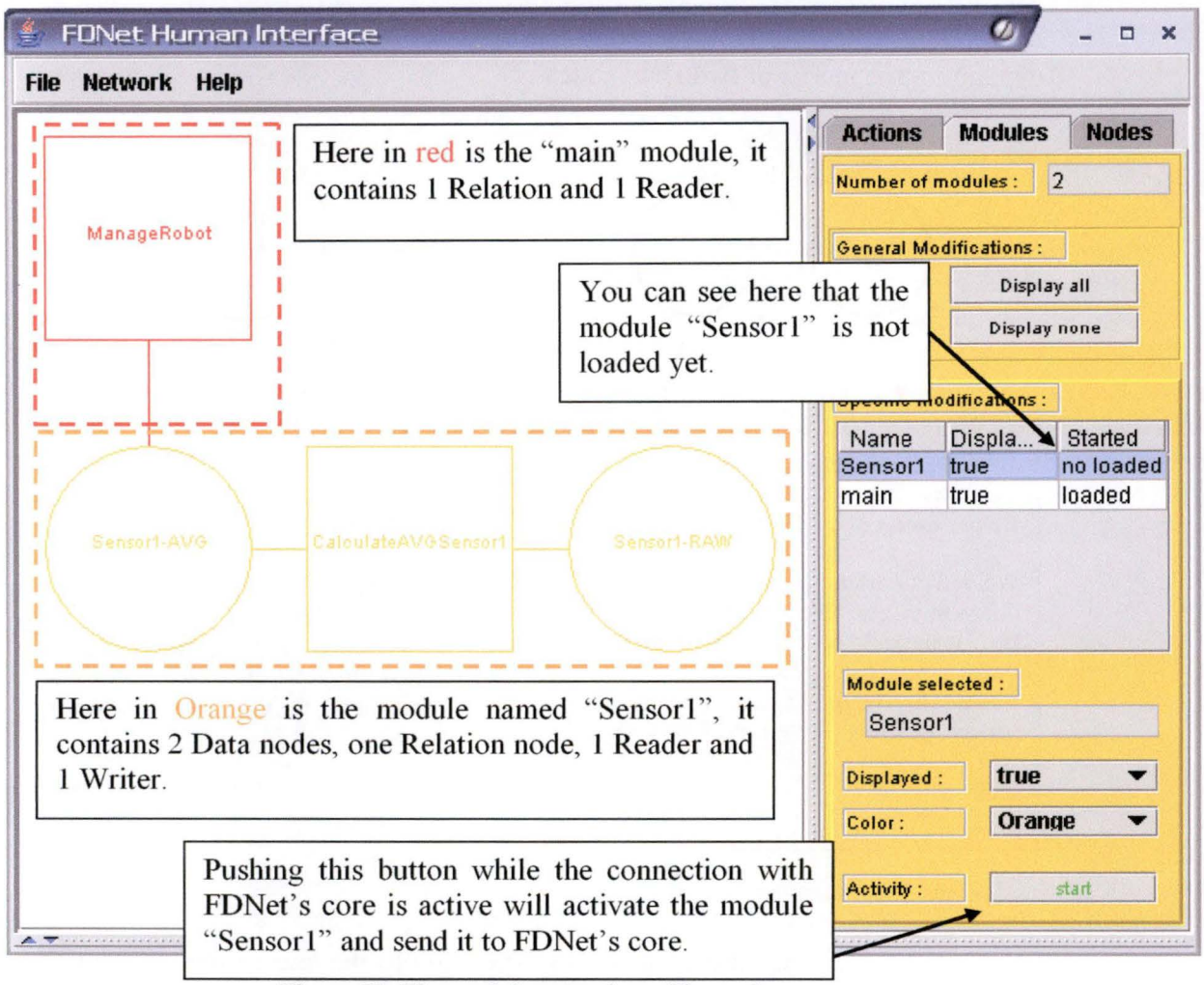


Figure 33: The module extension – Figure 1

And below are the command lines that will be sent to FDNet's core when we activate the module "Sensor1".

addData Sensor1-AVG ... ...
addData Sensor1-RAW ... ...
addRelation CalculateAVGSensor1 ... ...
connect readss1raw Sensor1-RAW CalculateAVGSensor1 R
connect writess1avg Sensor1-AVG CalculateAVGSensor1 W

At this point the whole module "Sensor1" has been sent to FDNet's core.

connect readss1avg Sensor1-AVG ManageRobot R

This was necessary to link the already loaded "main" module with our "Sensor1" module.

