MONITORING AND CONTROLLING VRVS REFLECTORS

Catalin CIRSTOIU, catac@cs.pub.ro

Computer Science Department, "POLITEHNICA" University of Bucharest
313 Splaiul Independentei St., Sector 6, Bucharest, Romania

- June 23, 2003 -

Working in a collaborative environment, where people are dispersed across several countries and continents requires means of communication that are low-cost, bandwidth-efficient, extensible and robust. VRVS is a videoconferencing system based on a set of servers called reflectors that route the audio/video streams to the participating clients. This project proposes and explains a method to monitor and control the VRVS reflectors in order to enhance the quality of the service.

1. Design

For each VRVS reflector, a MonALISA service is running using an embedded Database, for storing the results locally, and runs in a mode that aims to minimize the reflector resources it uses (typically less than 16MB of memory and practically without affecting the system load). Dedicated modules to interact with the VRVS reflectors are developed: to collect information about the topology of the system; to monitor and track the traffic among the reflectors and report communication errors with the peers; and to track the number of clients and active virtual rooms. In addition, overall system information is monitored and reported in real time for each reflector: such as the load, CPU usage, and total traffic in and out.

1.1. Monitoring Modules

The main component that gathers data, injecting it into the system is a monitoring module. A monitoring module is a Java class that can be dynamically loaded from any location specified by a URL. At the same time with importing, data is also translated (usually by parsing) to a format understood by the MonALISA. With numerical data received from the monitored device, information about monitored node, such as name, cluster and farm, is also added.

Usually, these modules are invoked at fixed time intervals, using a priority queue. They can extract SNMP data, run rsh and ssh scripts where this is possible, connect through a TCP socket and query a device etc. In order to maintain up-to-date large distributed systems, these modules are built to be dynamically instantiated from certain, possible fixed, URLs.

All modules implement a MonitoringModule interface that allows different implementation for each module.

When it is invoked, the module returns a vector of Results that are passed further to the MonaLISA core.

1.1.1. Number of clients and virtual rooms – SyncVrvsClientsT

A reflector always keeps track of the current connected users. This information can be gathered by making a TCP connection to the reflector on a certain port and issuing a command. The reflector will return on the same connection the information requested and then it will close the link. The received text is parsed and we get:

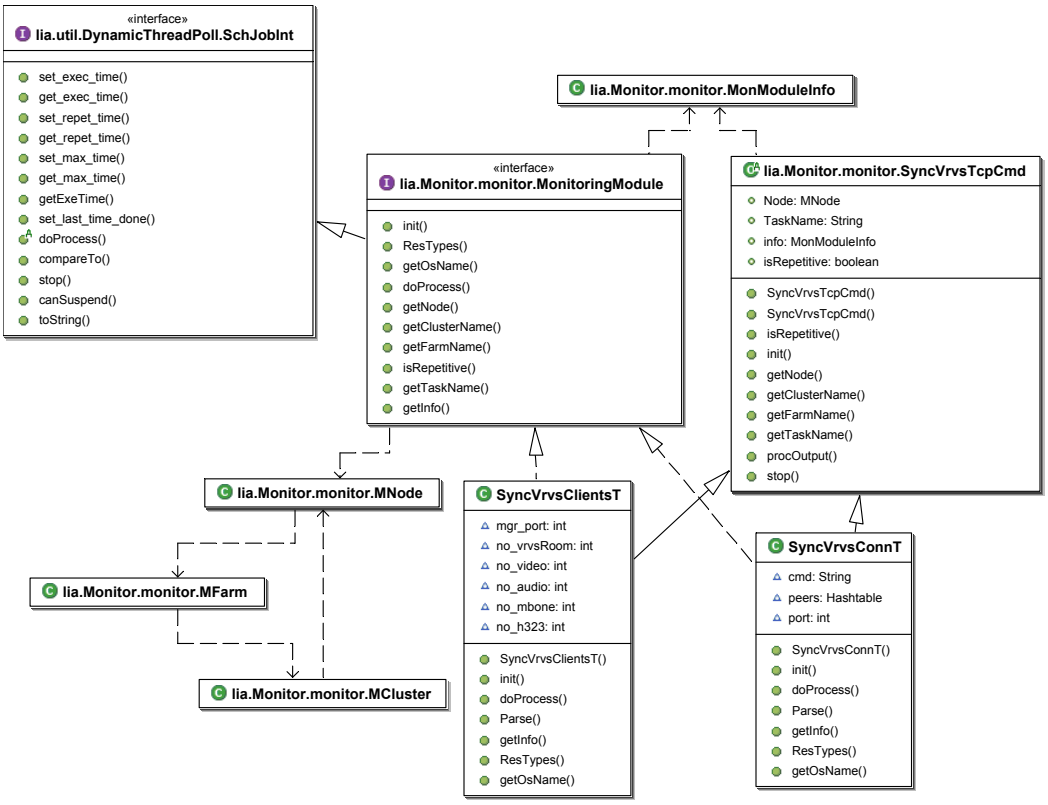
- number of audio clients;
- number of video clients;
- number of virtual rooms.

1.1.2. Peer links for a Reflector – SyncVrvsConnT

Peer links represent the current active tunnels for a reflector. The reflector forwards packets to the other reflectors based on users and virtual rooms information (a packet from a user in a certain virtual room must reach all users in the same virtual room wherever they would be in the whole system). For each tunnel, we can query the reflector for the following data, which is forwarded to the listeners:

- quality of the link, as computed by the reflector software;
- lost packages

1.1.3. Reflector querying modules diagram



1.1.4. Getting internet links' qualities between reflectors – ABPing

In order to be able to supply a routing alternative for the current selected tunnels, we have to monitor links' qualities between reflectors. For each reflector a set of possible peers is chosen and the module monitors the quality of each link. The module sends UDP packets to the other reflectors. The other reflectors respond sending back the received packet. This way we can determine simple, but important factors that influence the quality of each link. The quality is computed with the following formula:

$$RTimeQuality = OVERALL\_COEF + RTT\_COEF * rtt + PKT\_LOSS\_COEF * loss\% + JITTER\_COEF * jitter$$

This formula is flexible enough to permit calculating any kind of quality, based on RTT, Packet Loss and Jitter. The values obtained by pinging peers are:

- rtt – the round trip time for packets to travel to the peer and back;
- loss – percent, ranging from 0 to 1 of lost packets sent to the peer;
- jitter – sum of the variations of rtt for a set of samples, divided by the average rtt and number of samples.

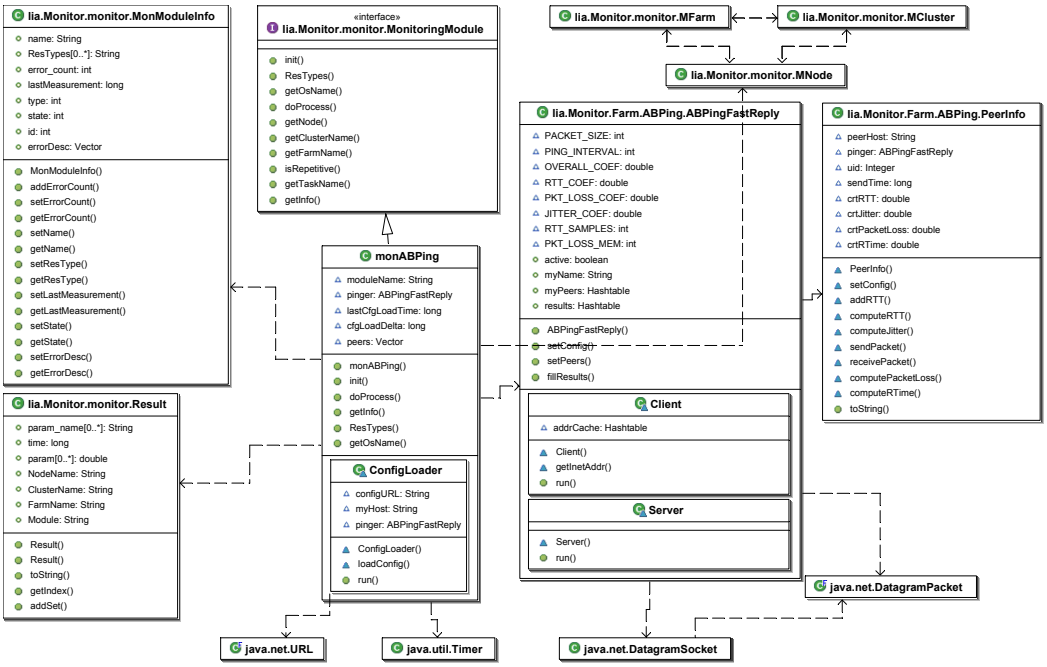
The list of available peers for each reflector and the *_COEF coefficients should be highly configurable to allow easy reconfiguration. To reach this goal, the configuration file is the same for all reflectors, each one knowing to extract only the information that is needed. The coefficients must be the same for all reflectors in order to obtain comparable RTime qualities.

