5 Implementation

The following chapter describes the techniques and actual code used in creating NTS. As NTS is in three parts: ntsRouter, ntsLibrary, and ntsMaster, each part will be dealt with separately.

Note: some of the classes being big, in UML diagrams, only the most important aspects of the class are shown.

5.1 ntsRouter

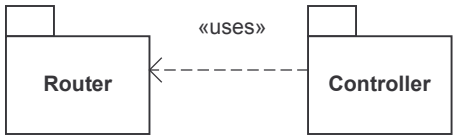

Figure 22: ntsRouter packages

As shown in Figure 22, the ntsRouter program is composed of two main parts. A set of classes to represent the Router and its Routing Table and a C style main which acts as a controller. The Router Package was initially implemented by Olivier Thonnard (see AOM description), and has only been modified to allow the setting of the "silent" working mode for the Router program (see section 4 Using NTS).

5.1.1 The Router Object

The main class is the *router*. The main data of this class is the Routing Table, which is an array of Complete Routes (initially: maximum 20 routes). This Complete Route is defined as a structure holding the following information:

- Destination: integer holding a TID,
- Path Cost: integer holding the number of hops to reach the destination,
- Next Hop [10]: an integer array of size 10 to hold the different alternative routers for the same destination,
- Sequence: integer to keep track of the sequence for the load balancing (points to the following route that has to be used),
- Status: a Boolean to know if the router has been changed or not (for the update process, to detect which routes have to be sent or not),

- Link or client: a Boolean that describes a link/route as a link (to another Router) or a client.

The other data's of the class *router* are: the **RouterID** (i.e., the TID), the **Size_RoutingTable** and the **DefaultRoute**. The class provides a set of methods to use the routing table:

Methods	**Functions**	**Comments**
Constructor	Router(**int** RouterID)	// Initializes the data & the RouterID
Methods for Initial Config (used by the Master)	Addlink(**int** neighbourID, **bool** LinkOrClient)	// Static config
	DeleteLink(**int** neighbourID)	
	SetDefaultRoute(**int** neighbourID)	
Methods for Dynamic Config (used by the Master)	AddlinkDyn(**int** neighbourID, **bool** LinkOrClient)	// Dynamic config // Uses Route Poisoning
	DeleteLinkDyn(**int** neighbourID)	
Methods for Dynamic Updates (with other Routers or with the Master)	**Packet** SendUpdate()	// Main functions for updates – use the Split Horizon principle – return a **Packet** object with Messages that need to be sent
	Packet ReceiveUpdate(**route** a)	// a **route** is a small object with: - a destination, - a path cost - a next hop - a link or client identifier
Methods for Routing	**int** GetNextHop(**int** destination)	// Uses the Load Balancing principle & Returns the next hop
Methods for Getting/Printing the Routing Table	**CompleteRoute*** GetTable()	// Returns the RT
	PrintTable()	// cout the RT

Table 2: Description of the "Router" class

Figure 23: Classes view of the Router Package

The interaction with the Router class happens through interface objects, namely:

- The class **route**: very small object with 4 integers (dest, path_cost, next_hop & linkorclient),
- The class **Message**: object to represent an "envelope" of routes that need to be sent to a certain neighbor,
- The class **Packet**: object to wrap all Messages generated by calling the Updates (ReceiveUpdate or SendUpdate) into one single packet.

Of course, one can easily see how dynamic are the sizes of the Messages and hence the size of the Packet as we can never predict how many routes a Router will need to exchange. All the classes are represented in Figure 23.

5.1.2 The Controller

As already stated, the ntsRouter program is an infinite loop, once it has been setup, which listens to incoming messages and processes them. Upon instantiation by the Master, the Router synchronizes its computer clock with that of the Master's (using PVM) so that timing records will be accurate. The Router then waits for the ok from the Master that all Routers have been initialized properly, and the simulation starts.

Initially, the Master will send to each Router its initial neighbors via configuration messages, and the Routers will automatically propagate the changes in between themselves to update the Routing Tables. Also each Router will receive the options currently in use.

