"Politehnica" University of Bucharest
The Faculty of Computer Science and Automatic Control

DIPLOMA PROJECT

Distributed Delay Constrained Multicast Algorithms for the MonALISA Framework

Algoritmi distribuiti de multicast pentru mediul MonALISA

UPB Scientific Coordinator:
Prof. Dr. Eng. Valentin CRISTEA
California University of Technology Coordinator:
Dr. Iosif LEGRAND

Author:
Alexandru COSTAN

2004

Table of Contents:

1. Introduction

With recent advances of computer and network technologies, many emerging services such as teleconference, video on demand, distance education, etc., require the network to deliver information to multiple destinations. In a connection oriented packet switched network, point-to-point (unicast) connections may be used to transmit information from a source to a set of destinations by establishing point-to-point connections from the source to each of the destinations. To send a packet to the set of destinations, the source makes copies of the packet and transmits one copy to each of the destinations. This may result in sending two or more copies of the same packet on a single communication link. Obviously, the bandwidth of the communication link can be used more efficiently by eliminating extra copies of the same packet. Thus, it is desirable to setup a point-to-multipoint (multicast) connection to transmit the packet through a tree shaped path and make copies of the packet only at branching nodes in order to make efficient use of the network resources. A packet switched network is said to have multicast capability if it can establish a point-to-multipoint connection to deliver a packet to a group of destinations. It is important for broadband networks to have multicast capability to support these emerging services.

For a service to take advantage of the multicast capability, a multicast connection must be established before information can be delivered from a source node to multiple destinations. The set of nodes consisting of the source node and the destination nodes is called the multicast group. It is desirable to use as little network resources as possible to set up the multicast connection for a multicast group. The amount of resources required by the connection is affected by the route for the multicast connection. The problem of determining the route for a multicast connection is known as the multicast routing problem.

The problem of finding an optimal multicast tree in a point to point network translates to the Steiner Problem in graphs. Since the Steiner problem is NP complete, heuristic approaches are required for path setup. The problem takes a new dimension in Wide Area Networks, where centralized algorithms are not feasible, and distributed schemes are needed. It is also desirable that node participation for path setup is limited to nodes directly involved in the multicast. An additional requirement that comes from the nature of the applications such as videoconferencing that use the multicast support from the network is that of bounded end-to-end delays along any path from the source to each destination in the multicast tree. We present here a heuristic algorithm that ensures delay bounds, is distributed, and produces trees that are only slightly more expensive than those

produced by centralized algorithms. Further, we examine the degradation in performance in case of changing delays along network links (where QoS guarantees on delay are not available), and propose ways of making the tree adaptive to these changes. This dynamic routing approach minimizes resource reservation demands and also makes changing multicast groups permissible.

As multimedia data transfer capability in networks becomes increasingly available, applications such as video conferencing and distance education are gaining popularity. Multicast support is currently available from networks, but the current schemes are concerned only with connectivity, not optimality, and do not provide QoS (Quality of Service) guarantees such as delay bounds and jitter control that are needed for such applications. The bandwidth savings obtained from the use of multicast trees can be maximized by using optimal tree setup algorithms. Future networks will require such schemes to be integrated at lower layers in the protocol stack.

The problem of finding an optimal multicast tree has been shown to be equivalent to the NP-complete Steiner tree problem in graphs. For a large sized network explicitly finding this optimal tree is prohibitively expensive. Heuristic algorithms for setting up multicast trees have been proposed by various authors.

The multicast path setup schemes can also be categorized on the basis of QoS guarantees they provide. Again, most of the centralized and distributed schemes produce trees that are optimal only in terms of a metric on the links but are silent in terms of parameters such as end-to-end delays. A centralized scheme that finds delay constrained multicast trees was proposed by Kompella, and a distributed version of the same was later given by the same authors. This delay constrained distributed scheme is based on pruned MSTs and suffers from the drawback mentioned above. A distributed algorithm that does not require MST construction and requires limited participation by nodes during path setup has been proposed by Bauer and Varma but it produces unconstrained trees.

Finally, schemes can be classified as dynamic and static. Dynamic schemes allow for dynamically changing multicast groups with the possibility of joining and leaving an active multicast. These are also adaptive, i.e. they change the tree in response to changes in the network parameters. Static schemes, on the other hand, build a tree before the beginning of a multicast and the tree is used throughout the lifetime of the multicast. Static schemes also require strict resource reservation to be made at the time of the path setup.

Heuristic algorithms for the dynamic multicast routing problem can be found in the specific literature The dynamic greedy algorithm proposed by Waxman takes the shortest path to an existing multicast tree when adding a node. The source routed shortest path algorithm proposed by Dear finds the shortest path to the source node when adding a node to the multicast group. The Geographic Spread Dynamic Multicast (GSDM) routing algorithm proposed by Kadirire takes the geographic spread defined in the article into account when adding a node. It deals with the node and the nearby nodes which are already in the connection at the same time. The possible routes for these nodes are reconsidered. Among the routes that have the minimal cost, the route with the maximal geographic spread is chosen and that part of the connection is re-routed if necessary. When removing a node, the dynamic greedy, shortest path and GSDM routing algorithms first mark the node “deleted”. If the node is a leaf node in the connection, the node is removed from the multicast group and the node and the branch of which it is a part is

pruned from the multicast tree. A comparison of these dynamic multicast routing algorithms was presented in.

In the dynamic multicast routing problem, the members of the multicast group are dynamically changing. Let us take a snapshot of the multicast group and denote it by S. Suppose that a static multicast routing algorithm is applied to find a multicast tree for S in G. A good static multicast routing algorithm can generally produce better results than a dynamic one. This is because a dynamic multicast routing algorithm usually adds or removes a node based on existing connection. It does not take the multicast group as a whole into consideration and reconstruct the multicast tree as a static algorithm does. Therefore, a static multicast routing algorithm which produces near optimal results can serve as a reference for comparing dynamic multicast routing algorithms. The KMB algorithm was used as the reference algorithm since it can generally produce near optimal results. We also select the KMB algorithm as the reference algorithm.

For dynamic multicast routing algorithms, re-routing a portion of the existing connection may reduce the cost of the multicast connection (e.g., the GSDM routing algorithm). However, re-routing an existing connection may require additional network resources such as bandwidth, buffers to keep the order and integrity of packets. Also, an underlying network system with intentional re-routing capability is required to smoothly re-route existing connections.

2. The MonALISA Services Framework

2.1 Architecture

The MonALISA (Monitoring Agents in A Large Integrated Services Architecture) system provides a distributed monitoring service. MonALISA is based on a scalable Dynamic Distributed Services Architecture which is designed to meet the needs of physics collaborations for monitoring global Grid systems, and is implemented using JINI/JAVA and WSDL/SOAP technologies. The scalability of the system derives from the use of multithreaded Station Servers to host a variety of loosely coupled self-describing dynamic services, the ability of each service to register itself and then to be discovered and used by any other services, or clients that require such information, and the ability of all services and clients subscribing to a set of events (state changes) in the system to be notified automatically. The framework integrates several existing monitoring tools and procedures to collect parameters describing computational nodes, applications and network performance. It has built-in SNMP support and network-performance monitoring algorithms that enable it to monitor end-to-end network performance as well as the performance and state of site facilities in a Grid. MonALISA is currently running around the clock on the US CMS test Grid as well as an increasing number of other sites. It is also being used to monitor the performance and optimize the interconnections among the reflectors in the VRVS system.

We are developing a globally scalable ``Dynamic Distributed Services Architecture'' (DDSA) to serve large physics collaborations. This architecture incorporates many features that make it suitable for managing and optimizing workflow through Data Grids composed of hundreds of sites, with thousands of computing and

storage elements, and thousands of pending tasks, such as those foreseen by the LHC experiments.

In order to scale and operate robustly in managing global, resource-constrained Grid systems, the DDSA framework uses a set of Station Servers, one per facility or site in a Grid, that host a variety of dynamic, agent-based services. The services are registered with, and can be mutually discovered by a lookup service, and they are notified automatically in case of ``events'' signaling a change of state anywhere in a large distributed system. This allows the ensemble of services to cooperate in real time to gather, disseminate, and process time-dependent state and configuration information about the site facilities, networks, and many jobs running throughout the Grid. The monitored information is reported to higher level services, that in turn analyze the information, and take corrective action to improve the overall efficiency of operation of the Grid (through load balancing, for example) or to mitigate problems as needed. The DDSA framework is inherently distributed, ``loosely coupled'' and self-restarting, making it scalable and robust. Cooperating services and applications are able to access each other seamlessly, to adapt rapidly to a dynamic environment (such as worldwide-distributed analysis by hundreds of physicists in a major HEP experiment). The services are managed by an efficient multithreading engine that schedules and oversees their execution, such that Grid operations are not disrupted if one or more tasks (threads) are unable to continue. The system design also provides reliable ``nonstop'' support for large distributed applications under realistic working conditions, through service replication, and automatic re-activation of services. These mechanisms make the system robust against the failure or inaccessibility of multiple Grid components (when a key network link goes down, for example).

A service in the DDSA framework is a component that interacts autonomously with other services through dynamic proxies or agents that use self-describing protocols. By using dedicated lookup services, a distributed services registry, and the discovery and notification mechanisms, the services are able to access each other seamlessly. The use of dynamic remote event subscription allows a service to register to be notified of a selected set of event types, even if there is no provider to do the notification at registration time. The lookup discovery service will then automatically notify all the subscribed services, when a new service, or a new service attribute, becomes available. The code mobility paradigm (mobile agents or dynamic proxies) used in the DDSA extends the remote procedure call and the client server approach. Both the code and the appropriate parameters are downloaded dynamically into the system. Several advantages of this paradigm are: optimized asynchronous communication and disconnected operation, remote interaction and adaptability, dynamic parallel execution and autonomous mobility. The combination of the DDSA service features and code mobility makes it possible build an extensible hierarchy of services capable of managing very large Grids, with relatively little program code.

A prototype implementation of the DDSA was built based on JINI technology. The JINI architecture federates groups of devices and software components into a single, dynamic distributed system; functionality that the future Open Grid Services Architecture (OGSA) will need to include. JINI enables services to find each other on a network and allows these services to participate and cooperate within certain types of operations, while interacting autonomously with clients or other services.

This architecture simplifies the construction, operation and administration of complex systems by:
(1) allowing registered services to interact in a dynamic and robust (multithreaded) way;
(2) allowing the system to adapt when devices or services are added or removed, with no user intervention;
(3) providing mechanisms for services to register and describe themselves, so that services can intercommunicate and use other services without prior knowledge of the services' detailed implementation.

WSDL/SOAP was also included, bindings for all the distributed objects, in order to provide access to the monitoring information from other types of clients and to facilitate a possible future migration to the Open Grid Services Architecture.

An essential part of managing a global Data Grid is a monitoring system that is able to monitor and track the many site facilities, networks, and the many task in progress, in real time. The monitoring information gathered also is essential for developing the required higher level services, and components of the Grid system that provide decision support, and eventually some degree of automated decisions, to help maintain and optimize workflow through the Grid. Therefore was developed the agent-based MonALISA (Monitoring Agents in A Large Integrated Services Architecture) system, based on the DDSA framework. MonALISA is an ensemble of autonomous multi-threaded, self-describing agent-based subsystems which are registered as dynamic services and are able to collaborate and cooperate in performing a wide range of monitoring tasks in large scale distributed applications, and to be discovered and used by other services or clients that require such information.

MonALISA is designed to easily integrate existing monitoring tools and procedures and to provide this information in a dynamic, self describing way to any other services or clients. MonALISA services are organized in groups and this attribute is used for registration and discovery.

2.2 The Data Collection Engine

The system monitors and tracks site computing farms and network links, routers and switches using SNMP, and it dynamically loads modules that make it capable of interfacing existing monitoring applications and tools (e.g. Ganglia, MRTG, Hawkeye). The core of the monitoring service is based on a multithreaded system used to perform the many data collection tasks in parallel, independently. The modules used for collecting different sets of information, or interfacing with other monitoring tools, are dynamically loaded and executed in independent threads. In order to reduce the load on systems running MonALISA, a dynamic pool of threads is created once, and the threads are then reused when a task assigned to a thread is completed. This allows one to run concurrently and independently a large number of monitoring modules, and to dynamically adapt to the load and the response time of the components in the system. If a monitoring task fails or hangs due to I/O errors, the other tasks are not delayed or disrupted, since they are executing in other, independent threads. A dedicated control thread is used to stop properly the threads in case of I/O errors, and to reschedule those tasks that have not been

successfully completed. A priority queue is used for the tasks that need to be performed periodically. A schematic view of this mechanism of collecting data is shown in Figure 1.