The configuration file is loaded at start, and then it is periodically checked, from a URL configured when starting MonALISA service on the reflector. If there is a new peer for a reflector, it is added to the list of peers in the monABPing module. Similarly, if a known peer isn't found anymore in the configuration file, it is deleted from the peer list. If at least one of the coefficients modify, all measurements are reset and the new values are computed using the previous formula.

Here is a sample configuration file:

```
vrvs.co.pub.ro vrvs-eu.cern.ch vrvs-us.cern.ch
vrvs-eu.cern.ch vrvs-us.cern.ch vrvs.co.pub.ro cinxia.tv.funet.fi
vrvs-us.cern.ch vrvs-eu.cern.ch vrvs.co.pub.ro vrvs-starlight.cern.ch
…
OVERALL_COEF 0
RTT_COEF 0.5
PKT_LOSS_COEF 100
JITTER_COEF 20

# We keep last RTT_SAMPLES rtts (integer value)
RTT_SAMPLES 10

# The history of Lost Packages is PKT_LOSS_MEM long (integer value)
PKT_LOSS_MEM 20

# The size of the packet sent over the net (must be > 3 bytes)
PACKET_SIZE 450

# Time between pings (ms). Should be big enough to allow reasonable
# time for packets to return to sender and not consider them lost
PING_INTERVAL 4000
```

1.1.5. ABPing module diagram



1.2. Filters

A filter is, generally speaking, both a client and a data producer for other clients of the MonALISA service. As a client, a filter receives all data from the monitor modules. It analyzes this data and performs an action according to the received data. As a data producer, it generates Results for other clients.

For monitoring Reflectors, the MonALISA service must be started using two filters that we are going to present in the next section. Starting these filters can be done activating some properties in the MonALISA service configuration files, as we will see in the User Guide.

1.2.1. Filtering data – TriggerAgent

This filter has two functions. First, it computes time mediated values for reflector peer links quality. Three types of results are produced:

- Qual2h – mediated peer links quality over last 2 hours;
- Qual12h – the same, for last 12 hours;
- Qual24h – as above, for last 24 hours.

The second function of this filter is that it listens for all results coming from the reflector, i.e. results from vrvsClientsT and vrvsConnT. In this case, the results received are of no concern. It is important whether these results come or not. If there is no data for more than 2 minutes, a special type of result, “alarm”, is sent to the listening clients. These clients can interpret and announce the user in different ways that a certain reflector encounters problems.

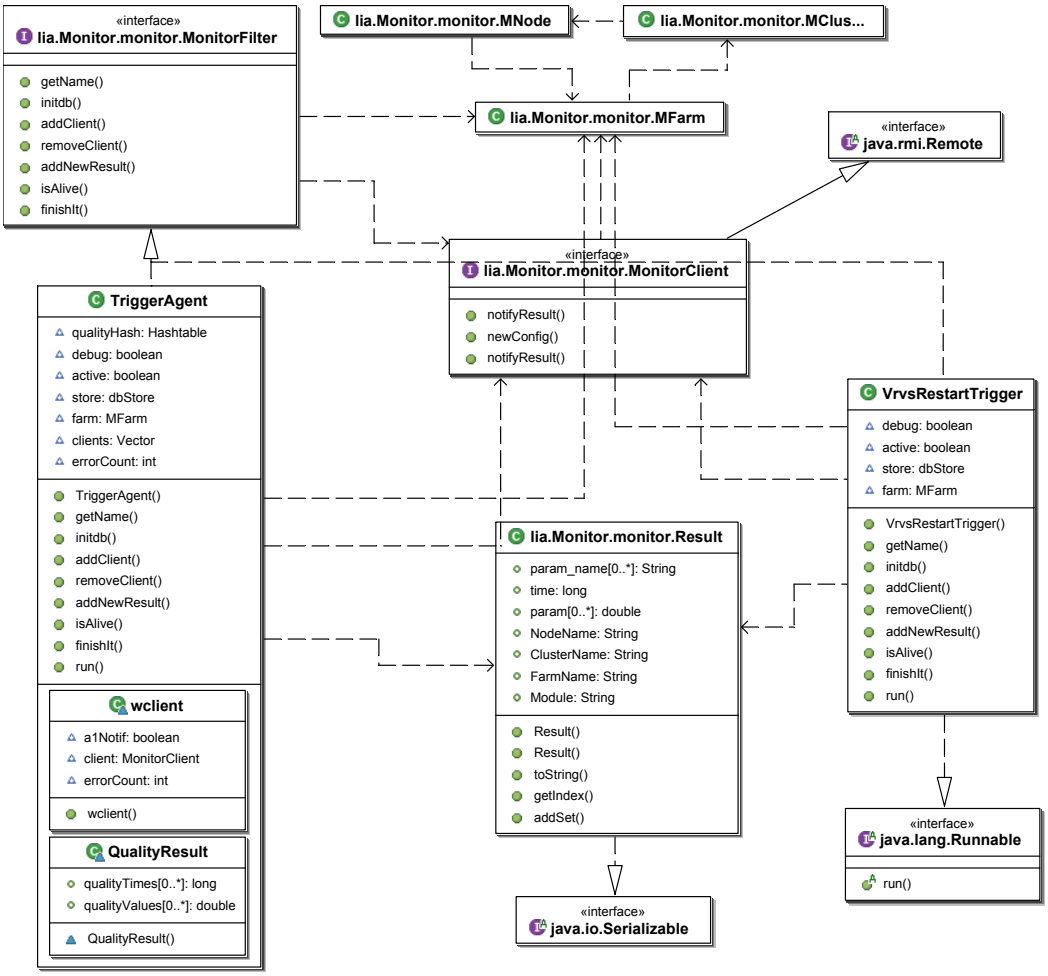
1.2.2. Triggering Actions – vrvsRestartTrigger

As the first filter, this one also listens for data coming from the reflector (the two monitoring modules). The difference is that this time, no other results are produced. Instead, two actions are taken.

First, if there is no data for one minute, it concludes that the reflector has a problem and the easiest way to solve it is by restarting the reflector. The reflector software is started/restarted/stopped with a shell script located in a certain directory in the current user home path. To restart it, a new process is created, launching the script.

Second, if there is still no response for another minute, then it can be concluded that the reflector cannot be started. One last attempt is made, invoking for a second time the script. To help debugging, the output and/or exceptions generated by this action are captured by the filter and are packed into a mail that will be sent to the administrator. The e-mail address of the administrator can be configured in the ML service parameters. E-mail sending is just a form of administrator notification. Exactly the same principle would be applied to send SMSs instead of e-mails.

1.2.3. VRVS Filters Diagram



1.3. MonALISA Clients for VRVS

To achieve the goal of this project, two MonALISA clients were developed. There is a monitoring GUI client suitable for VRVS users that want to observe in real time the reflectors activity. The other client runs on an internet site as background process, monitors the reflectors statuses and if needed it issues commands. We present a short description of both, in the following sections.