Figure 34: The module extension – Figure 2

3.3.2 Merging Nodes

Another functionality provided by the Human Interface is to be able to merge two Data or two Relation nodes together to create a resulting node having information of the two source nodes and their Connections redirected to it.

Below is a figure taken from the thesis of Mr. Achbany & Mr. Jadoulle, explaining the merge of two Data nodes.

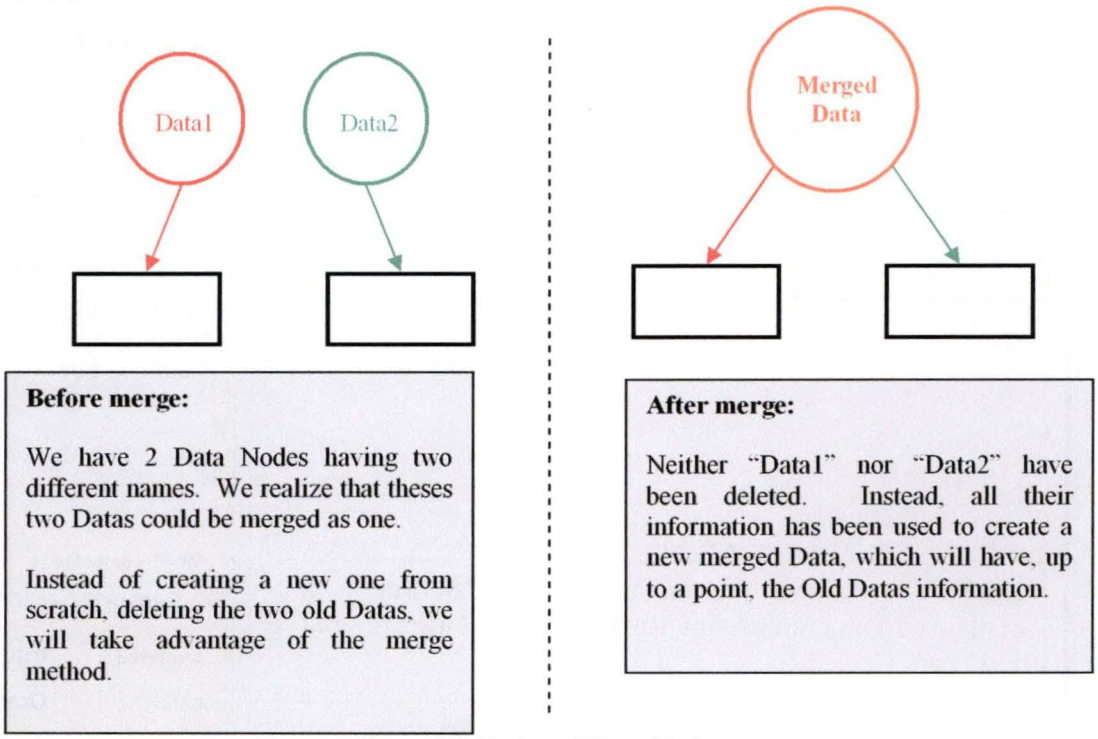


Figure 35: Fusion of Data Nodes

As you can see it only takes one operation to the user to merge two nodes together, but when the FDNetwork is working (when the connection with FDNet's core is active) it requires several command lines in order to send that network modification to FDNet's core:

- First we have to delete the two old nodes (when deleting a node in the FDNetwork on FDNet's core, all Connections with this node are deleted automatically, so we don't have to do it ourselves).
- Then we add the new node
- And finally, we have to retrieve all the Connections for the new node and add them.

3.3.3 Replacing Node

This functionality allows overwriting a Node with another one, deleting all Connections from the replaced Node and using only the ones from the Replacement one.

Below is a figure taken from the thesis of Mr. Achbany & Mr. Jadoulle, explaining the replacement of a Data node.