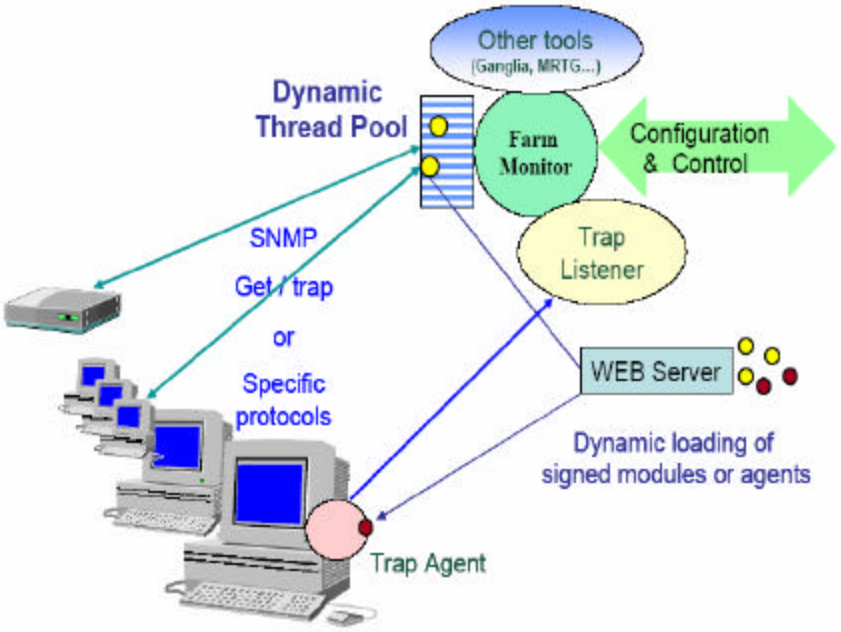


Figure 1. A schematic view of the data collection mechanism based on a multi-threaded engine.

This approach makes it relatively easy to monitor a large number of heterogeneous nodes with different response times, and at the same time to handle monitored units which are down or not responding, without affecting the other measurements. As an example, there were monitored 500 compute nodes performing a request for ~200 metric values per node every 60 seconds. This provided a sustained rate of ~1600 metric values per second collected, using an average of 20 active threads. The number of threads necessary to monitor a complete site is dynamically adjusted, and very dependent on the response time for each node, which is related to its load as well as to the quality of the network connections.

2.3 Data Storage

The collected values are stored in a relational database, locally for each service. The JDBC framework in JAVA offers the flexibility to dynamically load any driver and connect to virtually any relational database. A normalized scheme is used to store the result objects provided by the monitoring modules in indexed tables, which are themselves generated as needed, dynamically. As data are becoming older, we are compressing the values stored in the database by evaluating the mean values on larger time intervals and at the same time keeping the fluctuation range for each parameter.

2.4 Registration and Discovery

Each MonALISA service registers with a set of JINI Lookup Discovery Services (LUS) as part of a group, and having a set of attributes. The LUSs are also JINI services and each one may be registered with the other LUSs. If two LUSs have common groups any information related with a change of state detected for a service in the common group by one is replicated to the other one. In this way it is possible to build a distributed and reliable network for registration of services and this technology allows dynamically adding or removing LUSs from the system. Any service should also provide for registration the code base for the proxies that other services or clients need to instantiate for using it. This approach is used to make sure that the right proxies are used for each service while different versions may be used in a distributed organization at the same time. The registration is based on a lease mechanism that is responsible to verify periodically that each service is alive. In case a service fails to renew its lease, it is removed from the LUSs and a notification is sent to all the services or clients that subscribed for such events.

Any monitor client services is using the Lookup Discovery Services to find all the active MonALISA services running as part of one or several group "communities". It is possible to select the services based on a set of matching attributes. The discovery mechanism is used for notification when new services are started or when services are no longer available. The communication between interested services or clients is based on a remote event notification mechanism which also supports subscription.

The client application connects directly with each service it is interested in for receiving monitoring information. To perform this operation, it first downloads the proxies for the service it is interested in from a list of possible URLs specified as an attribute of each service, and than it instantiate the necessary classes to communicate with the service. This procedure allows each service to correctly interact with other services.

2.5 Predicates, Filters and Alarm Agents

The clients can get any real-time or historical data by using a predicate mechanism for requesting or subscribing to selected measured values. These predicates are based on regular expressions to match the attribute description of the measured values a client is interested in. They may also be used to impose additional conditions or constrains for selecting the values. In case of requests for historical data, the predicates are used to generate SQL queries into the local database. The subscription requests will create a dedicated thread, to serve each client. This thread will perform the matching test for all the predicates submitted by a client with the measured values in the data flow. The same thread is responsible to send the selected results back to the client as compressed serialized objects. Having an independent thread per client allows sending the information they need, fast, in a reliable way and it is not affected by communication errors which may occur with other clients. In case of communication problems these threads will try to reestablish the connection or to clean-up the subscriptions for a client or a service which is not anymore active.

Monitoring data requests with the predicate mechanism is also possible using the WSDL/SOAP binding from clients or services written in other languages. The class description for predicates and the methods to be used are described in WSDL and any client can create dynamically and instantiate the objects it needs for communication.

Currently, the Web Services technology does not provide the functionality to register as a listener and to receive the future measurements a client may want to receive.

Other applications or clients may also use the Agent Filters to receive the information they need. The Agent Filter is a java module which can be dynamically deployed to any MonALISA service, and is design to perform a dedicated data processing task on local data (by subscribing with a predicate to the data flow) and returns back the processed information periodically. The MonALISA service provides the run time environment for these agents which must be digitally signed by a trusted certificate. As an example, such filters are used to compute the aggregate IO traffic in a farm, or to provide the number of nodes which are free. The same thread used for handling the predicate subscription is used for sending the filtered results back to each client.

Dynamically loadable alarm agents, and agents able to take actions when abnormal behavior is detected, are currently being developed to help with managing and improving the working efficiency of the facilities, and the overall Grid system being monitored.

2.6 Administration of Services

MonALISA also provides a secure mechanism (SSL with X.509 certificates) for dynamic configuration, using a dedicated GUI, of farms / network elements, and support for other higher level services that aim to manage a distributed set of facilities and/or optimize workflow. It allows reconfiguring any monitoring services by adding new nodes, network elements or clusters and at the same time to dynamically loaded into the system any new monitoring module as needed. It also allows stopping or suspending any monitoring module. Adding dynamically new monitoring modules is important for debugging and understanding the way certain applications perform.

The Administration interface connects to a service using Remote Method Invocation over SSL. X.509 certificates for trusted administrators are imported in the keystore of each service and they are used to establish a SSL connection based on a client authentification procedure. The administrative GUI can be stated automatically from the global web start client if it used by a trusted administrator. When the administrator loads his private key into the global GUI client it automatically gets administrative rights on the services that imported his certificate in the trust keystore.

2.7 Automatic Update for Services

MonaALISA is currently deployed on many sites and maintaining and updating such applications may require a significant effort. For this reason there was developed a mechanism in MonALISA that allows us to automatically update the monitoring service. A dedicated thread is used to periodically check for updates of the distribution. Alternatively a remote event notification can be used to notify only selected services to perform an update. When such an event is detected, the running service will trigger a restart operation. When a MonALISA service is started, it is using the web start mechanism to describe an application and all its dependencies and constrains into a XML file (jnpl). This will perform an automatic download of all the packages which were updated and will check all the necessary constrains to run the application. All the files

downloaded in this way must be digitally signed by a developer for which the certificate is imported in the trust keystore. This can be done when the MonALISA service is used for the first time.

All the running services, as well as the services which may be stated after an update was done will run the last "published" version and this is done in a secure way. Users may start a MonALISA service with the auto update flag switch off.

2.8 Monitoring Data Processing Farms

MonALISA is now deployed and operating round the clock monitoring the US CMS Test Grid and an increasing number of other sites. The MonALISA Web repository is now accumulating historical data for the US CMS Tier1 and Tier2 centers at Fermilab, Caltech, UCSD, and the University of Florida, as well as the production farms at CERN, at the Academia Sinica in Taiwan (ATLAS), and at the Polytechnic University in Bucharest. As an example, the number of nodes loaded on the US-CMS farms during a week is presented in the next figure.

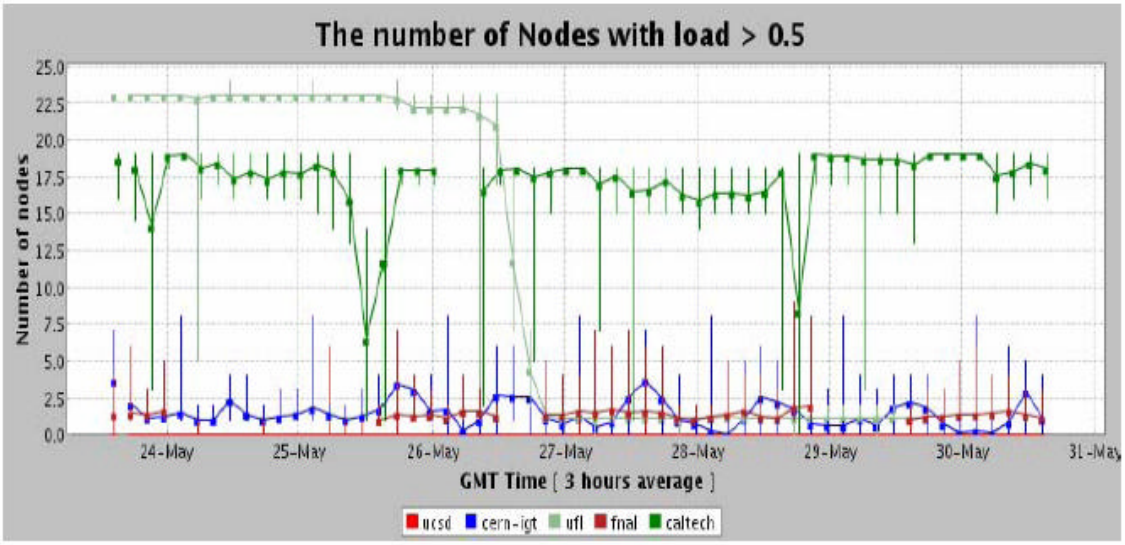


Figure 2: A global plot of the US-CMS farms showing the number of nodes with load higher than 0.5 during a period of one week. These plots are created with the web service repository.

There are also monitored the network traffic on the US-CERN production link, and the distribution of the traffic into the major networks and links with which we peer: EsNet, Abilene, Mren, StarTAP, the US-CERN DataTAG link, the CERN-Geant link, Taiwan-Chicago, and Bucharest- Budapest. In addition to the directed measurements performed on routers, MonALISA was interfaced to provide access to the Internet End to End Performance Measurements (IEPM-BW).

There are currently monitored the batch queuing systems at CERN (LSF) and at Caltech (PBS). From these modules we can report the number of (selected types) jobs running, pending or those which exit with errors.

2.9 Monitoring the VRVS System

The Virtual Rooms VideoConferencing System (VRVS) is an enhanced web based video conferencing system which is using a set of reflectors distributed world wide for an efficient real-time distribution of the audio and video streams.

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 were 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. A dedicated GUI for the VRVS version 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 bad). 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 next figure.

In addition to dedicated monitoring modules and filters for the VRVS system, we developed agents able to supervise the running of the VRVS reflectors automatically. This will be particularly important when scaling up the VRVS system further. In case a VRVS reflector stops or does not answer correctly to the monitoring requests, the agent will try to restart it. If this operation fails twice the Agent will send an email to a list of administrators. These agents are the first generation of modules capable of reacting and taking well defined actions when errors occur in the system. These agents, capable to take action in the system, may be dynamically loaded. For security reasons such agents must be digitally signed by developers with trusted certificates, declared for each running service.