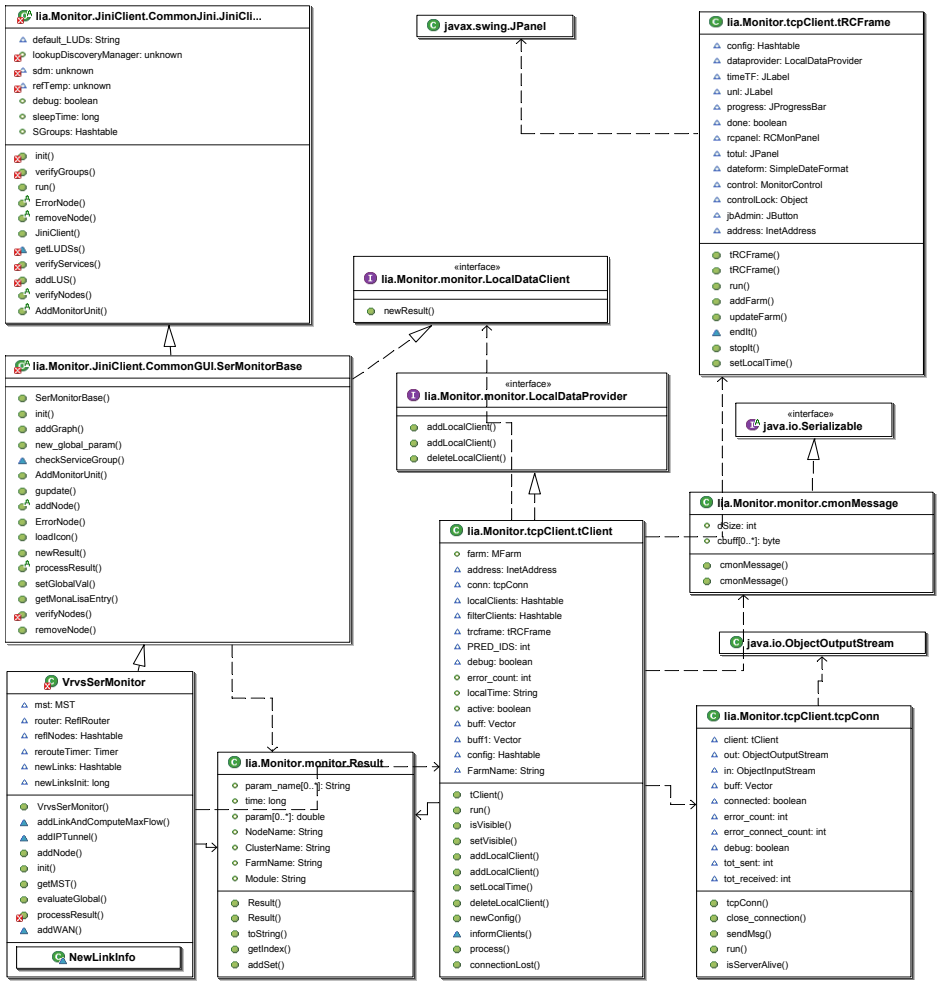
1.3.1. MonALISA GUI Client

A dedicated GUI for the VRVS version of MonALISA client was developed as a java web-start client. This GUI provides real time information dynamically for all the reflectors which are monitored. If a new reflector is started it will automatically appear in the GUI and its connections to its peers will be shown. Filter agents to compute an exponentially mediated quality factor of each connection are dynamically deployed to every MonALISA service, and they report this information to all active clients who are subscribed to receive this information.

It provides real-time information about the way the VRVS system is used (number of conferences or clients) the topological connectivity of the reflectors and the quality of it and system related information (IO traffic CPU load). Clients can also get historical data for any of these parameters.

The subscription mechanism allows one to monitor in real time any measured parameter in the system as all the updates are dynamically displayed on the open windows. Examples of some of the services and information available, visualizing the number of clients and the active virtual rooms, the traffic in and out of all the reflectors, as well as problems such as lost packets between reflectors are presented in the User Guide chapter.

The simplified diagram of the GUI client follows:



This client is also able to compute a minimum spanning tree, using the same code that ReflRouter client uses. The MST is displayed on the GUI to allow observing at any moment if the current selected tunnels provide the best performance.

1.3.2. MonALISA ReflRouter Client

We developed the ReflRouter client that is able to provide an optimized dynamic routing of the videoconferencing data streams. This client requires information about the quality of the alternative connections in the system and it solves, in real-time, a minimum spanning tree problem to optimize the data flow at the global level.

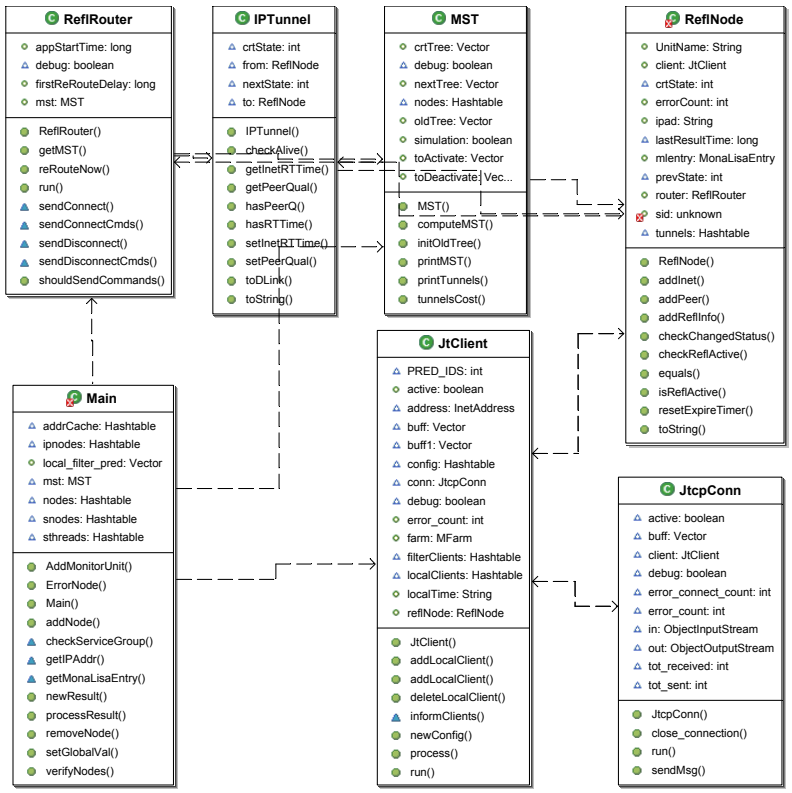
To evaluate the connection quality with possible peer reflectors we developed monitoring agents performing ping like measurements using UDP packages, which are deployed on all the MonALISA services. These agents perform continuously (every 4s) such measurements and with a selected set of possible peers, which can be dynamically reconfigured, for each reflector. We are using small UDP packages to evaluate the Round Trip Time (RTT), its jitter and the percentage of lost packages.

The best routing path for reapplication of the multimedia streams is defined as a Minimum Spanning Tree (MST) [19]. This means that we need to find the tree that contains all the reflectors (vertices in the graph G) for which the total connection "cost" is minimized:

$$MST = \min(\sum_{(v,u)\in G} w((v,u)))$$

The "cost" of the connection between two reflectors (w) is evaluated using the UDP measurements from both sides. This cost function is build with an exponentially mediated RTT and if lost packages are detected or the jitter of the RTT is high the cost function will increase rapidly. Based on these values provided by the deployed agents, the MST is calculated nearly in real - time. We implemented the Barůvka's Algorithm, as it is well suited for a parallel/distributed implementation. Once a link is part of the MST a momentum factor is attached to that link. This is to avoid triggering reconnections for small fluctuations in the system. Such cases may occur when two possible peers have very similar parameters (or they may be at the same location).

The UML diagram of this client:



2. Implementation

We will present the most interesting parts of the implementation of various components of the whole project.