Routers receive the following messages:

Type	Sender	Tag	Process
Update	Another Router	1	Open the message and update the Routing Table with the data, if the Table is updated, propagate the change
Add Neighbor	Master	5	Adds a new entry to the Routing Table and propagates the change
New Client	Client	13	A client wishes to join the simulated network, a new entry in the Routing Table is added, a message is sent to the Master, the ok is sent to the Client that it has been added, the change is propagated
Delete Neighbor	Maser	3	Removes the entry from the Routing Table, initially the Route is Poisoned first, then removed at next update
Delete Client	Client	2	Client leaves simulated network, Route is Poisoned and propagated, Master is notified
Die	Master	6	Message for Router to end, Master takes care of signaling all neighbors before it removes a Router.

Print Table	Master	10	Router prints its Routing Table to the standard output
Forwarded Message Logging	Master	11	Option to log all messages that are forwarded by sending the log to the Master
Message Printing	Master	12	Option to print messages to standard output or not while running
Forward Message	Client	Any other	Processes the message to forward it on through the network

Table 3: ntsRouter messages

To forward a message, the message tag of the incoming message must first be processed because of the information compression needed by NTS. The tag is extracted, and divided into its components. The Final Destination of the message is looked up using its Id in the PVM lookup table. The TID now accessible, the Next Hop can be looked up in the Routing Table, and the message forwarded by exchanging PVM buffers.

For logging of Messages, the time of processing is checked, and packed into a message for the Master along with all the needed information for the log: source, final destination, current location, next hop, message tag, sequence, etc.

5.2 ntsLibrary

The ntsLibrary is the overloaded PVM communication library proposed to Clients to communicate using the simulated NTS network. It is mad of two parts: the overloaded PVM functions and a settings package (Figure 24), which connects to the NTS network and stores all the details for overloading PVM.

Figure 24: ntsLibrary packages

5.2.1 Settings

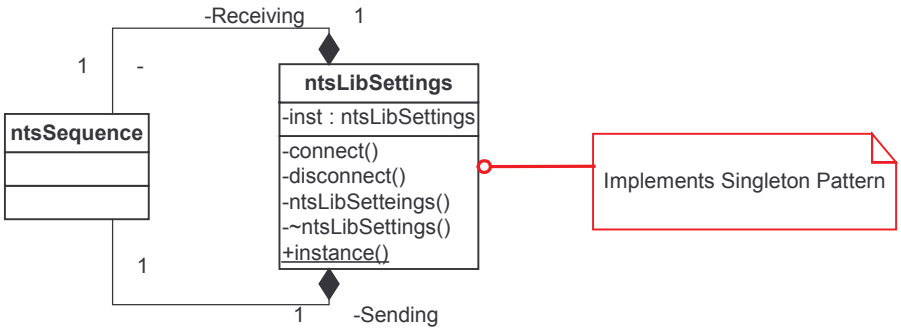

Figure 25: ntsLibSettings class

The Settings is composed of a class *ntsLibSettings*, which implements the singleton pattern (Figure 25). The instance of this class is called in every overloaded function as upon creation, this class connects to the NTS network:

The class contacts the ntsMaster, asking the TID of the local Router for the code. If a local router exists the class contacts it, to ask to be added as a subscriber to the simulated network.

The ntsLibSettings keeps all the relevant information for the client to be able to run on the NTS network, such as the Local Router TID, and the sequence for each destination and source the client code sends or receives messages from or to.

5.2.2 Forwarding Functions

Not all PVM functions are overloaded. Many in fact are just forwarded on to their PVM equivalent. An added layer of error detection is added by NTS though. Also the instance of the ntsLibSettings is called to make sure that NTS is connected.

5.2.3 Sending Messages

PVM has many ways of sending off messages, the most simple being *pvm_send()*. NTS implements a “sender” function which takes care of redirecting the message to the local router and modifying the message tag to contain all the required information for the working of PVM (see section 3.6 ntsLibrary). Any needed information is taken from the *ntsLibSettings* class. Each overloaded PVM function calls the NTS sender with the required parameters, or calls it multiple times as with Group Sending, or multicasting.

5.2.4 Receiving Messages

Unlike for sending messages, NTS does not implement a receiver as each type of specialized PVM receive is different. Each overloaded PVM receive function therefore takes care of composing the composite message tag that needs to be listened for using the information from the ntsLibSettings. As explained in the Design part, NTS must "listen" for a specific message in order to reorder the sequence of the messages automatically.