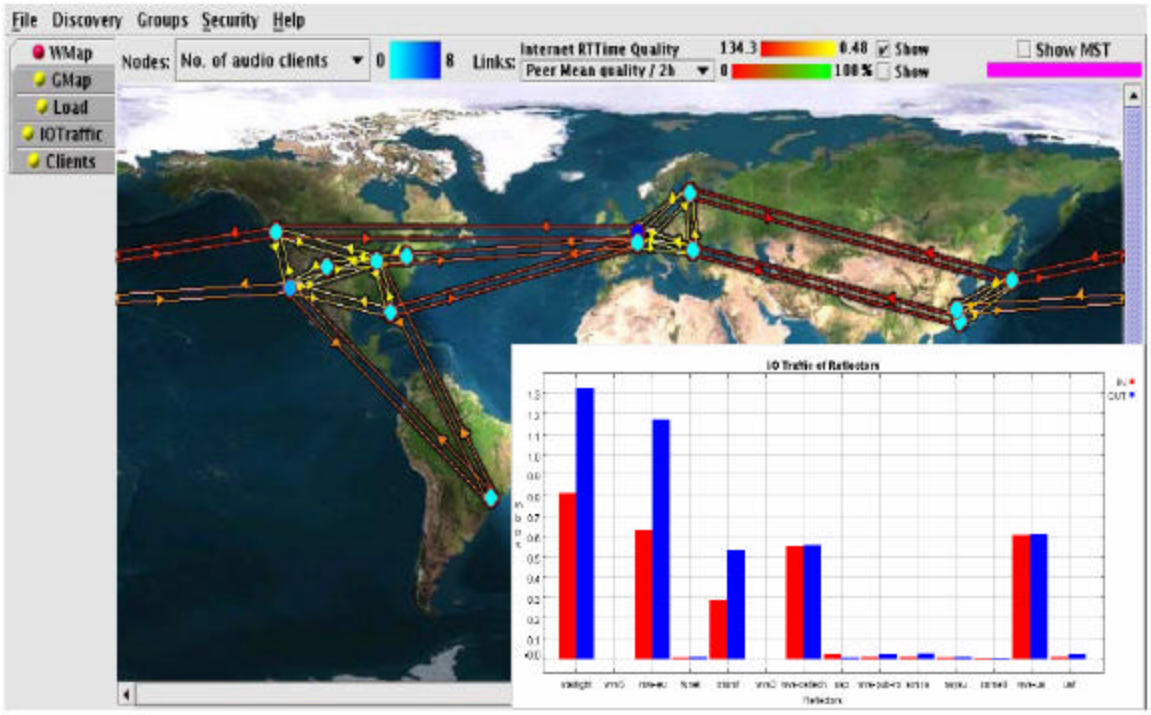


Figure 3: Monitoring the VRVS System.

2.10 Existing Approach on Dynamic Routing

In the current MonALISA framework, the multicast path setup is used in VRVS, a videoconferencing system based on a set of servers called reflectors that route the audio/video streams to the participating clients, for monitoring and controlling the VRVS reflectors in order to enhance the quality of the service.

A ReflRouter client was developed 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 there were 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.

The reflectors and all these possible peer connections we are measuring define a graph (as shown in the next figure).

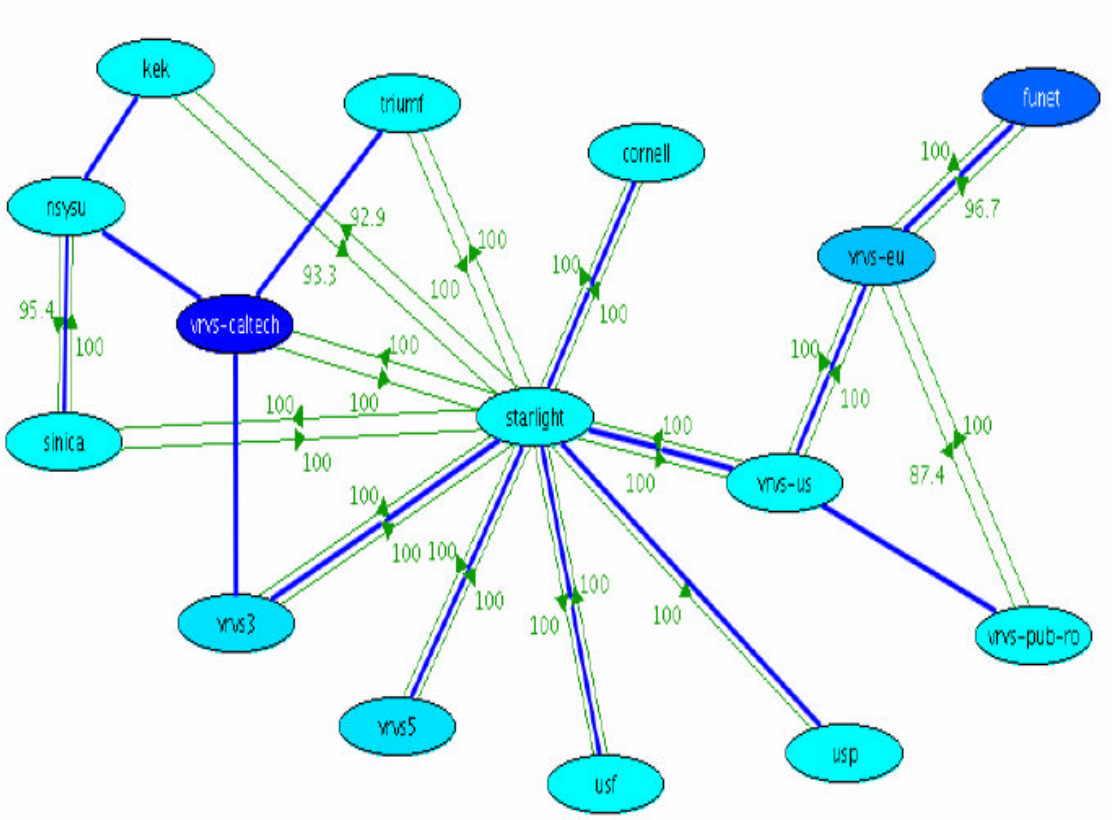


Figure 4: The graph defined by the reflectors and the peer connections.

The best routing path for reapplication of the multimedia streams is defined as a Minimum Spanning Tree (MST). 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:

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

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.

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.

For implementation, the Boruvka's algorithm was used, as it is also appropriate for a parallel 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). In the figure before an example of a dynamically MST for connecting the VRVS reflectors was presented.

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 conex. 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 iterative, 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.

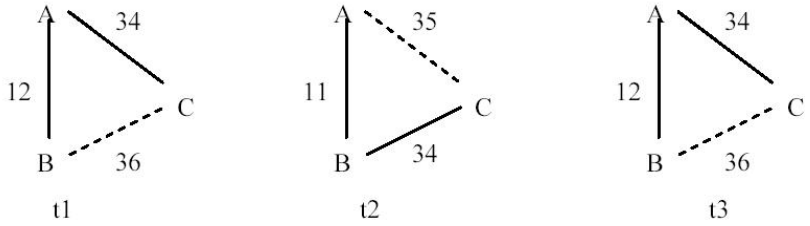


Figure 5: Active link oscilation.

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 significant 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.

This is an example of a high level service developed to optimize a real-time world wide distributed application and to help in operating such complex systems. These developments are transforming the VRVS system into a new class of large scale distributed systems with real time constraints.

The MonALISA framework is a means of carrying out the development of this system, both in terms of its operational characteristics (heuristic, self-discovering, autonomous) and the relatively short development time required for implementing a distributed monitoring and management system of this scale and complexity.

3. Our Contribution

The major disadvantage of the existing approach, as most of the existing schemes, is that of being centralized, i.e. they assume that information about each link in the network is available at one node. While the centralized schemes are fast and produce cheap trees, the requirement of all information to be present at one node is problematic in large sized networks as the overhead to collect and store the data is prohibitive. Among the distributed schemes that are available, many are based on the distributed minimum spanning tree algorithms, and work by first finding the MST of the whole network and then pruning off all edges and leaves that are not involved in the multicast. These, however, require participation of all the nodes in the network, and have an unsatisfactory theoretical upper bound on competitiveness.

The scheme we propose is distributed and produces delay constrained trees that are little more expensive than those produced by centralized Steiner heuristics. An extension to this scheme makes it adaptive to changing delays along links and permits dynamic *joins* and *leaves*. The scheme requires very little information in addition to that which is already maintained in routing tables for current protocols.

3.1 Problem Formulation

The delay constrained multicast path setup problem in a network can be formulated as follows. The network is modeled as a graph G(V,E) with cost and delay functions defined on the links. The capacities of the links are assumed to be fixed and known. The cost metric on the links could be any combination of monetary cost and network related parameters.

INPUTS :
C(e) : $C : E \rightarrow N$, gives cost of edge e

D(e) : D : E $\rightarrow$ N, gives the delay on e
s : Source node
S : Set of destinations
Δ : Max. permissible delay from source to destination. $\Delta \in N$

OUTPUT:
T, a tree rooted at s spanning all nodes in S.

CONSTRAINT
$\Sigma D(e) < \Delta \quad \forall v \in \Sigma$
$e \in P(s,v)$
where P(s,v) is the set of edges along the path from source s to destination v.

OBJECTIVE
Minimize: Σ C(e) .
$e \in T$

Using the above formulation, the proposed algorithm computes the (static) multicast tree. In the extension of the algorithm where dynamically changing link delays and multicast groups are permitted, the formulation is different in that the delay function D(e) and the set S are also functions of the (wall clock) time. D(e) is then not a specified function, but is calculated using a statistical model of the link.

4. The Distributed Delay Constrained Multicast Algorithms

Establishing a multicast tree in a point-to-point network of switch nodes, can be modeled as the NP-complete Steiner problem in networks. In this chapter, we introduce and evaluate two distributed algorithms for finding multicast trees in point-to-point data networks.

The first algorithm is based on Steiner heuristics, the shortest path heuristic (SPH) and the Kruskal-based shortest path heuristic (K-SPH), and have the advantage that only the multicast members and nodes in the neighborhood of the multicast tree need to participate in the execution of the algorithm. We compare this algorithm by simulation against the second one, a baseline algorithm, the pruned minimum spanning-tree heuristic, which is the basis of many previously published algorithms for finding multicast trees. We compare the competitiveness of the two and decide which one to use for implementation and integration in MonALISA framework according to the found results.

4.1 The Steiner Heuristics Algorithm

Multicasting is likely to take an increasingly important role in data networks in the future. Previous authors have established that the multicast tree problem can be

modeled as the Steiner problem in networks, referred to hereafter as the SPN, and that finding explicit solutions in large networks is prohibitively expensive. A number of good, inexpensive, centralized heuristics exist for the SPN and have been reviewed extensively elsewhere. Most of the algorithms proposed in the literature for SPN are serial in nature.

However, a few distributed heuristics exist in the literature. Many of these algorithms are based on reducing the SPN to the minimum spanning tree problem, referred to here as the MST, and use a distributed minimum spanning tree algorithm such as the one described by Gallager, Humblet, and Spira. A Steiner tree is created by pruning the minimum spanning tree of unnecessary leaves and branches. Distributed Steiner heuristics based on a minimum spanning tree algorithm suffer from two drawbacks: first, all the nodes in the network must participate in the execution of the algorithm. This may be impractical in a large network with sparse multicast groups. Second, the theoretical upper bound on competitiveness of a pruned MST tree to that of an optimal Steiner tree has been shown to be s + 1, where s is the number of non-multicast nodes. Here competitiveness is defined to be the ratio of the sum of the heuristic tree's edge weights to that of an optimal tree. Thus the competitiveness of a multicast tree decreases with the size of the multicast group. In comparison, the equivalent theoretical upper bound for shortest path heuristic competitiveness for the Steiner tree problem is 2, regardless of the multicast group size. Our empirical evidence suggests that pruned MST heuristics often produce solutions of inferior quality to those produced by shortest path Steiner heuristics.

In this chapter, we present two distributed algorithms for the Steiner problem in networks. The algorithms are based on the shortest path heuristic (SPH) and the Kruskal-based shortest path heuristic (K-SPH). We provide analytical bounds for their message and convergence time complexities and compare their simulation results against those from a pruned MST algorithm.

We choose the distributed MST algorithm due to Gallager, Humblet, and Spira as our baseline algorithm for comparison. This algorithm is perhaps the simplest of all pruned MST algorithms, yet produces Steiner trees that are representative of other, more elaborate pruned MST heuristics. The distributed heuristics are compared on the basis of three criteria: competitiveness, the number of messages exchanged, and convergence time.