2.1. Monitoring Modules

A monitoring module is a Java class that must implement the following interface:

```
public interface MonitoringModule extends
                          lia.util.DynamicThreadPoll.SchJobInt {
      public MonModuleInfo init(MNode node, String args);
      public String[] ResTypes();
      public String getOsName();
      public Object doProcess() throws Exception;
      public MNode getNode();
      public String getClusterName();
      public String getFarmName();
      public boolean isRepetitive();
      public String getTaskName();
      public MonModuleInfo getInfo();
}
```

The SchJobInt is an interface that represents a job that can be scheduled for execution. A monitoring module is such a job that monitors the activity on a certain MNode (monitored node that is part of a Cluster, on a Farm). It is invoked at configurable time intervals – the doProcess() method. If the job fails, it throws an exception. If it succeeds, it returns an object: a Result, or a Vector of Results. These results are serialized and passed to the listening clients.

2.1.1. Number of clients and virtual rooms – SyncVrvsClientsT

The SyncVrvsClientsT is a third generation class that gathers reflector specific information. The first version of this module launched command line program, designed by the VRVS team that retrieved the information from the reflectors. This was generating a big overhead – it meant creating a new process each 20 seconds. The next version connected to the reflector on a certain port and issued the following command: "`passwd/check_host/params`". This would return a text containing the number of virtual rooms, audio and video clients. This approach was far better, but we discovered that the reflector was implemented as a single threaded server and there were problems if both modules tried to connect and query it at the same time. Third version corrected this problem by using the "synchronized" attribute for methods that could interfere.

2.1.2. Peer links for a Reflector – SyncVrvsConnT

Both VRVS modules are implemented using a single class SyncVrvsTcpCmd that simply connects to the reflector, issues the given command and returns the output. This module checks reflector peers' status for the last 30 seconds, using the following command: "`passwd/check_peer_status/params`". The result contains a list of peers with the corresponding Quality and Lost Packages for each of them. This information is parsed in the `doProcess()` method and for each peer a result is created. It returns a vector with all results.

2.1.3. Getting links' qualities between reflectors – monABPing

When this module is initialized, the configuration must be read from a certain URL, passed as a parameter in the *ml.properties* configuration file. Then, at fixed intervals, the configuration is reread from the same URL. This can be easily achieved by defining an inner class to handle this problem:

```
class ConfigLoader extends TimerTask {
      ConfigLoader(String myHost, ABPingFastReply pinger){ … }
      void loadConfig(String host, String url){ … }
}
 reloadCfgTimer.schedule(new ConfigLoader(Node.getName(), pinger),
```

```
0, cfgLoadDelta );
```

In the doProcess() method we just call for each peer, the FillResults method of ABPingFastReply. This is the "worker" class for this module. It has two inner classes: Client and Server that implement the Runnable interface, working in separate threads. ABPingFastReply has a Hashtable with all its peers. A peer is represented by the PeerInfo class, holding all concerning data about it. When a peer is added, it receives a unique id (short integer value) that is used to identify very fast the peer when receiving a packet.

The Client thread sends a packet to each peer. The first byte contains the type of packet (ECHO or REPLY). The next two bytes hold the UID of the peer. The rest of the bytes in the packet are filled with random data to avoid compression of different network protocols. After sending an ECHO packet to all peers, the client sleeps for a period considered long enough to allow packets to return from peers. Then, the list of results (filled by the Server thread) is analyzed and PeerInfo classes information is updated. If a Peer doesn't receive a result, its packet loss counter is increased. Either way, rtt, jitter, RTime information is recalculated.

The Server listens on a port for UDP packets. When it receives a DatagramPacket, it checks the first byte. If it is ECHO, then it is changed to REPLY, the destination address is set to be the source address and the packet is sent. This operation is happening as fast as possible – this is why the module is called FastReply. If the first byte is REPLY, then this means that the packet is returning from a peer. The next 2 bytes are converted to an integer value using the fast bit shifting operators – the id of the peer. This id is the key used to add into the results hashtable the time when the packet was received.

Immediately before sending the packet, the time is recorded in the corresponding PeerInfo class. Having these two times, both taken on the same machine we can easily compute the RTT.

One can argue that computing this way the RTT we are prone to big errors, because this is done in application space, and not in kernel space, as it would have been if using a real ping utility. To avoid this kind of errors, we compute the mediated value for all variables for a period of time:

```
for(int i = 0; i < rttSamples.length; i++){
      sum += rttSamples[i];
      min = (min < rttSamples[i] ? min : rttSamples[i]);
      max = (max > rttSamples[i] ? max : rttSamples[i]);
}
crtRTT = (sum - min - max)/(rttSamples.length - 2);
medRTTSamples[crt] = crtRTT;
```

From the set of results, we eliminate first the biggest and the smallest time and only after that we compute the RTT for that period. Jitter is computed as the sum of RTT variations divided by the number of samples and mean value:

```
for(int i = 0; i < rttSamples.length; i++)
      delta += Math.abs(medRTTSamples[i] - avg);
crtJitter = delta/((medRTTSamples.length - 1) * avg);
```

This formula is good for a bigger RTT, but for small values it may be unstable, the *avg* being too small. This could happen especially in LANs. But in this kind of networks, the jitter cannot influence the quality of the videoconference, and therefore it can be considered equal to 0, if RTT is below a reference value.

The quality of a link is finally computed as follows:

```
crtRTime = pinger.OVERALL_COEF +
             pinger.RTT_COEF * crtRTT +
             pinger.JITTER_COEF * crtJitter +
             pinger.PKT_LOSS_COEF * crtPacketLoss;
```

The coefficients are those read from the configuration file.

2.1.4. Monitoring Modules configuration file

All these modules reside in a jar file in the MonALISA service distribution. They are dynamically loaded only if their name is found in the configuration file. For VRVS reflectors, the configuration file is usually like this:

```
*Reflector
```

```
>vrvs.co.pub.ro
SyncVrvsClientsT%30
monProcStat%30
monProcIO%30
monProcLoad%30
*Peers{SyncVrvsConnT, vrvs.co.pub.ro, " "}
*Internet{monABPing, vrvs.co.pub.ro, " "}
```

Each MonALISA service monitors a 'Farm' that is uniquely distinguished by a FarmName. This name is set in the configuration file of the ML service. A farm monitors multiple Clusters. In case of the VRVS, these clusters can be considered as fields of interest. In each cluster there are several Nodes – the final elements from where the data is coming.

Words after a '*' represent the cluster name. We have three clusters in the above configuration file: Reflector, Peers and Internet.

Reflector cluster contains data about the machine and the reflector. Its name is given with a '>' in front of it. For this cluster, several modules are started: SyncVrvsClientsT, monProcStat, monProcIO and monProcLoad. After the module's name, with '%30' it is specified that the respective module should be invoked each 30 seconds. This value can be different for each module. monProcStat, monProcIO and monProcLoad modules retrieve information read from the */proc/* file system (network traffic, current load etc.). The information returned by these modules (i.e. the types of results) is fixed.

The Peers and Internet clusters return information about the reflector peer links and the quality of network links to the other "neighboring" reflectors. Each peer is considered a Node in ML terminology. These can change while ML is running and therefore, these modules return variable types of results. These modules are configured with the parameters put between '{' and '}'. For example, the monABPing module reads its configuration from the specified URL (in the ml.properties file), looking for the hostname specified as parameter (here it is "`vrvs.co.pub.ro`").

2.2. Filters

Filters are Java classes that implement the MonitorFilter interface:

```
public interface MonitorFilter extends java.io.Serializable {
        public String getName();
        public void initdb(dbStore datastore, MFarm farm);
        public void addClient(MonitorClient client);
        public void removeClient(MonitorClient client);
        public void addNewResult(Result r);
        public boolean isAlive();
        public void finishIt();
}
```

When a client wants to receive data from this filter, it must implement the `MonitorClient` interface and must call the `addClient()` method. This way more different type of clients can register to receive information from this filter. The filter, also is a `Runnable` object, which means that it runs in a separate thread. Every 10 seconds the thread informs all registered clients, sending results (Result or Vector of Results objects).