5.2.5 Disconnecting

Disconnecting happens automatically at the destruction of the *ntsLibSettings* object. Disconnection happens therefore automatically at the end of a NTS application, but it is better to explicitly call the *nts_exit()* function when the NTS connection is no longer required.

5.3 ntsMaster

The ntsMaster being a GUI program, it is mostly based on the **View Control Data** (VCD) method. In this method a central Data repository can only be modified via a Control package. Various views can be used to display the Data in the repository and ask the Controller to change the Data. This method reduces coupling in between Packages and provides central accesses and control over each. Each Package also becomes independent from the other, only the Control is aware of every View viewing the Data at any one time. Each view operates independently, usually using the Observer pattern with the Data to update the contents of the view when the Data has been changed. Figure 26 exposes the packages of the ntsMaster.

As the Qt Designer method was used for the main window layout the VCD method was not completely implemented. The Qt Designer implements the control of an application directly into the main window of the application, therefore combining Graphical Display and Controller together. This method though makes it easier and faster for small layout changes. The Qt Designer method in combination with the adopted VCD method also makes it easy to add/remove/modify extra views of the Data structure as adding a new graphical module means just adding it to the main window rather then to the Control module and the Main Graphical representation. More of this method is presented later.

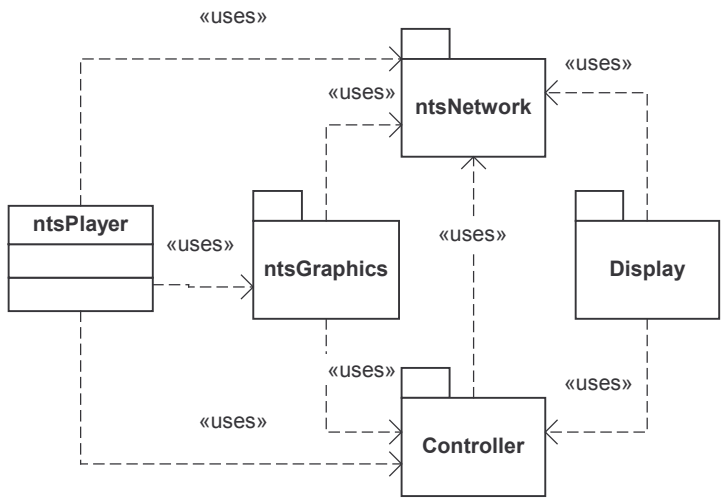

Figure 26: The main packages of the ntsMaster and their connections

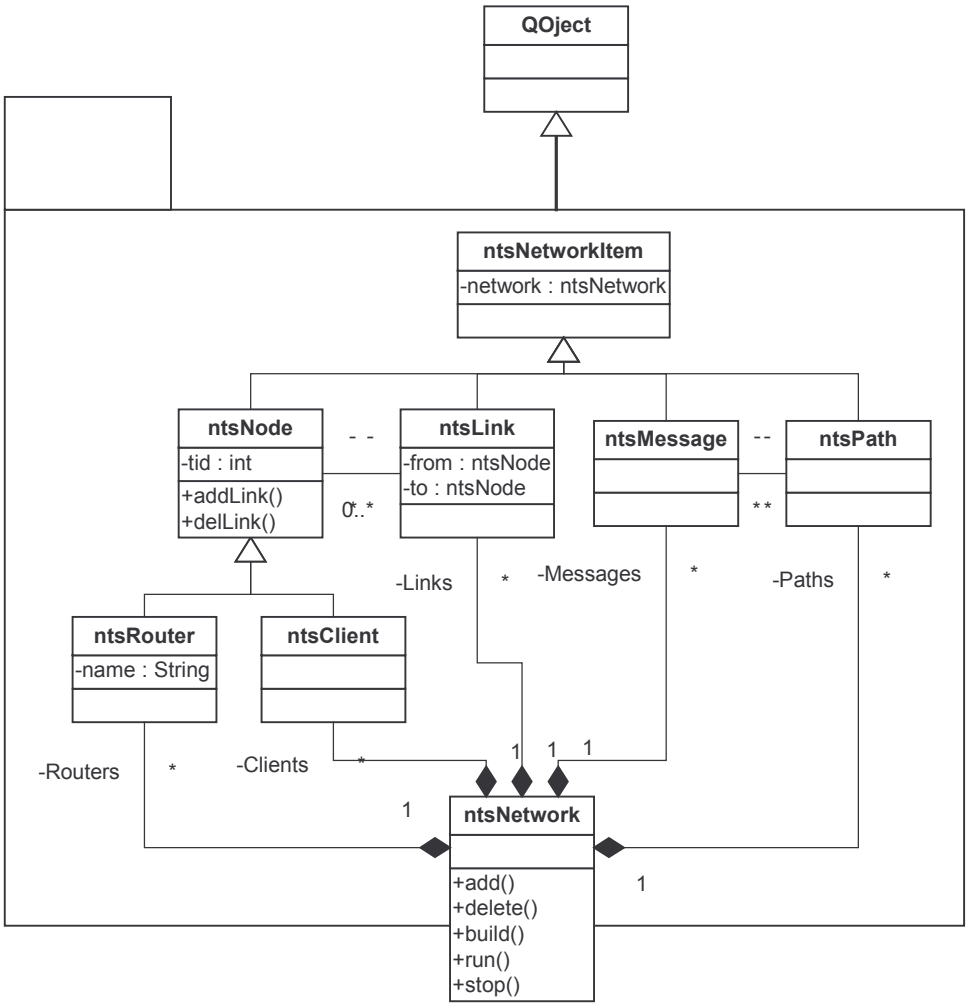

Figure 27: ntsMaster, ntsNetwork Package / data structure classes

5.3.1 ntsNetwork Package

The ntsNetwork package is the main data structure of the ntsMaster program and the Data of the VCD method implemented (Figure 27). The whole package inherits QObject so that the rest of the program can "observe" the data structure via the Qt signal/slot system. The data structure of course represents the real structure of the simulated NTS network. A façade is presented as the ntsNetwork class.

NtsNetwork class:

The ntsNetwork class is the front end of the data structure or the façade of the package. No classes from the actual structure should be instantiated by user code; the creation and network modification functions of the ntsNetwork should be used. These make sure that the structure is modified according to the rules of NTS network simulation, plus if a new class is created, keep its reference for easy finding. The class also holds all the set options of the NTS network.

The ntsNetwork class is also responsible for the state of the NTS network, whether it is running or not. It has the *build*(), *run*(), and *stop*() functions to control the state of the network. The *build*() function instantiates each router, passes the options to these, and connects the neighbors, turning the network into a "running" state.

The *run*() function is actually a separate thread, which listens to messages from Routers and potential Clients. The *stop*() sends the terminal signal to all Routers thus ending the simulation.

The ntsNetwork class also is full of non-shown navigating functions for Observers to navigate the data structure and retrieve the needed information. Observers link to this class to be notified of changes to the Data structure. The ntsNetwork class emits specific signals whenever the data structure is modified so that Observers can update only when a specific part of the structure they are interested in is modified.

Network Items:

The ntsNetworkItem class identifies any class belonging to the data structure. It is the super class, which is used by the Controller Package to select items directly in the structure. The subclass of a network item can be identified by its overloaded *type*() function. Network items always hold a reference to their network.

Graph Network Structure:

The *ntsNode*, *ntsClient*, *ntsRouter*, and *ntsLink* classes represent the structure of the NTS network being simulated by a user. This part of the structure is based on a graph design as a network behaves a bit like a graph. The two types of Nodes: Routers and Clients (terminal Nodes, i.e., can only have one link) are "linked" together by *ntsLinks*. In the real network, these links are exchanges of TIDs.

The usefulness of the graph structure is the way independent functionality can be controlled; such as the fact a Link attached to a Client must always be bi-directional. Also with the graph structure cascading deletes are easy to implement; as in when a Router is deleted, all of its connecting Links are deleted and each neighbor notified automatically of the departure of one of their connections.