The distributed heuristics have the advantage that the algorithm is initiated by only the multicast members and requires the participation of only nodes in the neighborhood of the multicast tree, instead of all the nodes in the network as required by the pruned MST approach. However, limiting the execution of the algorithm to a subset of the network nodes results in a substantial increase in the number of messages generated in our algorithms, with a corresponding increase in convergence time.

Before continuing, we make the following basic definitions and notations. Z is the set of multicast destinations, S is the set of non-multicast nodes V - Z, P_{ij} is the shortest

path between nodes i and j, d_{ij} is the distance of the shortest path between nodes i and j, and C_t is the cost of tree t (the sum of t's edge weights). Graph distances will be defined as follows: the distance between two nodes is the distance of the shortest path between them. Likewise, the distance between a node and a tree is the distance of the shortest path between the node and any node in the tree. Finally, the distance between two trees is the distance of the shortest among all paths between any node in one tree and any node in the other tree. We append the weight of an edge or path with the index of its destination node in determining shortest paths so that, in case of a tie, the actions of the individual nodes would be consistent. Since we do not allow multiple edges between node pairs, this ensures that all the nodes select the same edge or path, given the same set of edge weights.

To be suitable for distributed implementation, a heuristic must satisfy four criteria. It must (i) use the existing routing information available at each node in the network, (ii) use minimal computational and network resources, (iii) require a minimum of coordination between neighbors, and (iv) limit itself to nodes directly involved in the multicast. Of the centralized heuristics evaluated, we chose two heuristics for distributed implementation: SPH and K-SPH.

Distributed heuristics SPH and K-SPH are designed to run as asynchronous, independent processes running one per node in a network. Each distributed heuristic assumes that the routing tables in each node are up-to-date; no topology changes occur during the execution of the algorithm; the network is connected; every node has a unique index; each multicast member has knowledge of the indices of all other multicast members; and each multicast member is able to determine the distance to every other node from its routing table.

Heuristic SPH is inherently a serial algorithm, since there is only one subtree expanding itself at any time during the execution of the algorithm and nodes must join the tree serially. Heuristic K-SPH, on the other hand, allows many of the join operations to proceed in parallel. The latter, however, is substantially more difficult to parallelize because of the significant amount of coordination that may be needed while combining subtrees. In the following, we present distributed K-SPH first, followed by a similar distributed implementation of SPH.

4.1.1 Distributed Heuristic K-SPH

Like its centralized version, distributed K-SPH starts with a forest of Z multicast members (Z-nodes) and connects them pairwise into successively larger subtrees until a single multicast tree remains or no further connections are possible. We refer to the subtrees during the execution of the algorithm as fragments. Thus, at the beginning of the algorithm, there are Z fragments, each a trivial subtree consisting of one Z-node.

At any instant during the execution of the algorithm, each node in the network is either part of a fragment or has not been yet been included in the multicast tree. Note that every Z-node is always a fragment node and every non-member node (S-node) is initially

a non-fragment node. When two fragments merge, the nodes in both fragments and the nodes in the path connecting them become the fragment nodes of the new, merged fragment.

Each fragment has a fragment leader coordinating the activities of the fragment. This fragment leader is the fragment Z-node with the lowest index. Each fragment leader executes the same finite state machine shown in the next figure.

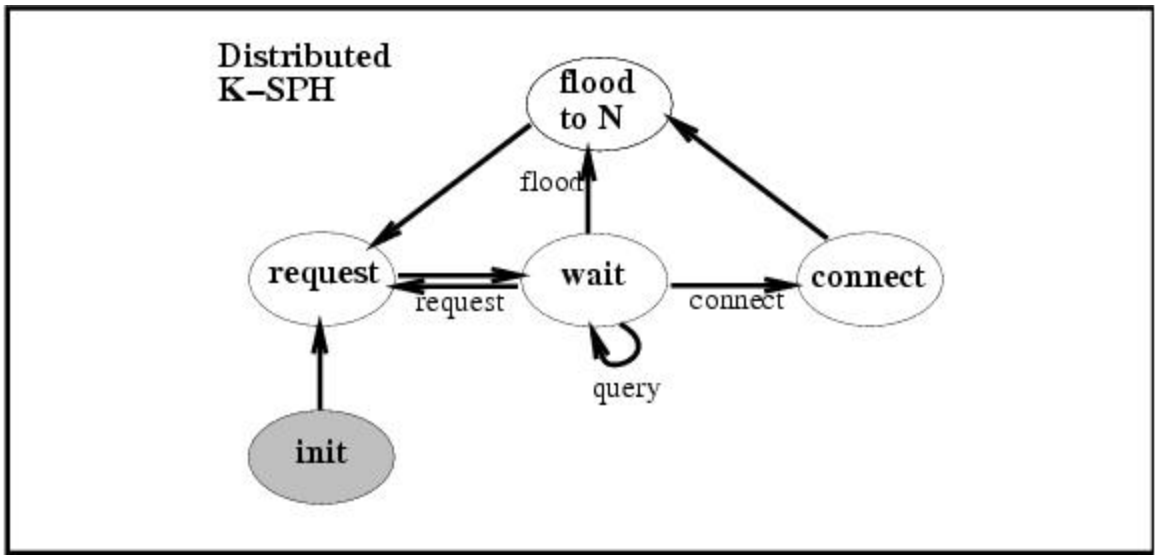


Figure 6: The finite state machine for fragment leaders.

Other fragment node executes a simplified version of the leader's finite state machine. Initially, each multicast member is the leader of its own one-node fragment; when two fragments merge, leadership is assigned to the fragment leader with the lower index. To identify fragments uniquely, each fragment has the same index as its leader and each fragment node is aware of its fragment index.

During the execution of the algorithm, each fragment, guided by its leader, attempts to merge with its closest neighboring fragment. This is accomplished in two steps: a discovery step and a connection step. During the discovery step, the leader gathers and updates its information on other fragments and graph nodes. Based on the information gathered, it determines the closest fragment to merge with. During the connection step, it communicates with the closest neighbor fragment's leader, requesting a merge. This closest fragment leader is simply the Z-node with the same index as the closest fragment. If accepted, the leader with the lowest index attempts to connect the two fragments. Regardless of the outcome (the request is rejected, the subtrees are connected, or the connection attempt fails), the cycle repeats until the algorithm terminates.

Distributed K-SPH processes running on each node rely on shortest path information available at its node, as well as information maintained by the fragment leaders. The shortest path information stored at each node is the distance, next hop and next-to-last hop of the shortest path to other nodes. This path information is similar to

that stored by distance-vector routing protocols and may already be available in each node's routing tables. If so, distributed K-SPH may use the existing tables, avoiding unnecessary extra storage at each node. If not, this information may be derived using a distance vector routing algorithm. The next-to-last hop table allows distributed K-SPH processes to derive the entire shortest path between nodes by recursively considering the path. Each node also stores the index of its fragment. Initially, only multicast nodes have a fragment index (its own index). Each leader maintains additional shortest path information for its fragment. This information augments the shortest path information at each node. For example, the leader stores only the distance, and the head and tail of the shortest path between its fragment and every other fragment. The additional details necessary to build the path between fragments is stored at the head of the path, a node in the leader's subtree (Note that the shortest path between fragments need not start or end at a leader node).

When distributed K-SPH starts, each Z-node, the leader of its own trivial one-node fragment, already knows its distance to every other fragment as provided by the initial distance tables and no discovery step is necessary. Instead, each distributed K-SPH leader starts with the connection step, described as follows.

The connection step

During the connection step, each leader attempts to connect its fragment with the closest fragment, known as its preferred fragment. It does so by sending a merge request message to the leader of the preferred fragment (That is, the Z-node with the same index as the preferred fragment). A leader receives one of three responses to its request: accept, reject, or busy. We consider each response in turn below. The busy response occurs when a fragment's request arrives at its preferred fragment while the latter is in its discovery step described below. While in the discovery step, a leader cannot accept or reject merge requests, as it is in the process of updating its information. Instead, the busy response is sent. When the requesting leader receives the busy response, it repeats its request in the hopes or reaching its preferred fragment after its discovery step. A leader will repeat its connection request until it receives either an accept or reject response.

When a leader receives a connection request from a fragment other than its preferred fragment, it returns a reject message. This message forces the requesting fragment into a discovery step to find another preferred fragment. If a former leader node receives a connection request from any fragment, it returns a reject message since a connection is no longer possible to the old fragment.

When two fragments exchange merge requests with one another, each responds by returning an accept message. Once an accept message is sent, the fragment may not leave the connect step or accept a request from another fragment until the connection attempt completes. Of the two leaders in a connection attempt, only the leader with the lower index acts, while the leader with the higher index waits passively for the result of the connection attempt. This is because if the connection attempt succeeds, the leader with the lower index becomes the leader of the new, merged fragment. The leader with

the lower index initiates a connection attempt by sending a message to the head of the shortest path between the two fragments, a node in its fragment. In its message to the head of the shortest path, the leader specifies the tail of the shortest path, a node in the other fragment. Upon receiving the connect message, the head node sends a connect message along its shortest path to the tail node.

When two fragments A and B merge the shortest path used to join them must have its head in one fragment, its tail in the other, and pass through only non-fragment nodes.

The connect message may either reach the target fragment or be blocked; blocking occurs when the message reaches a node in a third fragment before reaching the target fragment. In either case, a status message returns to the head of the shortest path.

Consider the case of a successful connection first. In this case, the connect message travels down the shortest path, reserving intermediate nodes in the path as part of the new fragment, until it reaches a node in the target fragment. It is possible that the first node reached in the target fragment is not the specified tail. This occurs when the leader's shortest-path information for other fragments is stale and an intermediate node in the selected path is already part of the other fragment. The connect message stops at the first node in the target fragment it reaches and sends a status message back along the shortest path to the head of shortest path. Each reserved, non-fragment node along the path receives the status message, includes itself in the new, merged fragment, and passes the status message along the path. The head of the shortest path forwards the status message to its leader, now the leader of the new, merged fragment. This completes the connection step and the leader enters the discover step described below.
Now consider the case of an unsuccessful connection. In this case, the shortest path between the fragments is blocked. This occurs because a node in the shortest path, the blocked node, has become part of a third fragment. When the connect message reaches the blocked node, the blocked node returns a status message along the shortest path to the head of the path. As each intermediate node receives the status message, it removes its reservation from the new fragment, becoming an non-fragment node once again. The head of the shortest path forwards the status message to the leader. The leader informs the other fragment leader of the connection failure by sending a reject message. This completes the connection step. Both leaders then enter the discover step described below.

States *request*, *wait* and *connect* in the figure before comprise the connection step.

The discovery step

The discovery step accomplishes three tasks:

(1) it informs every node in the fragment of its new fragment index;
(2) it gathers fragment information about nodes close to the fragment;
(3) it refreshes its information on shortest paths to other fragments.

Each fragment leader achieves these tasks by performing a multicast on its fragment rooted at itself. In the multicast message, the leader includes the fragment index, the distance to the preferred fragment and shortest paths to other fragments. As each node in the fragment receives the multicast, it updates its fragment index, queries nearby nodes and passes the multicast message to its children. Only those nodes that lie within the distance from this fragment to the preferred fragment are queried for fragment index information. The objective of queries to nearby nodes is to find fragment nodes closer than those already known by the leader. Fragment B's leader believes that fragment C is the closest fragment. During fragment B's discovery step, it instructs fragment nodes to query those nodes closer than fragment C.

This distance is the distance between node 3, the head of the path to fragment C, and node 4, its tail, and is marked by the dotted circles around each of fragment B's nodes. Since nodes 1 and 2 fall within one such circle, they receive queries and fragment B's leader discovers the closer fragment A. Queries could be sent to all nodes in the graph, but are limited to nodes within a small distance for two reasons:

- a set distance avoids broadcast storms
- new shortest paths discovered should be shorter than those already available.

The discovery step is implement by state flood-to-N in the figure above.

4.1.2 Distributed SPH

The distributed shortest path heuristic is a special case of distributed K-SPH described in the previous section. In distributed SPH, any one of the multicast members may act as the source of the multicast, referred to here simply as the source node. In contrast to distributed K-SPH, only one fragment, the source fragment grows, connecting multicast members to itself until all the multicast members are part of the same fragment. The heuristic terminates when a single tree remains.

In SPH, the preferred fragment of every fragment is always the source fragment. The sole exception, of course, is the source fragment itself which prefers its closest fragment. Using the same connection step outlined for heuristic K-SPH, the source fragment merges with its closest fragment. As the source fragment grows, it uses the same discovery step to determine the new, closest fragment. The source fragment never changes its index. This preserves the source fragment's original index so that non-source fragments never need to change their preferred fragment index. As a consequence, non-source fragments do not enter the discovery phase. In all other respects, distributed SPH is very similar to distributed K-SPH.