It is important to understand that all results coming from all modules reach all filters. A result is passed to the filter through the `addNewResult()` method.

2.2.1. Filtering data – TriggerAgent

This filter has two functions. First, it receives Quality results from the SyncVrvsConnT modules. Then, it computes an exponentially mediated value for the last 2, 12 and 24 hours:

```
for (i = 0; i < QUALITY_NUMBER; i++) {
        double oldQ = qr.qualityValues[i];
        double newQ = r.param[j];
        long t = qr.qualityTimes[i];
        long dt = now - t;
        long diffdt = QUALITY_TIME[i] - dt;
        qr.qualityValues[i] =
                oldQ * ((double) diffdt / (double) QUALITY_TIME[i])
```

```
                + newQ * ((double) dt / (double) QUALITY_TIME[i]);
}
```

The corresponding values for the 2, 12 and 24 hours are in milliseconds:

```
public static long[] QUALITY_TIME =
{ 2 * 60 * 60 * 1000, 12 * 60 * 60 * 1000, 24 * 60 * 60 * 1000 };
```

The second function for this filter is that after 2 minutes of silence from the reflector, an alarm is triggered. This is accomplished easily, given the flexible structure of the filter. Each time a result from the reflector (not from the monProc* modules) is received, the time is recorded in a variable. Each 10, when the filter is about to inform its clients, the time of last pertinent result is checked. If it is earlier than 2 minutes, the alarm is triggered, i.e. it is created a Result with a parameter called "alarm" and the value true.

2.2.2. Triggering Actions – vrvsRestartTrigger

This is a different kind of filter. It just receives information from the modules but doesn't send any results to the clients. It can be considered an agent that can take some simple decisions based on the monitored data. The decisions are much simpler than those that can be taken into a client (see the ReflRouter client) because here, only the information from a single reflector is available. Therefore it hasn't any knowledge about the peers of the reflector, although this could be easily implemented by building a module that receives, and sends to peers information about its reflector. This information could be packed into results that could reach more intelligent filters (see future work section of this project).

Coming back to vrvsRestartTrigger, it has two functions, based on two alarm triggers. First alarm is triggered after one minute of silence from the reflector. The mechanism is the same as in the previous filter, but the action is different. Instead of informing clients, this alarm creates a process that launches a shell script to restart the reflector software. If after this operation the filter receives data, then it means that the restart was successful, but if after another minute there is no data, it means that there is a serious problem and an administrator must be notified.

The list of administrators is given as a property in the *ml.properties* configuration file: "`lia.Monitor.vrvs.MAIL=user1@host1,user2@host2`". One last attempt to restart the reflector is made, but this time the output of the script trying to restart the reflector (or any exception that may occur) is copied into an email that is sent to the list of administrators.

The mail is sent with the "mail –s subject address" command. The output of the script that tries to restart the reflector is copied to the mailing process. The mail utility sends mail only when receives a single "." after a new line. This way it can be controlled if the mail should be send or there was an error this time also.

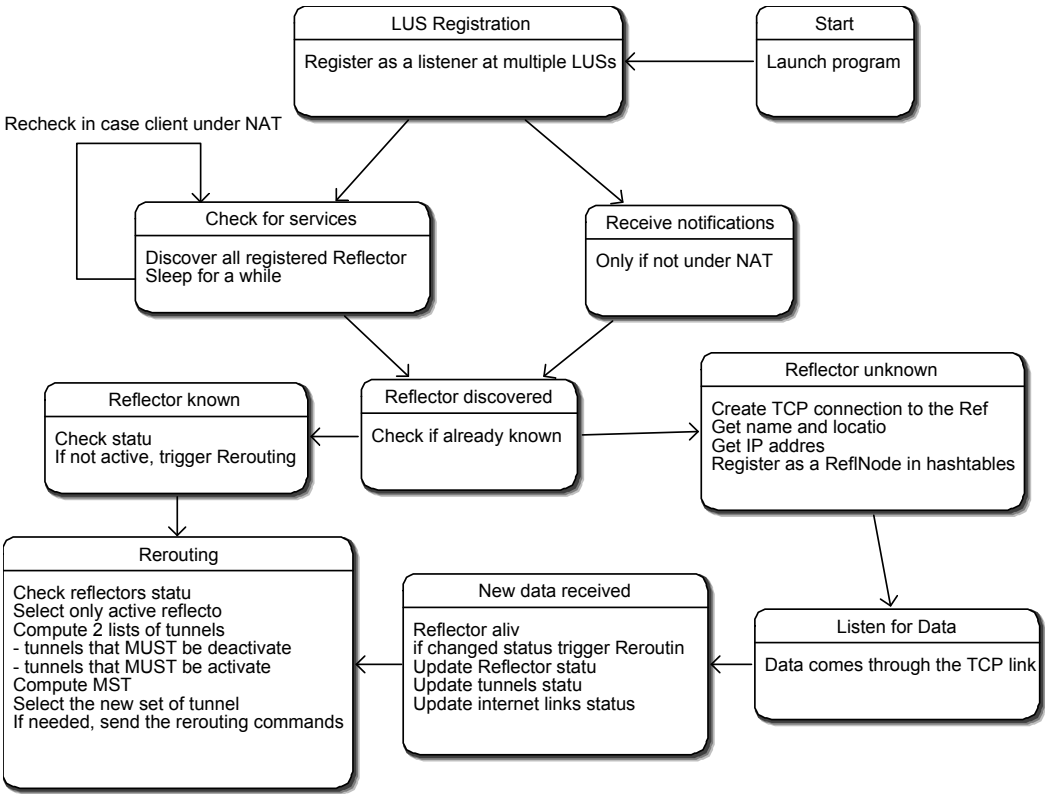
2.3. The ReflRouter client

Like other MonALISA clients, ReflRouter has a complex structure. We present a state diagram to ease the understanding of all processes and then, we will focus on the routing part.

In order to be able to perform the rerouting of the multimedia packets, we have to know anytime the status of the Reflectors, their peer links and the quality of their links with a set of "neighboring" reflectors. All this information is kept in a set of classes. The reflectors (ReflNodes) are kept in 2 hashtables, one having the key as reflector name and one the reflector's IP address. This is useful because the way data is received into the Results classes. A peer node (MNode) is distinguished by its full qualified domain name (such as "vrvs-eu.cern.ch") when the result is received, for example, from the vrvs-pub (FramName or UnitName and a full name as "vrvs.co.pub.ro"). From this full name, the IP address is found. Having the IP address of the peer we can find the peer reflector in the hashtable.

Currently, we do not monitor all Reflectors in the VRVS system, and therefore, Results containing peer links that are not found can be received. These links are ignored and no routing is performed with them. The condition to perform routing with just a part of the reflectors and not affect the functioning of all system (i.e. not make cycles) is that the monitored reflectors be the backbone of the VRVS. Cycles could be produced if, for example, we would monitor two reflectors that are leaves

in the VRVS tree (the system would issue commands to connect these two reflectors, being unaware that they are already connected through other reflectors).



The peer links are in fact tunnels over the Internet. They are kept in a hashtable "tunnels" in each ReflNode, the key being the name of the peer Reflector. The available links with the other reflectors are also considered tunnels and are kept in the same class (IPTunnel). IPTunnels have a set of attributes. They have `peerQual` (if it is an active tunnel) and `inetRTTime` – the result from ABPing (if the peer is in the list of the nodes for the current reflector). These values have an associated time – the time when last information about that peer was received. This way, we can get an expiring time for each type of link and issue critical rerouting commands.

Each tunnel has two attributes referring to its state: `crtState` and `nextState`. The current state of a tunnel can be either `ACTIVE` or `INACTIVE`, if it currently is in the tree of selected tunnels or not. This attribute depends on the age of the `peerQual` attribute, and is checked (and possibly modified) each time the status of this tunnel is requested by the MST algorithm. The `nextState` attribute is set by the MST algorithm and can have multiple values: `ACTIVE`, `INACTIVE`, `MUST_DEACTIVATE` or `MUST_ACTIVATE`. The MUST_* states can be set before running the MST algorithm to select the tunnels that either must or must not be active anymore, as we will se later. The first two states are set by the MST algorithm and they usually mean states that are recommended for optimizing the overall cost of the tree.