The network structure takes care to forward any changes to their instantiated programs (i.e., ntsRouter programs running on computers of the simulated network) if the network is in running mode, automatically sending the appropriate messages for the change to be reflected in the simulated network.

Messages and Paths:

The *ntsMessage* class is a container class to hold the information sent by Routers for forwarded messages logging.

Paths are classes, which hold lists of ordered Messages, so as to recreate the full forwarding path of a message through the NTS simulated network. Paths can then be played out by the *ntsPlayer*.

5.3.2 Controller Package

Because the ntsMaster is a Graphical User Interface, the Controller mostly forwards commands input by the Graphical User Interface to the appropriate subsystem of the application, mostly to the Data Structure. The Controller is the only Package, which is allowed to forward requests that modify the structure of the Data. Figure 28 depicts the process of forwarding a request to the Data Structure and the resulting update propagation

Plug-ins:

As stated before because of the Qt Designer method used to simplify the layout of the main window graphic, the Controller is contained within the class that represents this

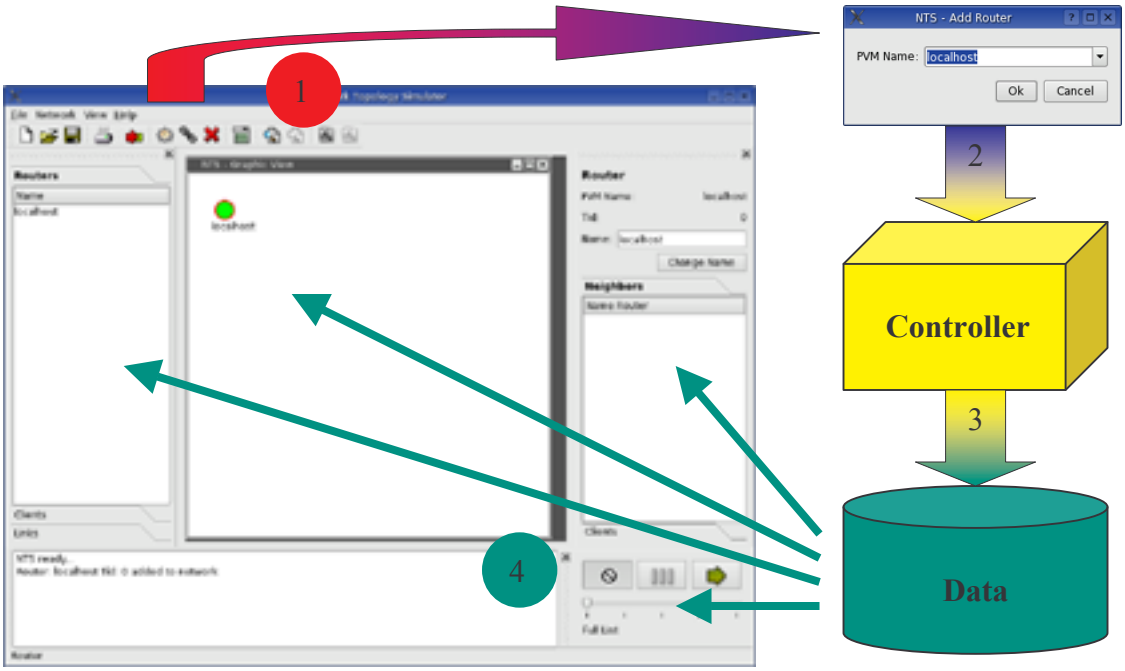

Figure 28: The forwarding process of adding a new Router

main window graphically. This simplifies the adding or removal of a subsystem from the application, as only the one class has to be modified.

Adding a subsystem to NTS compromises: instantiating the new subsystem in the Controller initiation (passing a reference to the Data structure so the subsystem subscribes as an Observer), connecting the Controller and subsystem via signal/slot references for the appropriate functionalities to be forwarded (usually the **select** mechanism, see section 5.3.2 Controller Package, Selection), and modifying the Main window graphic to display the new subsystem accurately. All of these modifications are carried out only to two functions of the Controller class because of the melding of the Graphical component and the Controller. None of the other subsystems or the Data structure need be aware of the new functionality added to the program.