4.1.3 Message and convergence bounds

Bounds for messages passed and convergence time are summarized in the next tables.

Bound	Distributed K-SPH	Distributed SPH
Lower Bound	$Z \log Z$	Z^2
Upper Bound	ZN	ZN

Messages bounds for distributed heuristics K-SPH, SPH and Fixup.

Bound	Distributed K-SPH	Distributed SPH
Lower Bound	$\log Z$	DZ
Upper Bound	DZ	DZ

Convergence-time bounds for distributed heuristics K-SPH, SPH and Fixup.

The distributed versions of SPH and K-SPH provide inferior solutions compared to their centralized versions because of the lack of global topology information in each node in the former. However, the degradation in the competitiveness was small in our test networks. In fact, the competitiveness produced by distributed K-SPH was often superior to that of centralized SPH.

The pruned minimum spanning tree algorithm, on the other hand, requires participation from every node of the network, a condition difficult to satisfy in practice in a large wide-area network.

Viewed from the perspective of convergence-time, however, the pruned MST heuristic enjoys an advantage over shortest path heuristics SPH and K-SPH. The convergence time for the solutions produced by pruned MST algorithm fell well within a much narrower range as compared to the results for distributed K-SPH and SPH.

On comparing the SPH and K-SPH algorithms, it is interesting to observe that the algorithms had the same level of communication complexity in terms of the number of messages generated, yet the range of convergence times produced by K-SPH was significantly tighter. This is primarily due to the disparate approaches used by the algorithms in growing the multicast tree. Distributed SPH grows the tree by adding one multicast member at a time to the source fragment, concentrating much of the work at the source, while distributed K-SPH allows multiple fragments of the tree to combine in parallel. This allows distributed K-SPH to provide lower convergence times without increasing the number of messages substantially.

Because the convergence times for distributed K-SPH are higher than those of the pruned MST algorithm by as much as 10 times, we shall use this second one, presented as follows in our implementation for the MonALISA framework.

4.2 The Pruned Minimum Spanning-Tree Heuristic

The distributed multicast tree setup algorithm has three distinct phases. Phase 1 is a 'Tree Construction' phase, Phase 2 is a 'Tree Repair' phase. At the end of Phase 2, the tree setup is complete and the multicast session can begin. Phase 3 of the algorithm handles changes in link parameters and/or changes in the multicast group, and may be invoked at any point during the lifetime of a multicast.

4.2.1 Phase 1: Tree Construction

Given the multicast group and the cost and delay functions on the edges, this phase constructs a tree rooted at the source and spanning all destination nodes. The tree construction can be done using the distributed Highly Asynchronous MST Algorithm.

4.2.1.1 A highly asynchronous minimum spanning tree

We present a distributed protocol for obtaining a minimum spanning tree in an asynchronous network. We assume that each edge has a distinct weight associated with it. When the protocol terminates, each node knows which edges incident on it are in the minimum spanning tree.

This protocol maintains a spanning forest of trees (referred to as *fragments*), each of which is a subtree of the MST. Fragments are merged over their minimum weight outgoing edges until a single fragment that spans the entire network remains. In order to keep the message complexity low, each fragment has a *level number* associated with it which is a measure of the number of nodes in the fragment.

We present a protocol, *CompMST*, which requires O(min (N, (D+d) log N) time and O(E+N log N/log log N) messages where D is the maximum degree of a node and d is the diameter of the MST. To arrive at this protocol we first present a protocol *Async*. In Async, a fragment does not wait for another fragment to reach a particular level before it can combine with it. The protocol takes at most O(min(N,(D+d)log N) time and $O(N^2)$ messages. The features of Async and those from the other protocol are combined to obtain CompMST. The requirement of balanced growth is relaxed and a fragment at level l has to wait for a neighbour fragment to reach a level greater or equal to $l - log\ l$ before combining with it. The CompMST protocol behaves like the classic protocol when the fragment size is small and like Async when the fragment size reaches N/log N.

Problem formulation

The network is modeled like an undirected graph with N nodes and E edges. All nodes are assumed to have distinct identities. We assume that all the edges have distinct weights and each process knows the weight of all edges incident on it. The nodes communicate via messages. Messages are not lost and they arrive at their destination within finite but unpredictable time. Further, messages sent over an edge arrive in the order in which they are sent. On the initiation of the protocol, we assume that each process knows the weight of each edge incident on it. On the termination of the protocol, each node knows which edges incident on it belong to the minimum spanning tree.

The protocol maintains a forest of rooted trees (referred to as fragments). The root of the fragment is the root of the corresponding tree and the root's identity is used to identify the fragment. The *best edge* of a fragment is the minimum weight edge among all edges leading out of the fragment.

Each fragment has a level number associated with it. Fragments containing only a single node are at level 0. When two fragments at level l merge, a new fragment at level l+1 is created. For such a level numbering scheme, it can be shown that a fragment with the level number l contains at least 2^l nodes. Therefore, the level number of a fragment cannot exceed log N. The level of a node is the level number of the fragment to which it belongs.

A node may be in one of the following states:

- the initial state Sleeping (a node is in this state until it starts executing the protocol)
- the state Find while participating in a fragment's search for its best edge
- the state Found at other times.

The algorithm starts with each node as a separate fragment. Fragments are merged iteratively on the best edges and the algorithm terminates when only one fragment which spans the entire network remains. In each iteration, a fragment determines its best edge and combines it with the fragment at the other end of the edge. When two fragments having the same best edge, e, combine to form a new fragment, the node with the larger identity among the two end-points of e becomes the new root of the combined fragment.

Each node i keeps the identity of edges incident on it sorted according to their weights in a list. The node picks each time the first edge from this list and sends a *test* message to the other end of the edge with its fragment identity and level number. When node j receives a test message from node i, it behaves as follows:

- If the fragment identity of j agrees to that of i, then i and j belong to the same fragment and therefore j responds by sending the *reject* message. When a reject message is received the current edge is deleted from the list and the process repeats with the next edge in the list.
- If the fragment identities of i and j are different and the level number of j is greater than or equal to that of i, it sends an *accept* message to i.

- Otherwise j delays the response until its level number becomes at least as large as that of i.

A minimum spanning tree protocol

In this section we describe the Async protocol. Each iteration is executed in two phases. In the first phase, the fragment identity is propagated to all sites in the fragment. After this phase is over, the root initiates the second phase for finding the best edge.

First-phase The root of a fragment initiates the first phase by sending an $initiate_1$ message with the fragment identity (which is the identity of the root) as a parameter to its children. On receiving the message $initiate_1$ a node updates its fragment identity and propagates $initiate_1$ to its children. When the $initiate_1$ message reaches a leaf node, it sends a finish message to its parent. An intermediate site waits for a finish message from all children before sending a finish message to its parent. When the root receives the finish message from all children, it knows that all nodes in the fragment know the current fragment identity. The root initiates the second phase.

Second phase The root of a fragment initiates the second phase by sending $initiate_2$ to its children. In this phase the best edge of the fragment is found. A node sends a test message over an edge to ascertain that the edge is outgoing. However, the reply to a test message *is not delayed* (because if the receiving node is in the same fragment, then it must know the correct fragment identity since the first phase of the iteration has completed). After a node has determined its local best edge it propagates this edge weight towards the root using *report* messages. The root picks the edge with the minimum weight among the local best edges and sends a *change-root* message to the node in the fragment with this as an incident edge. This node becomes the new root of the fragment and sends a *connect* message over the best edge in an attempt to combine with the fragment at the other end.

Consider the case when a *connect* message from a site i in fragment F reaches a site j, which is in fragment G. We have the following cases:

- if j receives $initiate_1$ and has not sent a *finish* message, then j treats (i,j) as an edge of the fragment and sends $initiate_1$ to i. Further, site j waits for a finish message from i before sending its finish message. In this case, nodes in F are absorbed in G as a part of the current iteration of G
- if j has already sent its *finish* message then the response to the connect message is delayed. If (i,j) is also the best edge of G then G will also send a connect message over this edge and F and G will merge ending the iteration. The node with the larger identity among the two end-points of the best edge will become the new root of the combined fragment and will initiate the next iteration. Otherwise when j gets $initiate_1$ message during the first phase of the next iteration, it will send an $initiate_1$ message to i and as a result F will be absorbed as a part of that iteration.

Hence, fragments are absorbed only while a site is executing the first phase and no new sites are added to a fragment while in the second phase.

The composite protocol

CompMST behaves like the protocol presented in the literature when the fragment size is small and like Async when the fragment size becomes large. In contrast to Async, the level numbers are explicitly stored by the sites and we require that the response to a test message sent by a node i at a level l to a node j to be delayed only if the level number of j is less than *l-log l*. Since log l increases with l, the protocol becomes more asynchronous as l increases. In CompMST the level number of a fragment is proportional to the amount of time it has to wait before updating its level number. The changes required to Async to obtain this behaviour are explained in the following:

First-Phase The initiator site sends $initiate_1$ message to its children with its current level number and the fragment identity. On receiving initiate1 a site updates its level number and fragment identity and propagates $initiate_1$ to its children. The number of nodes are counted while propagating the *finish(count)* messages, where count is the number of nodes in the subtree rooted at the node sending the message. A leaf sends a *finish(1)* message to its parent. After site i has received a $finish(count_m)$ message from each child m, it sums up the counts received from the children, adds one to it and sends the resulting number in a finish message to its parent. The first phase terminates after the initiator receives a finish message from each child. Let M be the sum of the counts received from the children by the initiator. The initiator then updates its level number to log(M+1). This may be greater than the level number previously stored in the initiator due to fragments absorbed during this first phase.

Second Phase is modified as follows. The initiator propagates its new level number in the $initiate_2$ messages and sites update their level numbers on receiving this message. The level number of a node is included in the test message sent by it. If a node j receives a test message from a node with level l and j's fragment identity differs from the one received then the response is delayed by j until its level number becomes at least l-log l.

In addition, we use a protocol *Update* which allows a node to update the level number and fragment identity of the nodes in its fragment. The initiator site starts the protocol by sending the *update* message to its children with the fragment identity and level number in it. On receiving *update(level,id)*, site i updates its level number and fragment identity and propagate the *update* message to its children.

Consider the case when a connect message from i in fragment F at level l_i is received by a node j in fragment G at level l_j. If j has already sent a connect message to i (so that both F and G have the same best edge) then F and G are merged. If j>i then j becomes the root of the combined fragment and initiates a new iteration. Otherwise, i becomes the new root. If j has not sent a connect message to i then j behaves as follows:

- $l_i > l_j$

In this case, site j delays response to the connect message until its level becomes at least l_i (the connect message is then handled as described in case 2 below) or it sends a connect message to i (in this case the fragments are merged as described above)

- $l_i <= l_j$

a. Site j has received the *initiate*$_1$ and has not sent the finish message:

In this case site j propagates *initiate*$_1$ to i and waits for a finish message from i before sending a *finish* message to its parent. Thus F is absorbed in G and nodes in F participate in the current iteration of the protocol in G). The number of nodes in F are therefore inceluded in updating the number of G.

b. Site j has sent the finish message and $l_i < l_j - log\ l_j$

In this case since j has already sent the *finish* message, the nodes in F will not be included in updating the level number of G. Therefore, we require that i counts the number of nodes in F and reports that count to j before the *connect* message is processed by j. To do this, j sends a message to i instructing it to count the number of nodes, and temporarily refrains from sending a report message to its parent in G if it has not already sent it. Let C be the fragment rooted at i after the completion of *first-phase* and *count* be the number of nodes in C which is reported to i when first phase completes.

- if *log (count)* $>= l_j$ then site i decides to keep C distinct from G. It notifies j of this fact so that j can resume execution of the second phase in G. In addition, site i updates its level to *log(count)* and initiates *Update* to update the level number of the nodes in C
- if *log(count)* $< l_j$ then G absorbs C. In this case, i notifies j of its decision to get absorbed and then updates its fragment identity to G and level number to j. Further it initiates *Update* to update the level number and fragment identity for the nodes in its subtree. If j has not already sent the *report* message then nodes in C participate in the second phase of the current iteration of G. When j receives *initiate*$_2$ it propagates it to i and waits for a report message from i before sending its own *report* message.

c. Site j has sent the finish message and $l_j >= l_i >= l_j - log\ l_j$

In this case nodes in F cannot participate in the current iteration of G. As the previous site i updates its fragment identity to i and initiates first-phase. However, site j does not refrain from sending messages to its parent while counting is in progress. After the first phase is over, sit ei update its level number to *max(l_j, log (count))* where count is the number of nodes reported to i when the first phase complets. Site i then initiates the update procedure of the level number of nodes in its fragment.