The ReflRouter is a class that extends `TimerTask` in order to be invoked from time to time to optimize the tree. But usually it is necessary to react to certain events, like a reflector become active or inactive. Each 20 seconds (by default, but this value can be changed editing the *ml.properties* file) all reflectors are checked. This includes checking all tunnels. If a reflector or tunnel isn't active anymore, a rerouting is triggered. This happens also when a reflector becomes active and it must be included into the MST.

The rerouting process is further analyzed in the next three sections.

2.3.1. Setting the Restrictions for the MST algorithm

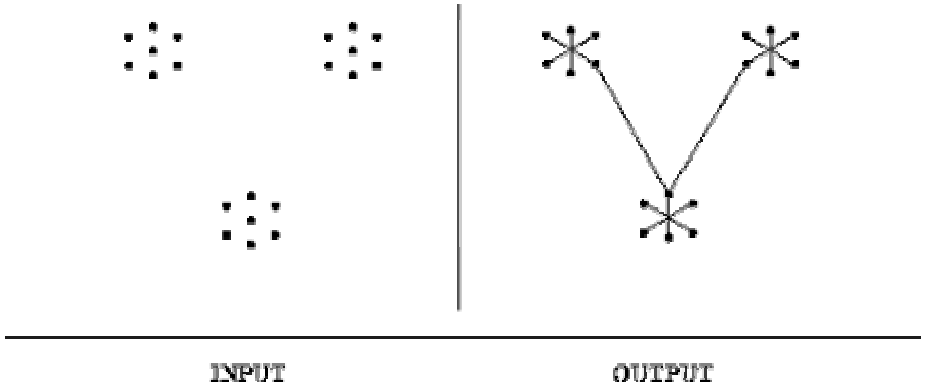
There are some critical cases that must be analyzed before running the MST algorithm. For this, each ReflNode is checked. If a node isn't active then it must not appear in the MST. Further, the tunnels that start from the inactive node must also not be present in the computed tree. Therefore, the next state will be set to `MUST_DEACTIVATE`. If the node is active, then each link to the other reflectors (either active peers or neighbor reflectors) is checked. If the peer reflector isn't active the respective tunnel must not be active.

Another problem arises when between two reflectors there is no ABPing information, or there is only one ABPing link. In this case, the state of the both peer links depends on the current status of the peer link. If there is at least one peer link, then both must be activated. If none is active, then no peer link must be active.

For the other cases the next state of a tunnel is initialized as `INACTIVE`, and the MST algorithm will set it as needed.

2.3.2. The MST algorithm

The minimum spanning tree (MST) of a graph defines the cheapest subset of edges that keeps the graph in one connected component. The input is a graph $G = (V,E)$ with weighted edges. The problem is to find the subset E' of E of minimum weight forming a tree on V.



For implementation, we used the Boruvka's algorithm, as it is also appropiate for a parallel implementation.

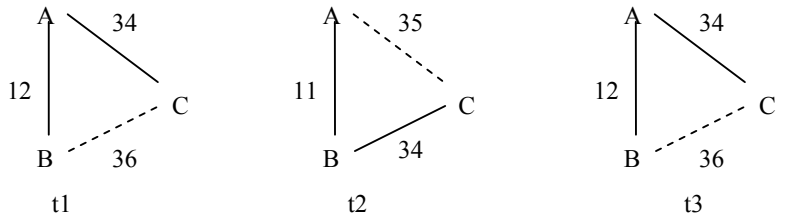
The original Borvuka algorithm is:

```
Given G = (V,E)
T = graph consisting of V with no edges
while T has < n-1 edges do
   for each connected component C of T do
      e = min cost edge (v,u) s.t. v in C and u not in C
      T := T union {e}
```

But there can be a problem if the graph isn't connex. In this case, there is no way to connect n-1 edges, so that condition is modified such that the while cycle repeats as long as there is at least one union made into the for cycle.

In our case, while joining subtrees, we also mark the next state of each tunnel that is used to perform the respective joint as ACTIVE.

Another modification that must be done to this algorithm is that the process is going to be running interative, i.e. we compute the MST, issue commands to change the tree, then we compute the MST and change the tree again and so on. A problem that could appear is that of active links oscillation.



For example, as in the above figures: at moment t1, the link between B and C is worse and therefore, is inactive; at the next moment, the link between A and C is worse and the algorithm would issue the commands to deactivate link A-C and activate instead the B-C link; but at the third moment, link between A and C is better once more than B-C, ant the algorithm would send new commands. This would be very bad for a system where there are live conferences ongoing. Therefore, we must take care and issue the commands for changing the route only when the new route is much better than the current route.

This problem can be solved by setting an inertial factor for the links belonging to the MST. Links that are currently in the MST have an artificial cost lowered by, for example 20%. It is important to give this value relative, not absolute as the cost of the links can vary very much – for example links between the reflectors in the same LAN have very low cost, compared to those separated by oceans. Using this inertial factor we are sure that the oscillations cannot happen very often, and that when a new link is chosen, it will bring an semnificative improvement in quality.

It's worth saying that this algorithm runs in O(m log n), where m is the number of edges and n the number of vertexes.

2.3.3. Generating commands sent to the Reflectors

Having the MST algorithm complete, we have to send the needed commands to the reflectors in order to change the current network topology to be as the one calculated. There are two types of commands: critical and optional commands. A command is considered critical when it must be sent to the reflectors; its next state is either `MUST_ACTIVATE` or `MUST_DEACTIVATE`. There is one more situation when the commands must be sent: the number of links to be deactivated differs from the number of links to be activated. In this case, it means that there are reflectors with only one peer link between them – the other must be activated, or the one that is active must be deactivated and one of the reflectors must be connected by some other path to the others. The optional type of commands refers to modifications that can be made in order to optimize the overall cost of the tree. It is only recommended to send these commands to the reflectors.

For each activation command, there is a tunnel that must be activated. Similarly, for each deactivation command, there is a tunnel to be deactivated. There is always a list of tunnels that are currently in the tree; the MST computes another list of tunnels to be in the next tree. Selecting the links that are in the current list of tunnels but not in the next, gives the list of tunnels to be deactivated; selecting tunnels that are in the next tree, but not in the current tree, gives the tunnels to be deactivated.

3. Tests and evaluation

The project integrates several modules. We will present a short description of how each of them was tested and the results obtained.

The vrvsRestartTrigger, responsible to the restart of the reflector in case of not responding was tested on a test reflector, "vrvs-test", located at Caltech. The ML service was started, having the filter activated. Then, the reflector software was stopped using the shell script. First, the TriggerAgent sent the "alarm" result to the GUI client, which displayed it in a different color; then, the vrvsRestartTrigger reacted by launching the vrvs shell script. The reflector started and responded to queries from the SyncVrvs*T modules. The results reached the GUI client which reverted the color of the reflector to normal state. Then, the reflector was stopped again and quickly after that, the script was renamed. After the first minute, the vrvsRestartTrigger filter tried to restart the reflector but failed; the color changed in the GUI client to indicate error state. Then, after one more minute, the filter tried to

restart once again the reflector; this time it has built an e-mail containing a message that it couldn't find the vrvs shell script. The mail was sent, reaching in less than 5 minutes from the failure of the reflector to the administrator.

The next module that was thoroughly tested is monABPing. In normal conditions, the values of the RTT returned were with 2 to 4 milliseconds bigger than those returned by the "ping" utility. These values were relatively constant in time, the mediation of the values succeeding to eliminate cases when the java garbage collector was run, or other threads were executing. The module was tested in different network environments, from UTP switched network, cable connection to the Internet and WAN links, performing as expected, showing a direct relation between the physical network quality and the value returned by the module. The only problem that could appear resides from the liberty given by allowing user to modify 8 parameters for this module. Therefore, in order to obtain pertinent results, a lot of tests must be performed.