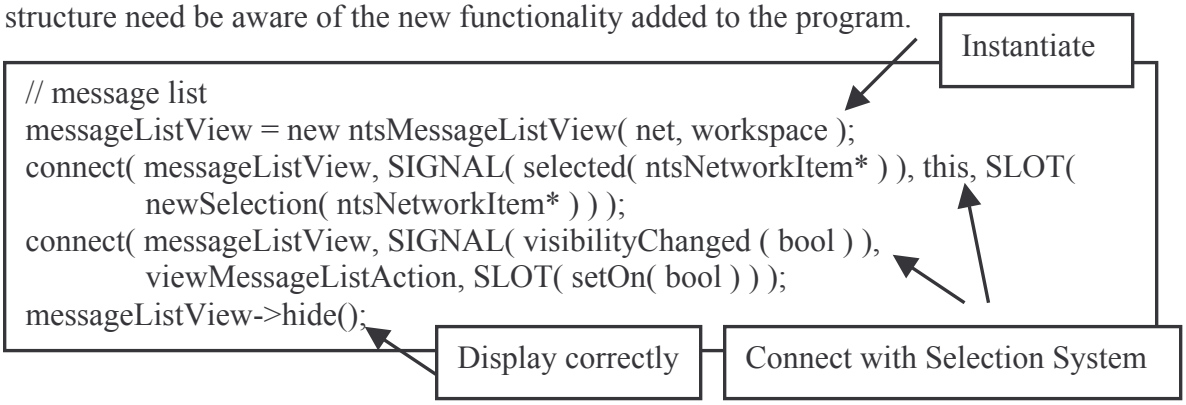

Figure 29: Instantiation and connection of the Message List

Figures 29 shows the actual lines of code needed to instantiate the Message List subsystem, and connect it, in the initiation of the controller.

Selecting:

One of the main advantages of using the VCD method is the ease with which one can cascade a change through all the various views automatically. The main example of the use of this technique is the selection process of NTS. Many views within the NTS Master can select various components of the network structure. When an element is selected, view emits a "selected" signal with the selected element as argument. The Controller "selection" method being linked with the view (see section 5.3.2 Controller Package, Pug-ins) emits its own "selected" signal. Each plug-in which is connected to the Controller "selected" then updates its view, looking up the new selected element's attributes directly from the data structure (Figure 30).

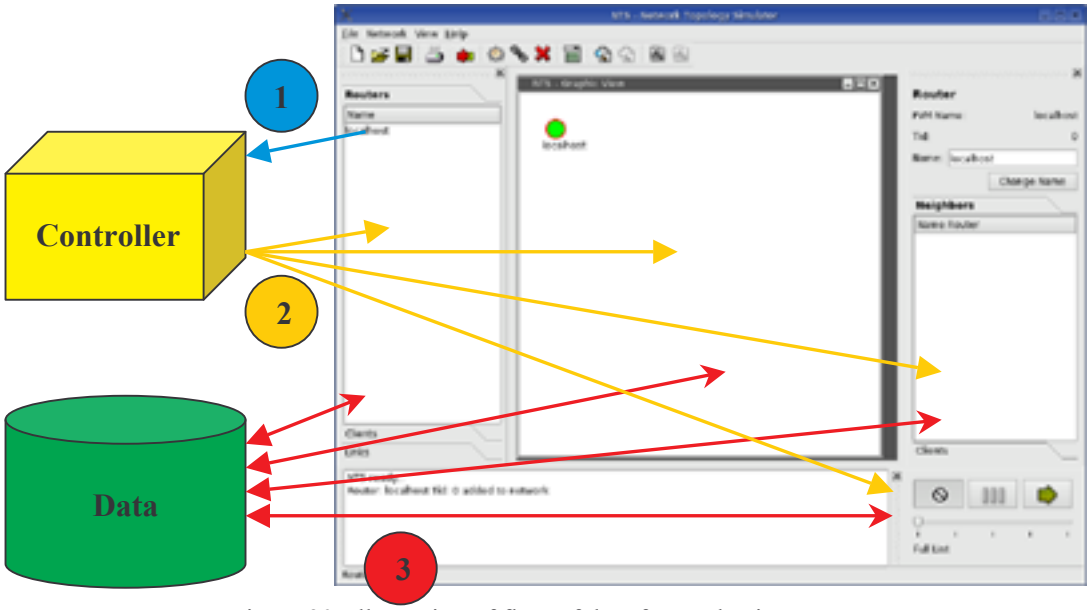

Figure 30: Illustration of flow of data for a selection

5.3.3 Graphics

The *ntsGraphics* is the most complex subsystem of the ntsMaster program. It is the only subsystem that is not an integrated VCD method like most of the other views. The Package has its own data structure, façade to control it, and view as separate classes (Figure 31).