4.2.1.2 Optimizing the Tree

The Async MST results in a multicast tree which minimizes a metric, i.e. it can be used to form a tree that is a minimum cost tree or a minimum delay tree. Two separate approaches can be adopted for optimizing (or meeting the bound on) the parameter not already optimized :

(a) Cost First Heuristic (CFH) : which optimizes the cost using Async MST and then 'repairs' the tree wherever delay bounds are violated.
(b) Delay First Heuristic (DFH) : which first obtains the minimum delay tree using Async MST and then attempts to reduce the cost by making changes wherever delays are unnecessarily small.

The two methods can be expected to give different cost-competitiveness and differ in their running complexities. The cost-first heuristic, while can be expected to give lower cost trees, may result in too much modification to ensure delay bounds, which may reduce its cost-competitiveness. The delay-first heuristic ignores the costs at the first step, but needs fewer modifications since the delay bounds have already been met. DFH will never cause a total degeneration of the tree built by the Async MST. CFH, on the other hand, may lead to this situation if modifications of the tree are unable to guarantee delay bounds. Our experiments with the two approaches show that DFH gives better overall results, and hence CFH is not explored further.

4.2.2 Phase 2: Tree Repair

Phase 2 begins when the source **s** sends a DISCOVER_DELAY packet to all its children using the tree setup in phase 1. As this packet trickles down and reaches a node **v**, the node **v** and the packet is marked with the delay encountered on the path from **s** to **v**. Each node stores this 'delay mark'.

Next, all destination nodes whose marked delay exactly matches with the delay bound, send a NO_CHANGE packet upwards to their parents. As soon as a NO_CHANGE packet travels over a link, the link is declared permanent, which means that this link will not be removed from the tree. The packet is forwarded upwards till the source.

The leaf nodes that have delays marked smaller than the delay bound (no leaf node can have a mark larger than the bound since this is a minimum delay tree, else it is impossible to meet the delay bound in the network), calculate the SLACK (= delay bound -delay mark), and send a SLACK_PACKET upwards to their parents. This packet contains the sender's index, as well as the slack. As soon as the packet reaches a node (other than the originator) which is also a multicast member, the links from the originator to this node are discarded, and the links that come within slack plus delay of the discarded link's are considered for inclusion. These are discovered by a localized flooding to neighbours. If more than one links meet the criteria, the cheapest are chosen. Delay marks are revised, and according to new marked delays, links may get declared permanent. However, it may happen that no cheaper paths with delays within slack amount of the delay on current links can be found. In that case, the packet is sent further up and the cycle repeats. In case of new links being found, the balance slack, if any, is sent further up.

This method tries to achieve cost reduction by changes in levels closest to the leaf nodes whenever possible. This strategy is useful since the closer we are to the source, the larger is the number of destinations receiving the packets from the link, which makes the possibility of changing that link without disrupting the whole tree very small. Keeping the minimum delay links near the source is also desirable for adaptability of the tree (phase 3). It is thus a trade-off between the amount of modification and the cost. It is to be noted that the original links are discarded but not forgotten, as these may be required in the next phase.

4.2.3 Phase 3: Tree Adaptation

The third phase of the algorithm is invoked in two situations, when the delays along links change or when a node joins or leaves a multicast. If the delay on a link (that is a part of the tree) changes, the node receiving packets from that link sends a DISCOVERY-REQUEST packet to its parent. The parent marks the link on which this packet is received and forwards the packet upwards, till the packet reaches the source. The source then sends the DISCOVER_DELAY packet, and phase 2 is repeated in the partial tree formed by the marked links.

In this phase, the new situation that can occur is that of negative SLACK values. These are handled differently. The first node to receive a SLACK_PACKET carrying a negative value removes the link along which the packet was received, thus breaking the tree into two subtrees. The links that were obtained through K-SPH in phase 1 are now brought back into the tree to connect the two subtrees via the shortest delay path. The *discover delay* and *slack-reduction* steps are repeated till slack values become positive. It may be noted that positive slack values are handled just as in phase 2.

Dynamic *joins* and *leaves* are handled using a Weighted Greedy Approach and then running phase 2 on the partial tree consisting of new links. The performance of this scheme is better than the weighted greedy approach because of the cost reduction achieved by phase 2, while it retains the simplicity as re-optimization is built into the algorithm itself.

5. A Genetic Algorithm for Steiner Tree Optimization with Multiple Constraints Using Prüfer Number

Besides the heuristic approach we can use genetic algorithms which find the solution more rapidly, but not always the optimized solution.

5.1 Problem formulation

A network is modeled as a directed, connected graph $G = (V, E)$, where V is a finite set of *vertices* (network nodes) and E is the set of *edges* (network links) representing connection of these vertices. Let n = card (V) be the number of network nodes and l = card (E) be the number of network links. The link $e = (u, v)$ from node $u \in V$ to node $v \in V$ implies the existence of a link $e' = (v, u)$ from node v to node u. Three non-

negative real value functions are associated with each link *e* ($e \in E$): cost *C(e):E*? *R+*, delay *D(e):E*? *R+*, and available bandwidth *B(e):E*? *R+*. The link cost function, *C(e)*, may be either monetary cost or any measure of the resource utilization, which must be optimized. The link delay, *D(e)*, is considered to be the sum of switching, queuing, transmission , and propagation delays. The link bandwidth, *B(e)*, is the residual bandwidth of the physical or logical link. The link delay and bandwidth functions, *D(e)* and *B(e)*, define the criteria that must be constrained (bounded). Because of the asymmetric nature of the communication networks, it is often the case that *C(e)* ? *C(e ')*, *D(e)* ? *D(e ')*, and *B(e)* ? *B(e ')*. A multicast tree *T*(*s*, *M*) is a sub-graph of *G* spanning the source node $s \in V$ and the set of destination nodes $M \in V$-{*s*}. Let *m* =card(*M)* be the number of multicast destination nodes. We refer to *M* as the *destination group* and $\{s\} \cup M$ the *multicast group*. In addition, *T*(*s*, *M*) may contain relay nodes (Steiner nodes), that is, the nodes in the multicast tree but not in the multicast group. Let *PT*(*s*, *d*) be a unique path in the tree *T* from the source node *s* to a destination node $d \in M$.

The total cost of the tree *T*(*s*, *M*) is defined as the sum of the cost of all links in that tree and can be given by

$$C(T(s,M)) = \sum_{e \in T(s,M)} C(e)$$

The total delay of the path $P_T(s,d)$ is defined as the sum of the delay of all links along $P_T(s,d)$

$$D(P_T(s,d)) = \sum_{e \in P_T(s,d)} D(e)$$

The bottleneck bandwidth of the path $P_T(s,d)$ is defined as the minimum available residual bandwidth at any link along the path:

$$B(P_T(s,d)) = \min \{B(e),\ e \in P_T(s,d)\}$$

Let Δ_d be the delay constraint and B_d the bandwidth constraint of the destination node d. The bandwidth delay-constrained least-cost multicast problem is defined as minimization of C(T(s,M)) subject to

$$D(P_T(s,d)) \le \Delta_d \forall d \in M$$

$$B(P_T(s,d)) \le B_d \forall d \in M$$

5.2 Genetic algorithms

Genetic algorithms are the most widely known types of evolutionary computation methods today. In general, a genetic algorithm has five basic components

1. An encoding method, that is a genetic representation (genotype) of the solutions to the program
2. A way to create an initial population of individuals (chromosomes)
3. An evaluation function, rating solutions in terms of their fitness and a selection mechanism

4. The genetic operators (crossover and mutation) that alter the genetic composition of offspring during reproduction
5. Values for the parameters of genetic algorithm

A general structure of the genetic algorithm is as follows:

***Procedure**: Genetic Algorithms*
Begin
t := 0;
initialize P(t); {P(t) is the population of individuals in generation t}
evaluate P(t);
While** (not termination condition) **do
Begin
    recombine P(t) to yield C(t); {creation of offspring C(t) by means of genetic operators}
    evaluate C(t);
    select P(t + 1) from P(t) and C(t);
    t := t + 1;
End
End

The general structure of a genetic algorithm

5.3 Genotype: modified Prufer numbers

A spanning tree T has n nodes, n=3, and its Prüfer number, $P(T)$, is an n-2 digit number. Encoding of the Steiner tree by the Prüfer number is more difficult than encoding of the spanning tree.

Special difficulty arises because:
• The Steiner trees contain a variable number of nodes in the range from m+1 to n, and their associated Prüfer numbers include between m-1 and n-2 digits.
• In the spanning case, the set of eligible nodes for consideration in decoding algorithm is the set of all nodes that are not appeared in the Prüfer number. In the Steiner case, this rule is not applicable.

We adopt the encoding/decoding algorithms of the Prüfer numbers to be suitable for the Steiner tree problems. The next two figures show these algorithms, which convert a Steiner tree to its associated Prüfer number and vice versa. Let i be the lowest numbered leaf (node of degree 1) in T and j be the predecessor of i. The Prüfer number is built up by appending j to the right of $P(T)$ and removing i and the edge (i, j) from T. Thus i is no longer considered at all and if i was the only successor of j, then j has become a leaf. This process is repeated, until only two nodes remain in T to be considered. Thus, $P(T)$ is built and read from left to right. Let P be the set of nodes that are part of the Prüfer number, $P(T)$. In our modified Prüfer number decoding algorithm (see Figure 3), we consider that the set of eligible nodes, R, be all nodes in the multicast group, $\{s\} \cup M$, that are not member of P, i.e., $R=(\{s\} \cup M) \cap P'$.

```
Procedure: Convert a Tree to its Prüfer number
Begin
  P(T) := null;
  While (more than two nodes remain in T) do
    Begin
      Find i, the lowest numbered leaf in T;
      Let j be the predecessor of i;
      Append j to the right of P(T);
      Remove i and the edge (i, j) from T;
    End
End
```

```
Procedure: Convert a Prüfer number to a tree
Begin
  R := ({s}∪M)∩P';  {Let the set of eligible nodes, R, be all nodes in the multicast group that are not part of P(T)}
  While (one or more digits remain in P(T)) do
    Begin
      Find i, the lowest numbered eligible node in R;
      Let j be the leftmost digit of P(T);
      Add the edge (i, j) to T;
      Remove j from P(T);
      Remove i from R;     {Designate i as not no longer eligible}
      If (j∉ P(T)) then    {if j dos not occur anywhere in what remains of P(T)}
        Add j to R;        {Designate j as eligible}
    End
  {Now, there are exactly two nodes, i and j, in R which are still eligible for consideration}
  Add the edge (i, j) to T;
End
```

Figure 7: The Prufer and modified Prufer number decoding algorithm.

The Prüfer encoding establishes a one-to-one correspondence (non-redundancy property) between k-node trees and the set of all string of k-2 digits. This means that we can use only (k-2)-digit permutation (short encoding property) to uniquely represent a tree where each digit is an integer between 1 to k inclusive. The transformation back and forth between edges and Prüfer numbers can be carried out in $O(n \log n)$ with the aid of a heap.

5.3 The pre-processing phase

Before starting the genetic algorithm, we can remove all the links, which their bandwidth are less than the minimum of all required thresholds (Min $\{B_d \mid \forall d \in M\}$). If in the refined graph, the source node and all the destination nodes are not in a connected sub-graph, this topology does not meet the bandwidth constraint. In this case, the source should negotiate with the related application to relax the bandwidth bound. On the other

hand, if the source node and all the destination nodes are in a connected sub-graph, we will use this sub-graph as the network topology in our GA-based algorithms.