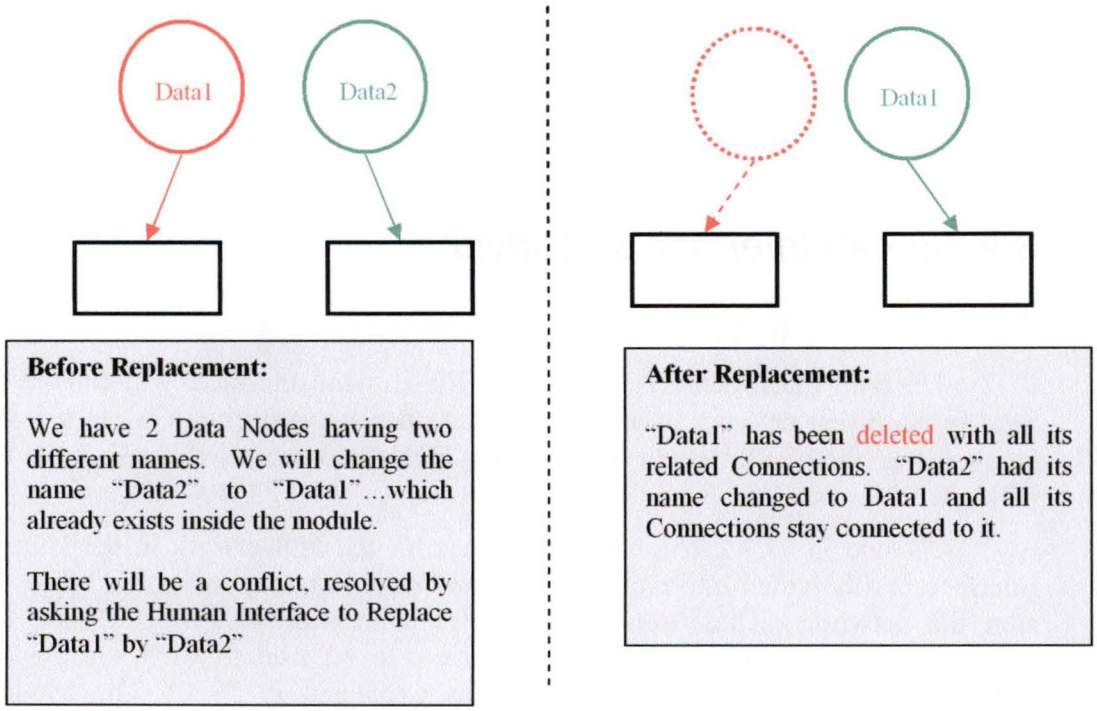


Figure 36: Replacement of a Data Node

As you can see it only takes one operation to the user to replace a node by another one, but when the FDNetwork is working (when the connection with FDNet's core is active) it requires several command lines in order to send that network modification to FDNet's core:

- First we have to delete the two nodes (when deleting a node in the FDNetwork on FDNet's core, all Connections with this node are deleted automatically, so we don't have to do it ourselves).
- Then we add the replacement node with its new name (and eventually new information)
- And finally, we have to retrieve all the Connections for the replacement node and add them.

3.3.4 Modifying Node

This is a really simple operation in the Human Interface, if a user simply modify the name of a node or any information it may have when the FDNetwork is working (when the connection with FDNet's core is active) it will require several command lines in order to send that network modification to FDNet's core:

- First we have to delete the Node as it was before (when deleting a node in the FDNetwork on FDNet's core, all Connections with this node are deleted automatically, so we don't have to do it ourselves).
- Then we have to add the Node with it's modifications
- And finally, we have to retrieve all the Connections with this Node and add them.

3.4 Human Interface's behaviour

3.4.1 Starting the connection

Once an FDNetwork is loaded into the Human Interface, you can ask to Start or Stop that network at any time. And once the connection is started, the Human Interface will try to connect to FDNet's core which will host the FDNetwork.

As said in "3.3.1 The Module extension", the FDNetwork in the Human Interface is subdivided into modules that can be set to be loaded or not when we start the network. Thus, upon a successful connection to FDNet's core, the "main" module and all the modules that have to be loaded when starting the network will be sent to FDNet's core (as explained in "3.3.1 The Module extension").

3.4.2 Connection activated

Once the connection is active, every modification made in the Human Interface on the FDNetwork will have to be accepted by FDNet's core before they can be applied in the Human Interface. If a modification is not accepted by FDNet's core, an error message will be displayed in the Human Interface and the modification is cancelled in order to keep the consistency of the FDNetwork between the Human Interface and FDNet's core.

There are also some functionalities in the Human Interface that can only be used when the connection with FDNet's core is active:

- As illustrated in "Figure 33: The module extension – Figure 1", the button allowing to Start or Stop a module can only be accessed when the connection is active.
- As illustrated in "Figure 30: Integration into the Human Interface – Figure 2", the Viewer can only be Started, Paused or Stopped when the connection is active.

3.5 Limitations of the connection

There are some limitations to this implementation of the connection. Firstly there can only be one Human Interface connected to FDNet's core at the same time. This is due to the fact that in FDNet, everything is at the same level, and thus a multiple connection would imply that any of the Human Interfaces that would be connected to FDNet's core could manipulate any part of the FDNetwork without any restriction. For this reason, and also because there was not much time left to implement the connection, this simple connection was realized.