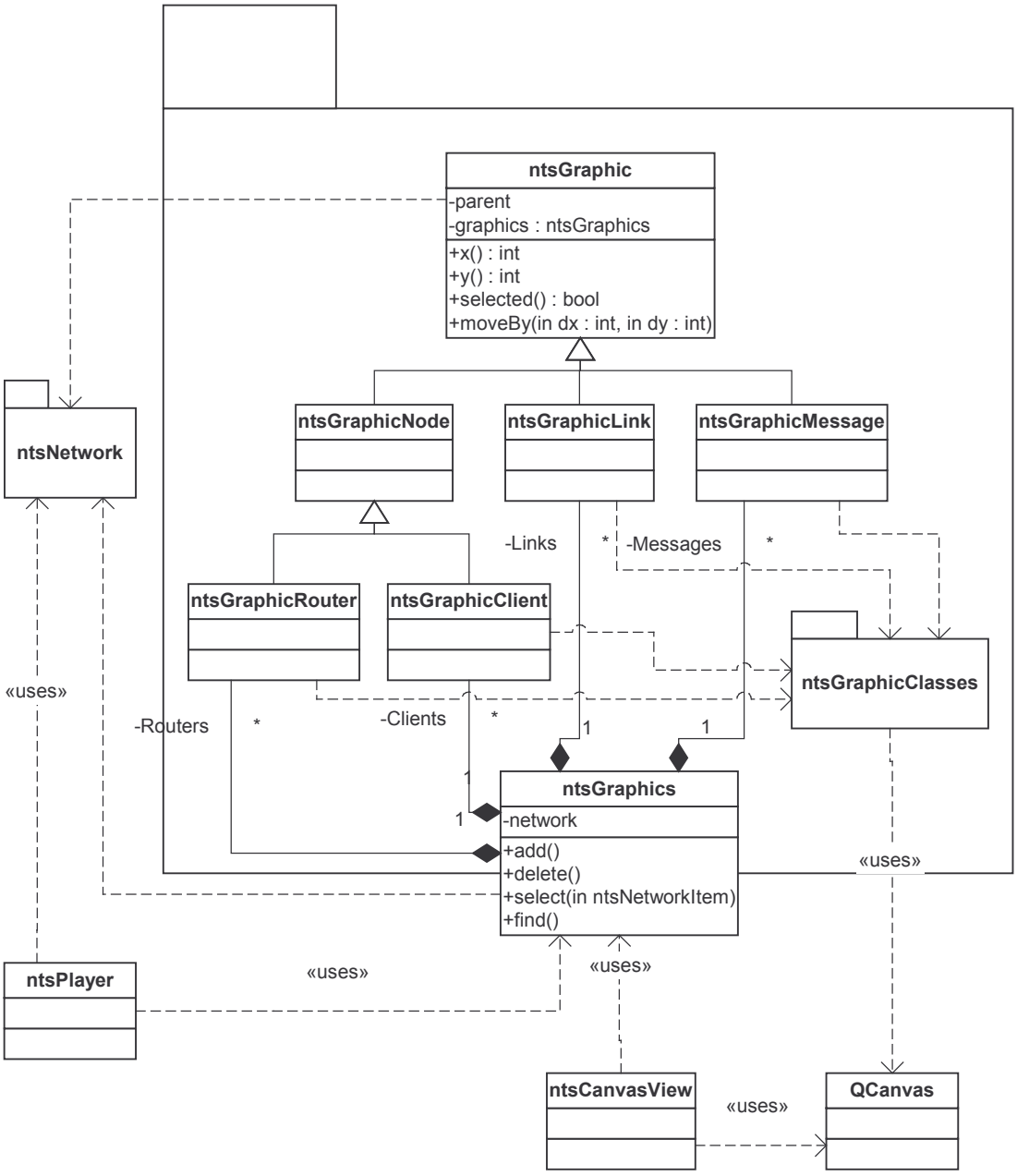

Figure 31: ntsMaster Graphical Package

QCanvas / QCanvasView:

The graphics of NTS use the *QCanvas* package to simply make the actual graphics. A *QCanvasView* class can display the contents of a *QCanvas* class in a graphical widget. By sub-classing the *QCanvasView*, one can manipulate the contents of the *QCanvas* directly, such as dragging elements around. *ntsCanvasView* is the subclass used for this purpose. It is connected to the graphical representations of the network

components via the sub-classed graphical components used as building blocks by the graphical system.

The *ntsCanvasView* is considered as a normal view of the Data by the Controller, and is connected to the Selection mechanism, allowing the user to click elements in the Graphical Display to select them. The *ntsCanvasView* is also connected to the Link addition system of the Controller so that a line can be dragged from Router to Router to connect them.

ntsGraphics:

ntsGraphics is the façade of the graphics subsystem. It is the object that is connected to the *ntsNetwork* object, and instantiates graphical representations any newly created network element. The *ntsGraphics* is also connected to the Controller selection system to display the current selection graphically. The selection is forwarded to the appropriate subclass.

Data Structure:

The graphical data structure mimics the ntsNetwork data structure, as it is to represent this one graphically. One can see the same basis as the ntsNetwork based on the graph structure. The difference here is that the connections serve for the graphics to move around freely around the canvas. There is no need for any cascading delete, as each representation will be automatically removed on the deletion of the network element by the link between the ntsNetwork and the ntsGraphics.

Graphic Representations:

The Graphic representations of each element have got references not only to the graphic system they belong to but also to the network element they represent. This is used to get the information needed to faithfully represent the element.

The Graphic representations are composed of various graphical elements put together. These elements must move as a group when dragged along the canvas. These graphical elements were therefore sub-classed to have a reference to the graphical representation they belong to. The graphical representation *moveBy*() function could then be called by the *ntsCanvasView* to move the full illustration rather then just a component of it.

5.3.4 Views

Views such as the Lists View, Properties View, Messages View, Paths View, and Log Views all plug into the Controller. Each is connected to the ntsNetwork Package (except for the Log Views which don't need to be), and the selection method of the Controller if needed. As explained in the Controller section it is relatively easy to add a view. Each view is independent of the others, and only minimal changes to the main window are required to display them properly.

5.3.5 Player

The player is not really a package or a view. Its graphical display is embedded in the main window, while its control is a set of separate classes (main class and a separate thread). It is coupled with the Data, requesting the Paths to play, but connects directly to the *ntsGraphics* class from the graphics package to tell it when to create a Message representation and where, as well as when to delete one. In terms of coupling, the Data and Graphics are not aware of the Player's presence.

5.3.6 Threads

Main separate threads are used within the ntsMaster program. The Qt thread implementation class was used, usually with two fail-safes: a running condition, and pausing condition. Threads are used in the ntsNetwork package (continuous listening to incoming messages while in running mode), the Player (playing the Paths out independently), and the Log Views (reading the open file continuously for any extra data added).

The Log View threads cause problem, as they have no pauses in them, making them very processor hungry. For some unknown reason, if a pause was added the reading of the open file would cease and no further displays would happen. No reasonable solution was found for the time being.

5.3.7 File Format

The ntsMaster permits the user to save to file network topologies he is testing for reuse later. The file format use is a simple text file with a ".net" extension. The file format is shown in (Figure 32).

NTSnetwork:
nodes:
"PVM name" "X location on canvas" "Y location on canvas" "given name"
links:
"PVM name" "PVM name"

Figure 32: save file format

The PVM name of each node is saved along with its coordinates on the graphical display and the user given name if any. For the links, the PVM names of each side of the link recorded. If a link is bi-Directional, then a second line for the link will be included with the names inversed. When adding an inverse link on top of an existing link, turns the existing link into a bi-directional one (see section 4.6.2 Network Building).