The ReflRouter client was tested simulating that the commands were sent to the reflectors. For this, the previous tree was saved and when the algorithm was rerun, the current tree was initialized to that. The same algorithm (and classes) was used in the GUI client to perform the MST. Using the graphical interface and seeing the costs of the links the correctness of MST could be easily checked. More tests were performed running more instance of the ReflRouter on different machines, in different places. The algorithm issued the same commands no matter where it was run. We encountered problems with the links oscillations, but that were solved as explained in the implementation.

We discussed critical cases like reflector stopping to respond, one of two peer links disappearing, monABPing miss-configurations, but nothing about MonALISA service failure. This could happen for various motives, given it run in the Java Virtual Machine. This would be no problem if the monitored reflector were a leaf in the reflectors tree, but what if ML was running on a reflector belonging to the main trunk? It would mean that the ReflRouter would receive no more data from the respective reflector and it would consider it inactive. In that case the neighboring reflectors would receive commands to disconnect all links to this reflector and establish a new path. This could also be very bad for the system, so another solution is in development now: the status of the MonALISA service is permanently checked by a shell script ran from the crontab. When it doesn't respond, the ML service is restarted and all goes to normal. This ML check is performed at periods much smaller than link and reflector's expiring time, so this event would likely remain undetected at the ReflClient level.

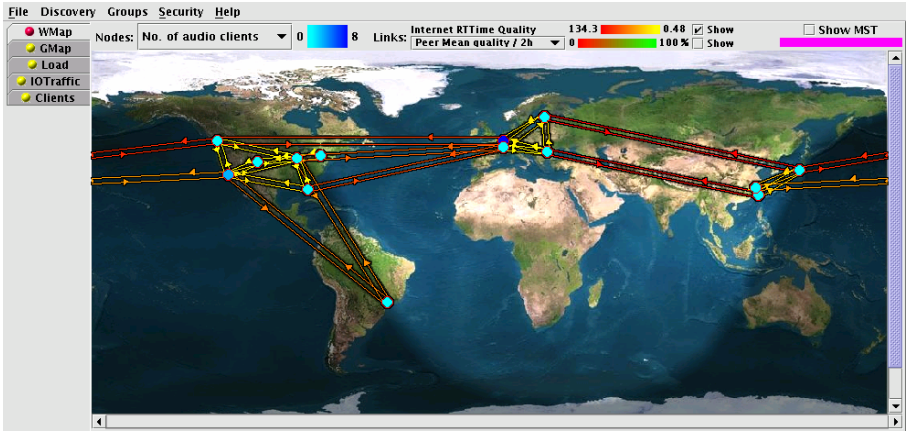
4. User Guide

The GUI client is started with the following command:

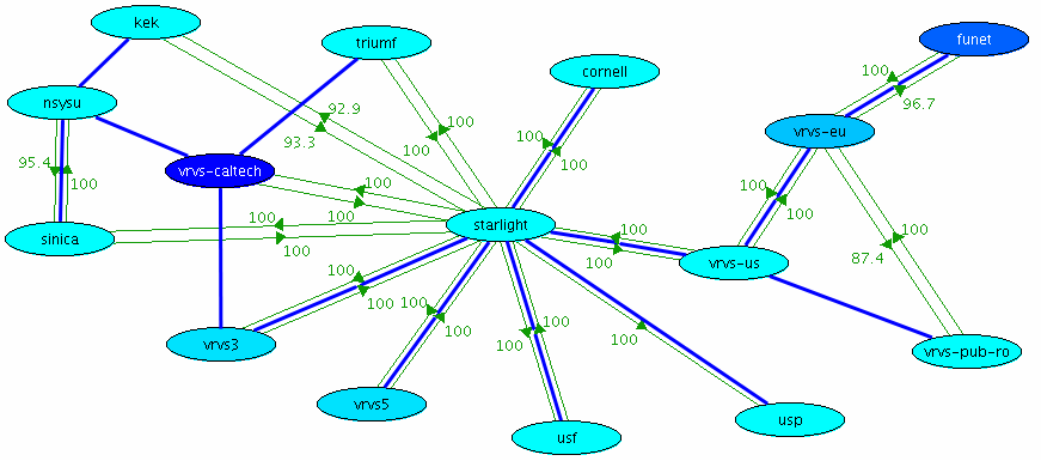
```
$ cd MSRC/MonaLisa/Clients/Gui
$ ./vrvsGlobal
```

This is a shell script with the following contents:

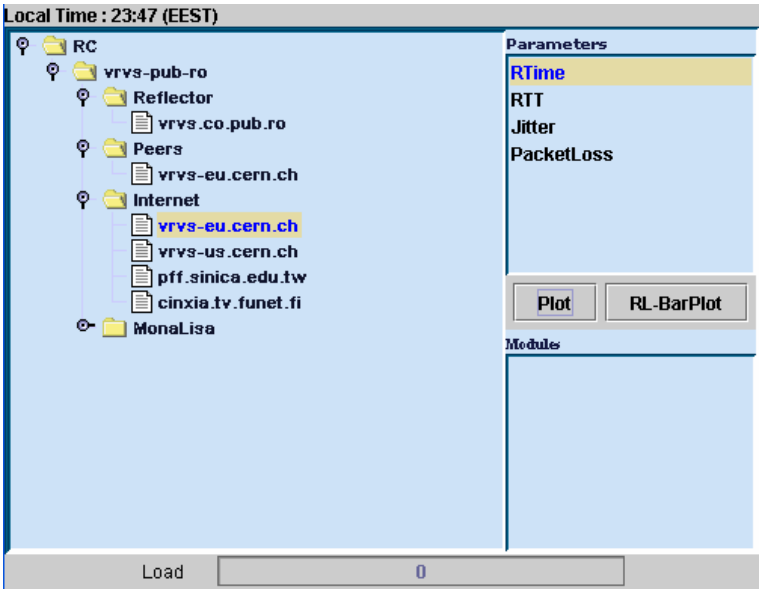
```
java  -jar -Djava.security.policy=policy.all \
      -Dlia.Monitor.debug=true \
      -Dlia.Monitor.keep_history=1000 \
      -Dlia.Monitor.LUSs=monalisa.cern.ch,pccit6.cern.ch \
      -Dlia.Monitor.group=vrvst1 \
      ../lib/vrvsJMonitorClient.jar
```



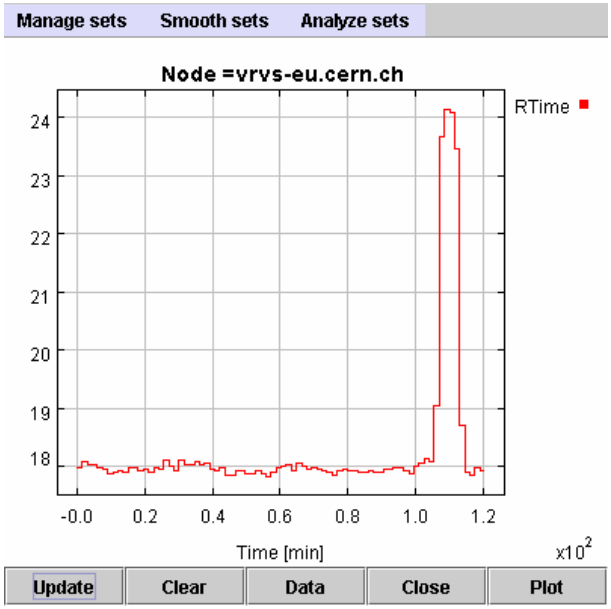
User can select what information to be displayed. For the reflectors one can se the number of audio and video clients, number of virtual rooms and total traffic. For the connections, peer links qualities are available (Qual2h… 24h) and inetABPing qualities, displayed in the previous picture.