Also, as there was not much time left to implement the connection, it was not possible to make tests in real situations with FDNet's core. For this reason and because there are differences between the FDNetwork in the Human Interface and in FDNet's core, it may be possible that some problem still exists in the connection.

Conclusion

At first, the work we had to do seemed very difficult, as it was not very easy to understand each other with the Japanese researchers because of the difference of language. We also had big difficulties to understand how FDNet (the base of our work) was functioning.

But with determination and through the retro engineering we made on FDNet's source code, we were able to progressively understand how it was functioning. Also, after some time, we became used to converse in English with the Japanese researchers and our relations became far better than in the beginning.

After this, we all worked on different parts. Mr Achbany and Mr. Jadoulle worked on the Logger and the Human Interface while I worked on the Dynamic Capabilities of the Human Interface.

The work I realized allowed me to learn many things in programming and also allowed me to work inside a team with Mr. Achbany, Mr. Jadoulle and the Japanese researchers. It has been a wonderful experience that I will never forget.

I think that my work, together with the work of Mr. Achbany and Mr. Jadoulle was an important improvement to FDNet and has allowed it to become more usable.

I hope FDNet will continue to evolve in the future, as I know this project holds a great potential for helping rescuers to save lives.

In conclusion I would like to say that I was honoured to be given the chance to work on such an interesting project.

Bibliography

Youssef Achbany & Jérôme Jadoulle, "About adding Utility and Usability to FDNet A Flat Distributed Network Architecture", 2004

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

Annexes

Annex 1: Retro engineering on FDNet[1]

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.

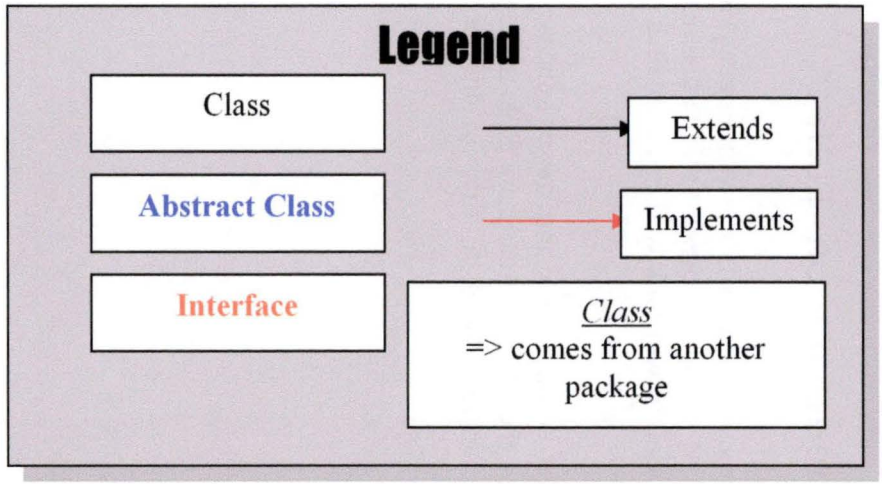
It also gives an idea of how FDNet works. By using theses schemes while trying to understand the code written by the I.R.S.I researchers gave us new ideas about what the code was doing and where to look to find an answer to our questions.