5.4 The initial population

Random individual creation algorithm: In this algorithm, a linked list is constructed from the source node *s* to one of the destination nodes. Then, the algorithm continues from one of the unvisited destinations and at each node the next unvisited node is randomly selected until one of the nodes in the previous sub-tree (the tree that is constructed in the previous step) is visited. The algorithm terminates when all destination nodes have been mounted to the tree.

```
Procedure: random individual creation
Begin
n := 1;
First := True;
While (n<=Number of Destinations) do
Begin
        Initialize the n-th link list;
        If (First) then
                Current-node := Source
        Else
                Current-node := One of unvisited Destinations;
        GTM := Temporary matrix of the network graph;
        Add the Current-node to the n-th link list;
        Link-list-comp := False;
        While (Not Link-list-comp) do
        Begin
                k := Number of connected nodes to the Current-node in GTM;
                If (k=0) then
                Begin
                        Remove the Current-node in the n-th link list;
                        Remove the link between the Current-node and the previous node in
                        Gold;
                        Current-node := previous node in the n-th link list;
                        GTM := Gold
                End
                Else
                Begin
                        i := a random natural number in interval [1,k];
                        Add the i-th node to the n-th link list;
                        Gold := GTM;
                        Remove all links to the Current-node in GTM;
                        Current-node := the i-th node;
                        If (First) then
                                If (Current-node is one of the destinations) then
                                Begin
                                        Link-list-comp := True;
```

Make an individual by n-th link list;
n := n+1;
First := False;
Mark the found destination as a visited destination
End
Else
If *(the Current-node is a node in one of the previous link lists(for example j-th link list))* ***then***
{ *if the Current-node has a connection to the source node, this link has higher priority}*
Begin
n-th link list := j-th link list from the source node to found position + Inverse (n_th link list);
Link-list-comp := True;
Add the n-th link list to the individual;
n := n+1;
Mark this destination as a visited destination
End
End *{Else}*
End *{inner while}*
End *{outer while}*
End *{procedure}*

5.5 The fitness function

We define the fitness function for each individual, the tree $T(s, M)$, using the penalty technique, as follows:

$$F(T(s,M)) = \frac{\alpha}{\sum_{e \in T(s,M)} C(e)} \prod_{d \in M} \phi(D(P(s,d)) - \Delta_d) \prod_{d \in M} \phi(B(P(s,d)) - B_d)$$

$$\phi(z) = \begin{cases} 1 & z \le 0 \\ \gamma & z > 0 \end{cases}$$

where a is a positive real coefficient, f(z) is the penalty function and ? is the degree of penalty (? is considered equal to 0.5).

5.6 Selection

The selection process used here is based on spinning the roulette wheel *pop-size* times, and each time a single chromosome is selected as a new offspring. The probability *Pi* that a parent *Ti* is selected is given by:

$$p_i = \frac{F(T_i)}{\sum_{j=1}^{pop-size} F(T_j)}$$

Where $F(Ti)$ is the fitness of the Ti individual.

5.7 Crossover

The algorithm uses two crossover schemes for recombination of two individuals, which represent Steiner trees:

Crossover I: Let $\{P_F(s, d1), P_F(s, d2), \ldots, P_F(s, dm)\}$ be the set of paths from the source node s to all destination nodes in T_F and $\{P_M(s, d1), P_M(s, d2), \ldots, P_M(s, dm)\}$ be the same set in T_M. Since, we have found these paths for all individuals in the current population for calculating the fitness function of them, the algorithm will not be complex.

A fitness function for the path $P(s, di)$ based on the total cost, the total delay and the minimum bandwidth of the path using the penalty technique, is defined as follows:

$$F(P(s,d_i)) = \frac{\alpha}{\sum_{e \in P(s,d_i)} C(e)} \phi(D(P(s,d_i)) - \Delta_{d_i}) \phi(B(P(s,d_i)) - B_{d_i})$$

$$\phi(z) = \begin{cases} 1 & z \le 0 \\ \gamma & z > 0 \end{cases}$$

where a is a positive real coefficient, f(z) is the penalty function and ? is the degree of penalty (? is considered equal to 0.5). According to the crossover probability of Pc, two multicast trees $T_F(s,M)$ and $T_M(s, M)$ are selected as parents and the crossover operation produce an offspring $T_O(s, M)$. Each individual may be recombined with its right individual and its left individual through the crossover operator. For each destination node di, we compute the fitness of $P_M(s, di)$ and $P_F(s, di)$ and select the better path.

Procedure*: The crossover operator*
Begin
    For *i:=1* ***to*** *m* ***do { m is the number of destination nodes}***
    If *F(PM(s, di)) > F(PF(s, di))* ***then***
        PO(s, di) := PM(s, di)
    Else
        PO(s, di) := PF(s, di);
    Current-tree := PO(s, d1);
    For *i:=2* ***to*** *m* ***do***
    Begin
        Previous-node := s;
        Start-node := s;
        Current-node := The second node in the PO(s, di);

```
        New-link := False;
        While (Previous-node <> di) do
        Begin
            If the Current-node does not exist in the current-tree then
            Begin
                Add the link between the Current-node and the Previous node to
                the current-tree;
                New-link := True;
            End
            Else
            Begin
                If the New-link = True then
                Remove all link from Start-node to the Previous-node in PO(s,
                di) in the current-tree;
                Start-node := Current-node
                New-link := False;
            End
            Previous-node := Current-node;
            If there is another node in PO(s, di) then
                Current-node := the next node in the PO(s, di)
        End
    End
End
```

Crossover II: In this scheme, it is used a simple one-point crossover. The constructed offspring do not necessarily represent Steiner trees. Then, the effective and fast *check and recovery* algorithm proposed in Q. Zhang, Y.W. Lenug, "An orthogonal genetic algorithm for multimedia multicast routing" is used to connect the separate sub-trees in the offspring and also connecting the absent nodes of multicast group to the final tree.

5.8 Mutation

There are two following algorithms for mutation operator:

Mutation I: The mutation procedure randomly selects a subset of nodes and breaks the multicast tree into some separate sub-trees by removing all the links that are incident to the selected nodes. Then, the effective and fast *check and recovery* algorithm is used to connect the separate sub-trees and also connecting the absent nodes of multicast group to the final tree.

Mutation II: According to the mutation probability *Pm*, the mutation procedure randomly selects an infeasible chromosome from one of the following class (If the first class is empty, a chromosome is selected from the second class and so on)

- Class 1: The chromosomes, which do not satisfy the delay and the bandwidth constraints.
- Class 2: The chromosomes, which do not satisfy the delay constraint.

• Class 3: The chromosomes, which do not satisfy the bandwidth constraint.

If all chromosomes in the current population satisfy both of the QoS constraints, we exit from the mutation procedure. Then, we select only the paths that satisfy both of the QoS constraints in the selected chromosome.

6. Implementation

The implementation of the algorithms uses the existing framework, namely the JINI network technology.

6.1 JINI

The Jini system architecture consists of three categories: *programming model*, *infrastructure*, and *services*. The original *Jini Architecture Specification* defines these categories as follows:

The infrastructure is the set of components that enables building a federated Jini system, while the services are the entities within the federation. The programming model is a set of interfaces that enables the construction of reliable services, including those that are part of the infrastructure and those that join into the federation.

Originally, the programming model defined models for leasing, event notification, and transactions. The basic infrastructure consisted of the discovery/join protocol and the lookup service. Previous versions of the starter kit delivered implementations of the following Jini technology-enabled services (Jini services):

- Lookup Service (reggie)
- Transaction Manager Service (mahalo)
- Lease Renewal Service (norm)
- Event Mailbox Service (mercury)
- Lookup Discovery Service (fiddler)

6.1.1 Discovery and Join

Entities that wish to start participating in a distributed system of Jini™ technology-enabled services and/or devices, known as a *djinn*, must first obtain references to one or more Jini lookup services. The protocols that govern the acquisition of these references are known as the *discovery* protocols. Once these references have been obtained, a number of steps must be taken for entities to start communicating usefully with services in a djinn; these steps are described by the *join* protocol.

Terminology

A *host* is a single hardware device that may be connected to one or more networks. An individual host may house one or more Java™ virtual machines[1] (JVM).

Throughout this document we make reference to a *discovering entity*, a *joining entity*, or simply an *entity*.

- A *discovering entity* is simply one or more cooperating objects in the Java programming language on the same host that are about to start, or are in the process of, obtaining references to Jini lookup services.
- A *joining entity* comprises one or more cooperating objects in the Java programming language on the same host that have just received a reference to the lookup service and are in the process of obtaining services from, and possibly exporting them to, a djinn.
- An *entity* may be a discovering entity, a joining entity, or an entity that is already a member of a djinn; the intended meaning should be clear from the context.
- A *group* is a logical name by which a djinn is identified.

Since all participants in a djinn are collections of one or more objects in the Java programming language, this document will not make a distinction between an entity that is a dedicated device using Jini technology or something running in a JVM that is hosted on a legacy system. Such distinctions will be made only when necessary.

Host Requirements

Hosts that wish to participate in a djinn must have the following properties:

- A functioning JVM, with access to all packages needed to run software written to the Jini specifications
- A properly configured network protocol stack

The properties required of the network protocol stack will vary depending on the network protocol(s) being used. Throughout this document we will assume that IP is being used, and highlight areas that might apply differently to other networking protocols.

Protocol Overview

There are three related discovery protocols, each designed with different purposes:

- The *multicast request protocol* is employed by entities that wish to discover nearby lookup services. This is the protocol used by services that are starting up and need to locate whatever djinns happen to be close. It can also be used to support browsing of local lookup services.
- The *multicast announcement protocol* is provided to allow lookup services to advertise their existence. This protocol is useful in two situations. When a new lookup service is started, it might need to announce its availability to potential clients. Also, if a network failure occurs and clients lose track of a lookup service, this protocol can be used to make them aware of its availability after network service has been restored.

- The *unicast discovery protocol* makes it possible for an entity to communicate with a specific lookup service. This is useful for dealing with non-local djinns and for using services in specific djinns over a long period of time.

The discovery protocols require support for multicast or restricted-scope broadcast, along with support for reliable unicast delivery, in the transport layer. The discovery protocols make use of the Java platform's I/O libraries to exchange information in a platform-independent manner.

Discovery in Brief

Groups

A group is an arbitrary string that acts as a name. Each lookup service has a set of zero or more groups associated with it. Entities using the multicast request protocol specify a set of groups they want to communicate with, and lookup services advertise the groups they are associated with using the multicast announcement protocol. This allows for flexibility in configuring entities: instead of maintaining a set of URLs for specific lookup services that needs to be updated if any of the services change address, an entity can use a set of group names.

Although group names are arbitrary strings, it is recommended that DNS-style names (for example, "eng.sun.com") be used to avoid name conflicts. One group name, represented by the empty string, is predefined as the *public* group. Unless otherwise configured, lookup services should default to being members of the public group, and discovering entities should attempt to find lookup services in the public group.

The Multicast Request Protocol

The multicast request protocol, shown in the next figure, proceeds as follows:

1. The entity that wishes to discover a djinn establishes a TCP-based server that accepts references to the lookup service. This server is an instance of the *multicast response* service.

2. Lookup services listen for multicast requests for references to lookup services for the groups they manage. These listening entities are instances of the *multicast request* service. This is *not* an RMI-based service;
3. The discovering entity performs a multicast that requests references to lookup services; it provides a set of groups in which it is interested, and enough information to allow listeners to connect to its multicast response server.

4. Each multicast request server that receives the multicast checks if it is a member of a group specified in the request; if it is, it connects to the multicast response server described in the request,

and uses the unicast discovery protocol to pass an instance of the lookup service's implementation of net.jini.core.lookup.ServiceRegistrar.

At this point, the discovering entity will have obtained one or more remote references to lookup services.