The above is the view of the currently monitored reflectors, as shown in the GMap tab. The green links with numbers represent the quality returned by the reflectors. The blue lines represent the MST computed by the algorithm. The bubbles can be dragged with the middle mouse button and if clicked with the left button, detailed info is shown about respective reflector:



In the above window is shown the information configured in the configuration file for the reflector. Selecting the Internet cluster, one can see a detailed graph of the RTime quality for the respective link:



The ReflRouter client is started with the following command:

```
$ cd MSRC/MonaLisa/Clients/JReflRouter
$ ./jGlobal
```

The output corresponding to a rerouting event is something like:

```
[ Fri Jun 06 22:01:00 EEST 2003 ] --> ReRouting process STARTED ...
MST: Current tree's tunnels:
  MUST -> IPTun:starlight->sinica pQ=100.0 iRTT=1.0E50 crtState=ACTIVE nextState=MUST_ACTIVATE
  MUST -> IPTun:sinica->starlight pQ=100.0 iRTT=1.0E50 crtState=ACTIVE nextState=MUST_ACTIVATE
  OPTN -> IPTun:vrvs-eu->vrvs-us pQ=100.0 iRTT=0.769 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-us->vrvs-eu pQ=100.0 iRTT=0.5800000000000001 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-caltech->vrvs-test pQ=-1.0E50 iRTT=1.5 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-test->vrvs-caltech pQ=-1.0E50 iRTT=1.02 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:sinica->nsysu pQ=100.0 iRTT=3.5171428571428573 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:nsysu->sinica pQ=100.0 iRTT=4.01 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs5->vrvs3 pQ=-1.0E50 iRTT=15.128305785123967 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs3->vrvs5 pQ=-1.0E50 iRTT=15.129628099173553 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs3->vrvs-caltech pQ=-1.0E50 iRTT=18.0 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-caltech->vrvs3 pQ=-1.0E50 iRTT=18.501081081081082 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-eu->vrvs-pub-ro pQ=100.0 iRTT=19.127614379084967 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-pub-ro->vrvs-eu pQ=100.0 iRTT=18.5 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-eu->funet pQ=100.0 iRTT=29.252051282051283 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:funet->vrvs-eu pQ=100.0 iRTT=29.252051282051283 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:nsysu->kek pQ=-1.0E50 iRTT=32.51538461538462 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:kek->nsysu pQ=-1.0E50 iRTT=37.231936026936026 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:starlight->vrvs3 pQ=100.0 iRTT=13.86923076923077 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs3->starlight pQ=100.0 iRTT=12.724 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-caltech->usp pQ=-1.0E50 iRTT=94.25742705570292 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:usp->vrvs-caltech pQ=-1.0E50 iRTT=93.8759587217044 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-test->vrvs-pub-ro pQ=-1.0E50 iRTT=100.25059850374065 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-pub-ro->vrvs-test pQ=-1.0E50 iRTT=100.25079800498753 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:starlight->cornell pQ=100.0 iRTT=9.851111111111111 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:cornell->starlight pQ=100.0 iRTT=9.256486486486487 crtState=ACTIVE nextState=ACTIVE
Total cost = 678.3698060609934
MST: Computed tree's tunnels:
  MUST -> IPTun:starlight->sinica pQ=100.0 iRTT=1.0E50 crtState=ACTIVE nextState=MUST_ACTIVATE
  MUST -> IPTun:sinica->starlight pQ=100.0 iRTT=1.0E50 crtState=ACTIVE nextState=MUST_ACTIVATE
  OPTN -> IPTun:vrvs-eu->vrvs-us pQ=100.0 iRTT=0.769 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-us->vrvs-eu pQ=100.0 iRTT=0.5800000000000001 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-caltech->vrvs-test pQ=-1.0E50 iRTT=1.5 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-test->vrvs-caltech pQ=-1.0E50 iRTT=1.02 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:sinica->nsysu pQ=100.0 iRTT=3.5171428571428573 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:nsysu->sinica pQ=100.0 iRTT=4.01 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:starlight->cornell pQ=100.0 iRTT=9.851111111111111 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:cornell->starlight pQ=100.0 iRTT=9.256486486486487 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:starlight->vrvs3 pQ=100.0 iRTT=13.86923076923077 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs3->starlight pQ=100.0 iRTT=12.724 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs3->vrvs5 pQ=-1.0E50 iRTT=15.129628099173553 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs5->vrvs3 pQ=-1.0E50 iRTT=15.128305785123967 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs3->vrvs-caltech pQ=-1.0E50 iRTT=18.0 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-caltech->vrvs3 pQ=-1.0E50 iRTT=18.501081081081082 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-eu->vrvs-pub-ro pQ=100.0 iRTT=19.127614379084967 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-pub-ro->vrvs-eu pQ=100.0 iRTT=18.5 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-eu->funet pQ=100.0 iRTT=29.252051282051283 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:funet->vrvs-eu pQ=100.0 iRTT=29.252051282051283 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:nsysu->kek pQ=-1.0E50 iRTT=32.51538461538462 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:kek->nsysu pQ=-1.0E50 iRTT=37.231936026936026 crtState=INACTIVE nextState=ACTIVE
  OPTN -> IPTun:starlight->vrvs-us pQ=100.0 iRTT=62.5 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-us->starlight pQ=100.0 iRTT=62.5 crtState=ACTIVE nextState=ACTIVE
```

```
  OPTN -> IPTun:starlight->usp pQ=100.0 iRTT=82.50048484848485 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:usp->starlight pQ=-1.0E50 iRTT=82.62657337367625 crtState=INACTIVE nextState=ACTIVE
Total cost = 579.8620819970191
Tunnels to deactivate:
  OPTN -> IPTun:vrvs-caltech->usp pQ=-1.0E50 iRTT=94.25742705570292 crtState=INACTIVE nextState=INACTIVE
  OPTN -> IPTun:usp->vrvs-caltech pQ=-1.0E50 iRTT=93.8759587217044 crtState=INACTIVE nextState=INACTIVE
  OPTN -> IPTun:vrvs-test->vrvs-pub-ro pQ=-1.0E50 iRTT=100.25059850374065 crtState=INACTIVE nextState=INACTIVE
  OPTN -> IPTun:vrvs-pub-ro->vrvs-test pQ=-1.0E50 iRTT=100.25079800498753 crtState=INACTIVE nextState=INACTIVE
Total cost = 388.6347822861355
Tunnels to activate:
  OPTN -> IPTun:starlight->vrvs-us pQ=100.0 iRTT=62.5 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:vrvs-us->starlight pQ=100.0 iRTT=62.5 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:starlight->usp pQ=100.0 iRTT=82.50048484848485 crtState=ACTIVE nextState=ACTIVE
  OPTN -> IPTun:usp->starlight pQ=-1.0E50 iRTT=82.62657337367625 crtState=INACTIVE nextState=ACTIVE
Total cost = 290.1270582221611
Commands SHOULD BE SENT to the Reflectors! (gain/tunnel=24.62693101599359); critical=false
[ Fri Jun 06 22:01:00 EEST 2003 ] --> ReRouting process FINISHED ...
```

The above text shows the previous list of active tunnels, the next list of tunnels and both the list of tunnels to activate and to deactivate. Before each tunnel can be seen the status of the tunnel: whether it MUST be active or its activation/deactivation is optional.

The pQ represents the peerQuality value and iRTT is the internetRTTime value. For each tunnel is shown the current and the next state.

5. Conclusions

VRVS is a highly evolved videoconferencing system providing its users a set of important features: ease of use, scalability, flexibility, efficiency and robustness. One of its major advantages over competitors is the ability of being a portal between different conferencing systems, allowing, for example, H.323 clients to communicate with MBone clients (using vic/rat).

MonALISA is a robust monitoring system, providing upper layers a flexible framework, allowing rapid development of complex clients, ranging from pseudo-clients that store results in a database, to GUI clients started from a web page.

Routing and monitoring clients presented in this project are an example of high level services, created to optimize a worldwide distributed application and to provide help in operating a complex system.

Further work includes developing a distributed routing client that could offer a faster and more accurate response to the critical events that can appear in such a large and intricate system.