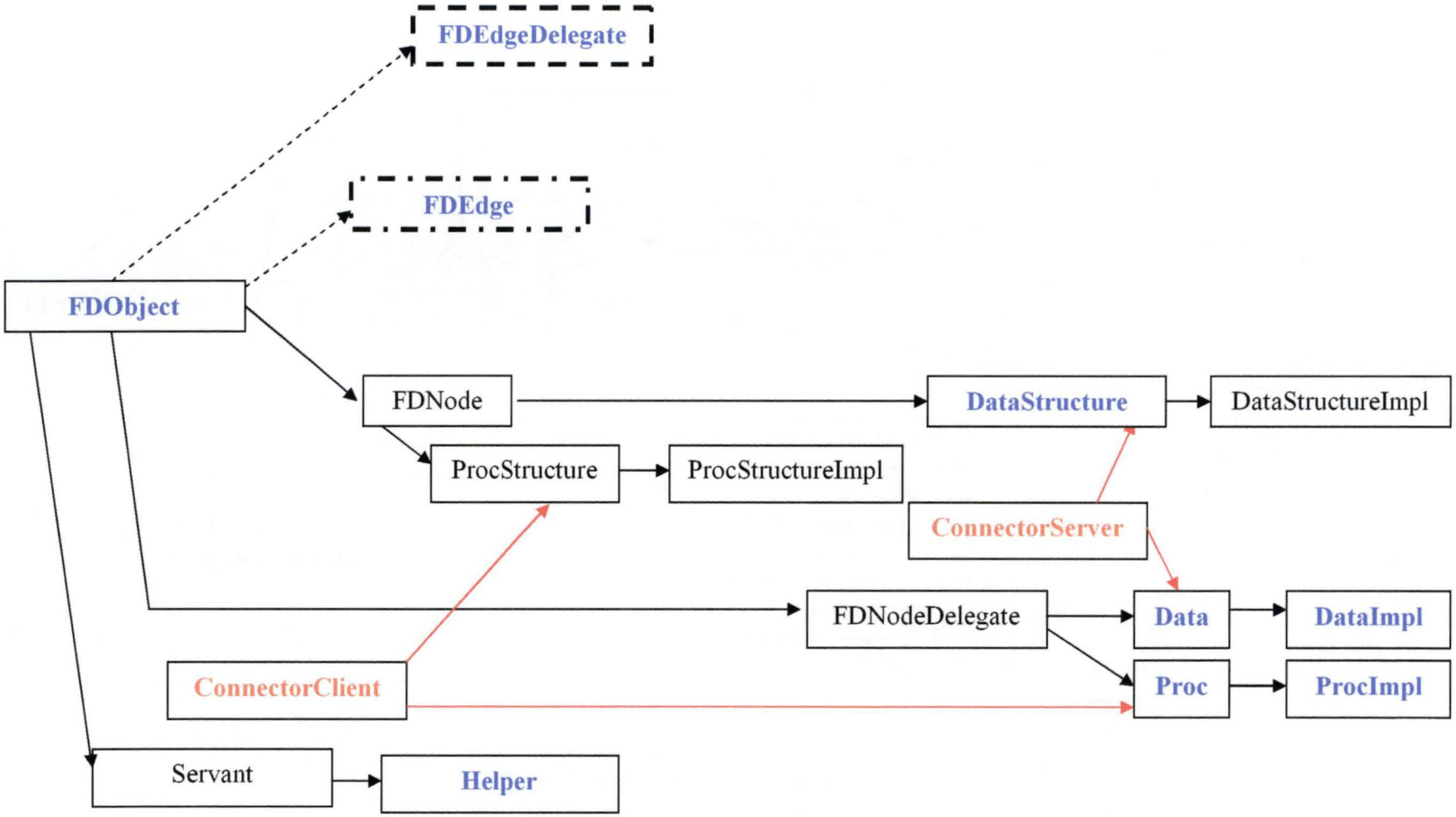
[1] This annex about « Retro Engineering » comes from the thesis of Mr Achbany & Mr Jadoulle [Achbany & Jadoulle 2004], as the three of us did this work together.

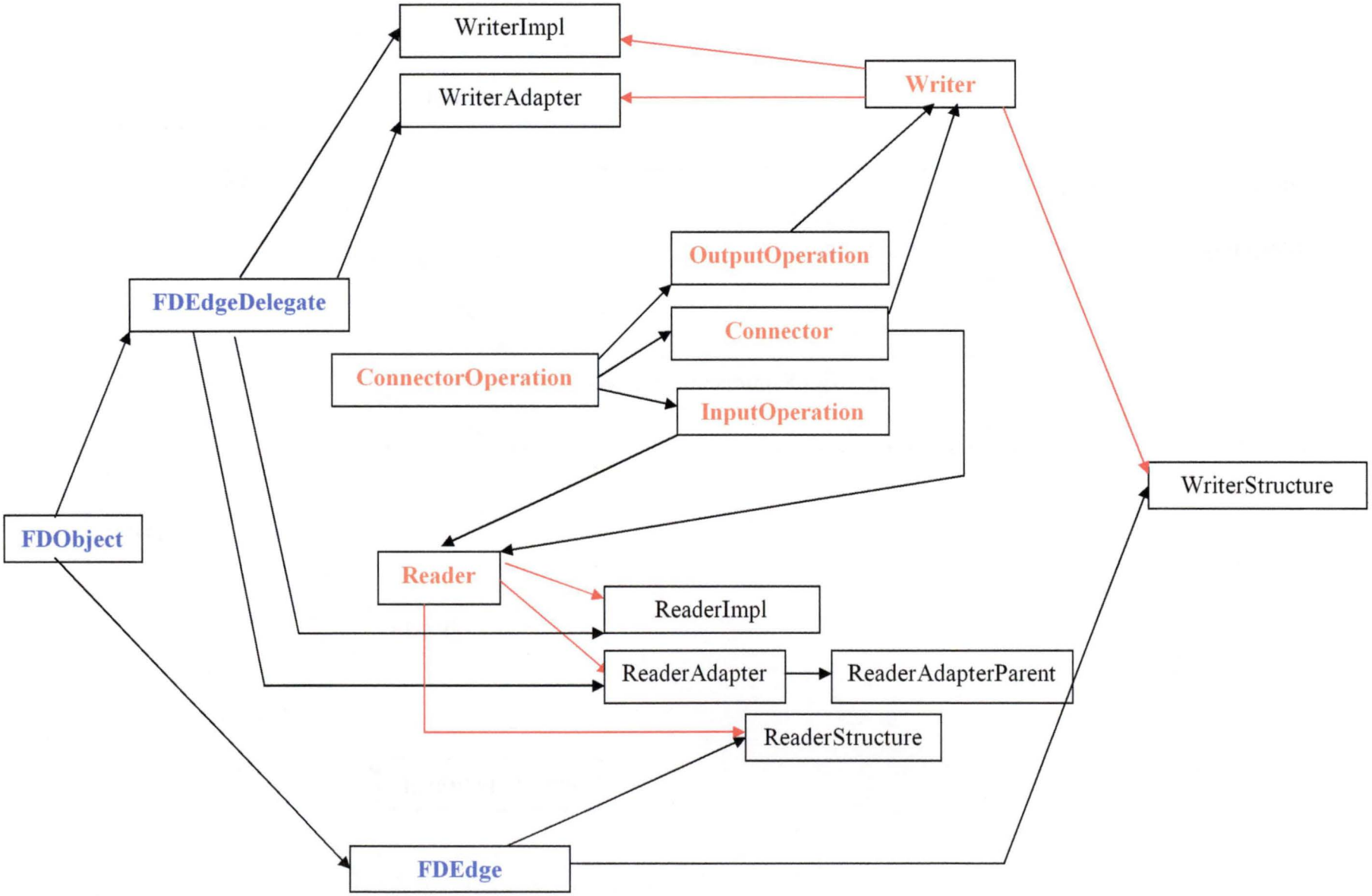
This is the classes hierarchy of the "cnet.core." package of FDNet project.

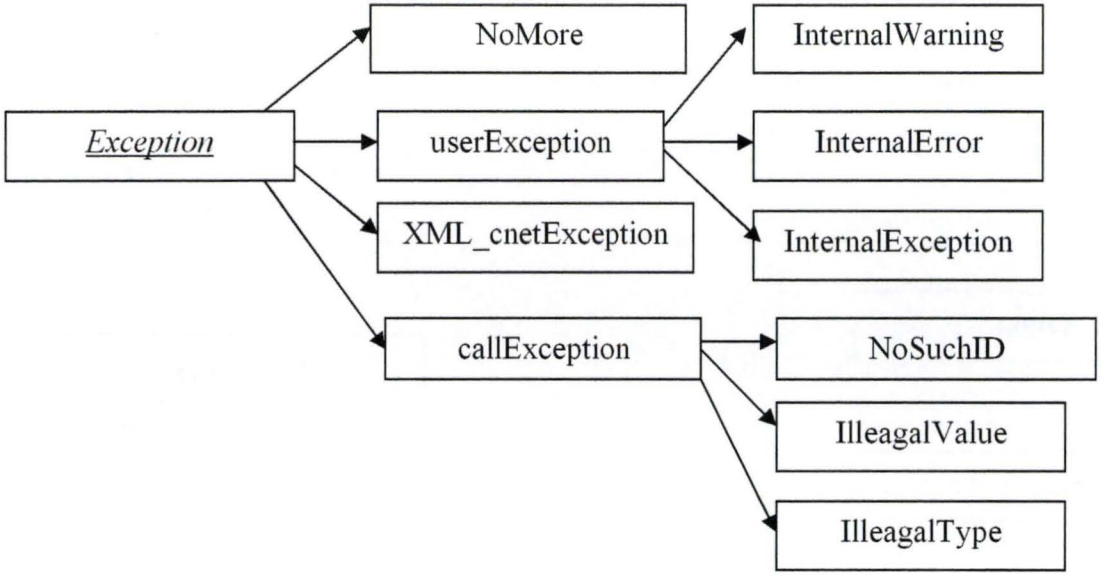
As written just right in the legend :

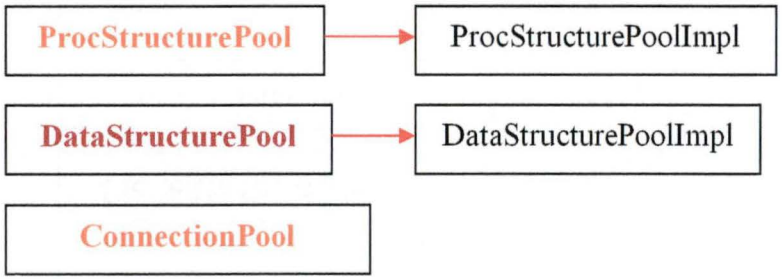
- the classes written in blue are abstract;
- the interfaces are written in orange;
- a class/interface extending another one will have a black arrow arriving to itself.
- a class/interface implementing another one will have a red arrow arriving to itself
- underlined and italic text means that the class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

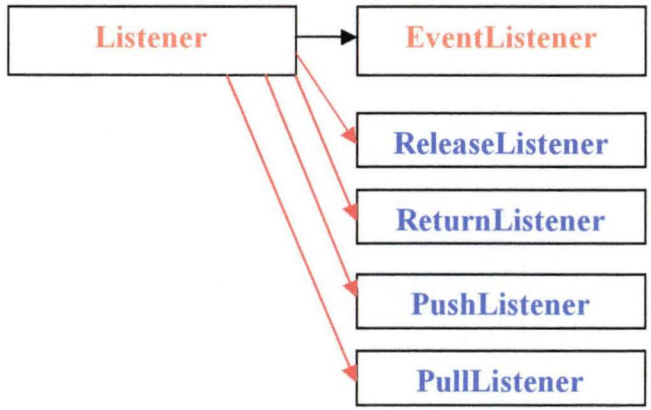


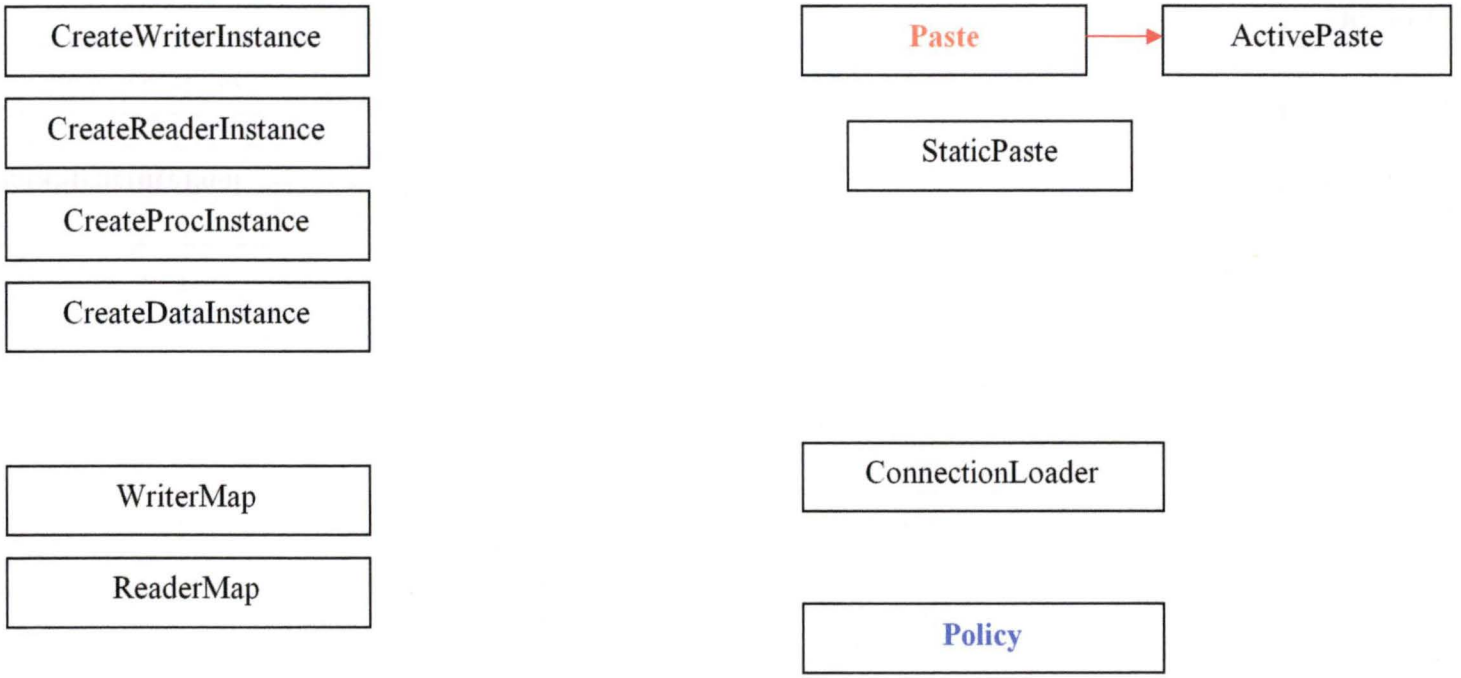

FDEdgeDelegate
FDEdge
FDObject
FDNode
DataStructure
DataStructureImpl
ProcStructure
ProcStructureImpl
ConnectorServer
FDNodeDelegate
Data
DataImpl
ConnectorClient
Proc
ProcImpl
Servant
Helper


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

Annex 2: Retro engineering documents[1]

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).

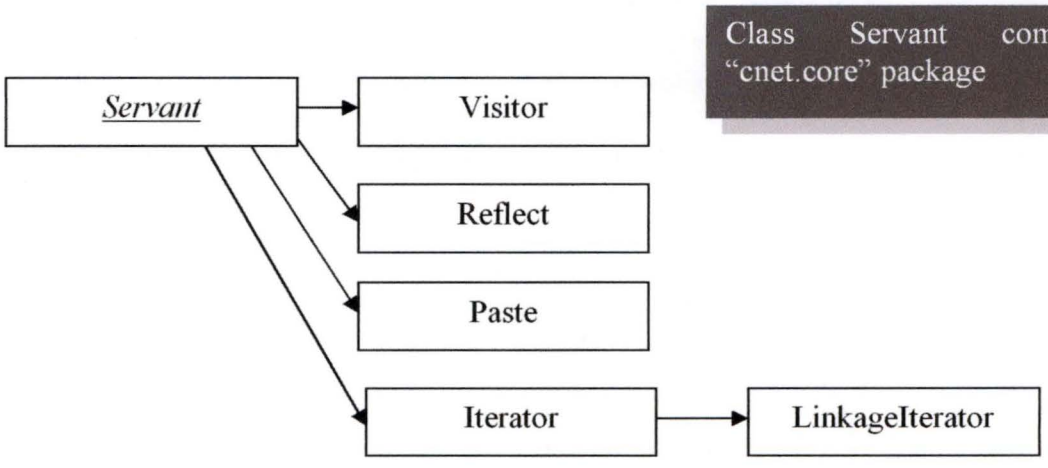
As a snapshot, the information it contains is not especially true. Most of what is written comes from the understanding we could have of FDNet's way of working. It thus means that it can still contain errors or information that is too vague to have a real meaning.

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.

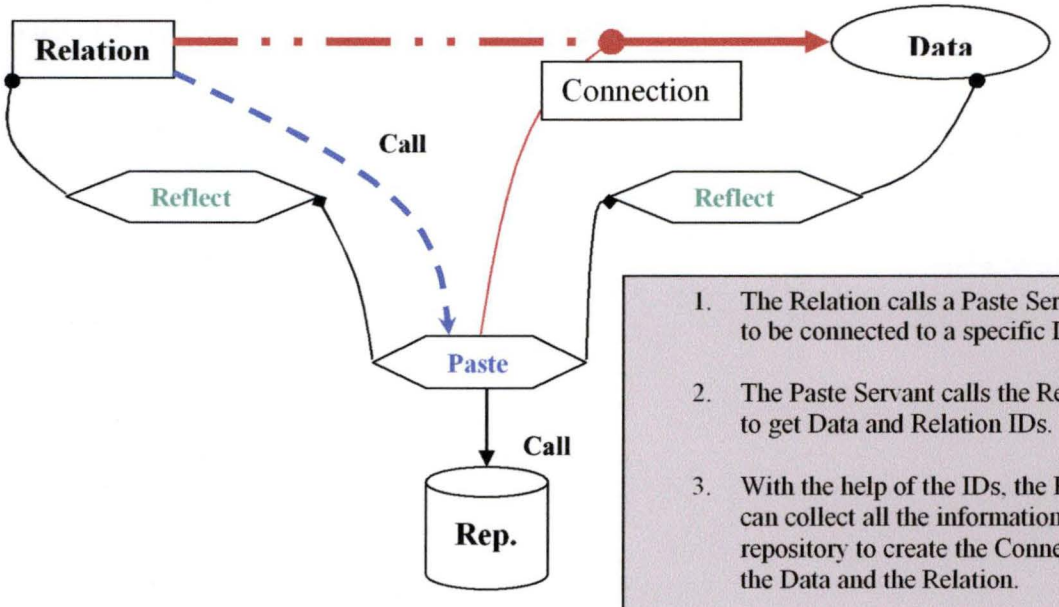
[1] This annex about « Retro Engineering » comes from the thesis of Mr Achbany & Mr Jadoulle [Achbany & Jadoulle 2004], as the three of us did this work together.

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



Class Servant comes from "cnet.core" package

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.