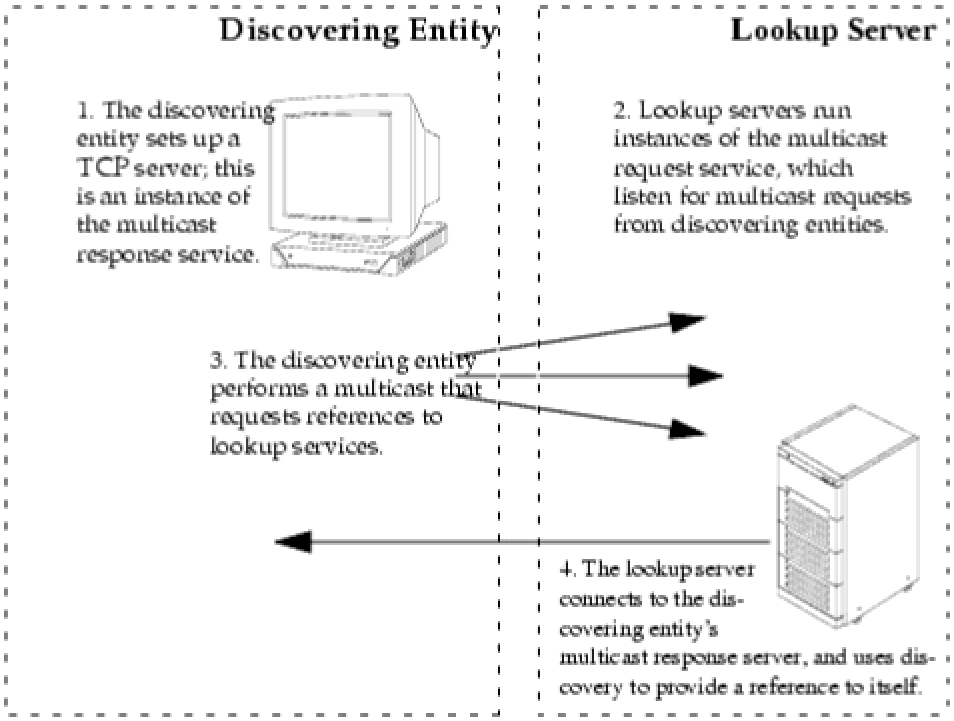


Figure 8: The Multicast Request Protocol

The Multicast Announcement Protocol

The multicast announcement protocol follows these steps:

1. Interested entities on the network listen for multicast announcements of the existence of lookup services. If an announcement of interest arrives at such an entity, it uses the unicast discovery protocol to contact the given lookup service.

2. Lookup services prepare to take part in the unicast discovery protocol (see below) and send multicast announcements of their existence at regular intervals.

The Unicast Discovery Protocol

The unicast discovery protocol works as follows:

1. The lookup service establishes a TCP-based server, on which it listens for incoming connections. When a connection is made by a client, the lookup service reads in request data sent by the client; if

the request is acceptable, the lookup service responds by sending an object that implements the net.jini.core.lookup.ServiceRegistrar interface over the connection.

2. An entity that wishes to contact a particular lookup service uses known host and port information to establish a connection to that service. It sends a discovery request and, if the request is accepted, receives a ServiceRegistrar object in response.

6.1.2 Entry

Entries are designed to be used in distributed algorithms for which exact-match lookup semantics are useful. An entry is a typed set of objects, each of which may be tested for exact match with a template.

Operations

A service that uses entries will support methods that let you use entry objects. In this document we will use the term "operation" for such methods. There are three types of operations:

- *Store operations*--operations that store one or more entries, usually for future matches.
- *Match operations*--operations that search for entries that match one or more templates.
- *Fetch operations*--operations that return one or more entries.

It is possible for a single method to provide more than one of the operation types. For example, consider a method that returns an entry that matches a given template. Such a method can be logically split into two operation types (match and fetch), so any statements made in this specification about either operation type would apply to the appropriate part of the method's behavior.

Serializing Entry Objects

Entry objects are typically not stored directly by an entry-using service (one that supports one or more entry operations). The client of the service will typically turn an Entry into an implementation-specific representation that includes a serialized form of the entry's class and each of the entry's fields. (This transformation is typically not explicit but is done by a client-side proxy object for the remote service.) It is these implementation-specific forms that are typically stored and retrieved from the service. These forms are not directly visible to the client, but their existence has important effects on the operational contract. The semantics of this section apply to all operation types, whether the above assumptions are true or not for a particular service.

Each entry has its fields serialized separately. In other words, if two fields of the entry refer to the same object (directly or indirectly), the serialized form that is compared for each field will have a separate copy of that object. This is true only of different fields of an entry; if an object graph of a particular field refers to the same object twice, the graph will be serialized and reconstituted with a single copy of that object.

A fetch operation returns an entry that has been created by using the entry type's no-arg constructor, and whose fields have been filled in from such a serialized form. Thus, if two fields, directly or indirectly, refer to the same underlying object, the fetched entry will have independent copies of the original underlying object.

This behavior, although not obvious, is both logically correct and practically advantageous. Logically, the fields can refer to object graphs, but the entry is not itself a graph of objects and so should not be reconstructed as one. An entry (relative to the service) is a set of separate fields, not a unit of its own. From a practical standpoint, viewing an entry as a single graph of objects requires a matching service to parse and understand the serialized form, because the ordering of objects in the written entry will be different from that in a template that can match it.

The serialized form for each field is a java.rmi.MarshalledObject object instance, which provides an equals method that conforms to the above matching semantics for a field. MarshalledObject also attaches a codebase to class descriptions in the serialized form, so classes written as part of an entry can be downloaded by a client when they are retrieved from the service. In a store operation, the class of the entry type itself is also written with a MarshalledObject, ensuring that it, too, may be downloaded from a codebase.

Generally speaking, storing a remote reference to a non-persistent remote object in an entry is risky. Because entries are stored in serialized form, entries stored in an entry-based service will typically not participate in the garbage collection that keeps such references valid. However, if the reference is not persistent because the referenced server does not export persistent references, that garbage collection is the only way to ensure the ongoing validity of a remote reference. If a field contains a reference to a non-persistent remote object, either directly or indirectly, it is possible that the reference will no longer be valid when it is deserialized. In such a case the client code must decide whether to remove the entry from the entry-fetching service, to store the entry back into the service, or to leave the service as it is.

In the Java(TM) 2 Platform, activatable object references fit this need for persistent references. If you do not use a persistent type, you will have to handle the above problems with remote references. You may choose instead to have your entries store information sufficient to look up the current reference rather than putting actual references into the entry.

Templates and Matching

Match operations use entry objects of a given type, whose fields can either have *values* (references to objects) or *wildcards* (null references). When considering a template *T* as a potential match against an entry *E*, fields with values in *T* must be matched exactly by the value in the same field of *E*. Wildcards in *T* match any value in the same field of *E*.

The type of *E* must be that of *T* or be a subtype of the type of *T*, in which case all fields added by the subtype are considered to be wildcards. This enables a template to match entries of any of its subtypes. If the matching is coupled with a fetch operation, the fetched entry must have the type of *E*.

The values of two fields match if MarshalledObject.equals returns true for their MarshalledObject instances. This will happen if the bytes generated by their serialized form match, ignoring differences of serialization stream implementation (such as blocking factors for buffering). Class version differences that change the bytes generated by serialization will cause objects not to match. Neither entries nor their fields are matched using the Object.equals method or any other form of type-specific value matching.

You can store an entry that has a null-valued field, but you cannot match explicitly on a null value in that field, because null signals a wildcard field. If you have a field in an entry that may be variously null or not, you can set the field to null in your entry. If you need to write templates that distinguish between set and unset values for that field, you can (for example) add a Boolean field that indicates whether the field is set and use a Boolean value for that field in templates.

6.1.3 Distributed Leasing

The purpose of the leasing interfaces is to simplify and unify a particular style of programming for distributed systems and applications. This style, in which a resource is offered by one object in a distributed system and used by a second object in that system, is based on a notion of granting a use to the resource for a certain period of time that is negotiated by the two objects when access to the resource is first requested and given. Such a grant is known as a *lease* and is meant to be similar to the notion of a lease used in everyday life. As in everyday life, the negotiation of a lease entails responsibilities and duties for both the grantor of the lease and the holder of the lease. Part of this specification is a detailing of these responsibilities and duties, as well as a discussion of when it is appropriate to use a lease in offering a distributed service.

There is no requirement that the leasing notions defined in this document be the only time-based mechanism used in software. Leases are a part of the programmer's arsenal, and other time-based techniques such as time-to-live, ping intervals, and keep-alives can be useful in particular situations. Leasing is not meant to replace these other techniques, but rather to enhance the set of tools available to the programmer of distributed systems.

Leasing and Distributed Systems

Distributed systems differ fundamentally from non-distributed systems in that there are situations in which different parts of a cooperating group are unable to communicate, either because one of the members of the group has crashed or because the connection between the members in the group has failed. This partial failure can happen at any time and can be intermittent or long-lasting.

The possibility of partial failure greatly complicates the construction of distributed systems in which components of the system that are not co-located provide resources or other services to each other. The programming model that is used most often in non-distributed computing, in which resources and services are granted until explicitly freed or given up, is open to failures caused by the inability to successfully make the explicit calls that cancel the use of the resource or system. Failure of this sort of system

can result in resources never being freed, in services being delivered long after the recipient of the service has forgotten that the service was requested, and in resource consumption that can grow without bounds.

To avoid these problems, we introduce the notion of a lease. Rather than granting services or resources until that grant has been explicitly cancelled by the party to which the grant was made, a leased resource or service grant is time based. When the time for the lease has expired, the service ends or the resource is freed. The time period for the lease is determined when the lease is first granted, using a request/response form of negotiation between the party wanting the lease and the lease grantor. Leases may be renewed or cancelled before they expire by the holder of the lease, but in the case of no action (or in the case of a network or participant failure), the lease simply expires. When a lease expires, both the holder of the lease and the grantor of the lease know that the service or resource has been reclaimed.

Although the notion of a lease was originally brought into the system as a way of dealing with partial failure, the technique is also useful for dealing with another problem faced by distributed systems. Distributed systems tend to be long-lived. In addition, since distributed systems are often providing resources that are shared by numerous clients in an uncoordinated fashion, such systems are much more difficult to shut down for maintenance purposes than systems that reside on a single machine.

As a consequence of this, distributed systems, especially those with persistent state, are prone to accumulations of outdated and unwanted information. The accumulation of such information, which can include objects stored for future use and subsequently forgotten, may be slow, but the trend is always upward. Over the (comparatively) long life of a distributed system, such unwanted information can grow without upper bound, taking up resources and compromising the performance of the overall system.

A standard way of dealing with these problems is to consider the cleanup of unused resources to be a system administration task. When such resources begin to get scarce, a human administrator is given the task of finding resources that are no longer needed and deleting them. This solution, however, is error prone (since the administrator is often required to judge the use of a resource with no actual evidence about whether or not the resource is being used) and tends to happen only when resource consumption has gotten out of hand.

When such resources are leased, however, this accumulation of out-of-date information does not occur, and resorting to manual cleanup methods is not needed. Information or resources that are leased remain in the system only as long as the lease for that information or resource is renewed. Thus information that is forgotten (through either program error, inadvertence, or system crash) will be deleted after some finite time. Note that this is not the same as garbage collection (although it is related in that it has to do with freeing up resources), since the information that is leased is not of the sort that would generally have any active reference to it. Rather, this is information that is stored for (possible) later retrieval but is no longer of any interest to the party that originally stored the information.

This model of persistence is one that requires renewed proof of interest to maintain the persistence. Information is kept (and resources used) only as long as someone claims that the information is of interest (a claim that is shown by the act of renewing the lease). The interval for which the resource may be consumed without a proof of interest can vary, and is subject to negotiation by the party storing the information (which has expectations for how long it will be interested in the information) and the party in which the information is stored (which has requirements on how long it is willing to store something without proof that some party is interested).

The notion of persistence of information is not one of storing the information on stable storage (although it encompasses that notion). Persistent information, in this case, includes any information that has a lifetime longer than the lifetime of the process in which the request for storage originates.

Leasing also allows a form of programming in which the entity that reserves the information or resource is not the same as the entity that makes use of the information or resource. In such a model, a resource can be reserved (leased) by an entity on the expectation that some other entity will use the resource over some period of time. Rather than having to check back to see if the resource is used (or freed), a leased version of such a reservation allows the entity granted the lease to forget about the resource. Whether used or not, the resource will be freed when the lease has expired.

Leasing such information storage introduces a programming paradigm that is an extension of the model used by most programmers today. The current model is essentially one of infinite leasing, with information being removed from persistent stores only by the active deletion of such information. Databases and filesystems are perhaps the best known exemplars of such stores--both hold any information placed in them until the information is explicitly deleted by some user or program.

Goals and Requirements

The requirements of this set of interfaces are:

- To provide a simple way of indicating time-based resource allocation or reservation
- To provide a uniform way of renewing and cancelling leases
- To show common patterns of use for interfaces using this set of interfaces

6.1.4 Distributed Events

The purpose of the distributed event interfaces specified is to allow an object in one Java™ virtual machine (JVM) to register interest in the occurrence of some event occurring in an object in some other JVM, perhaps running on a different physical machine, and to receive a notification when an event of that kind occurs.

Distributed Events and Notifications

Programs based on an object that is reacting to a change of state somewhere outside the object are common in a single address space. Such programs are often used

for interactive applications in which user actions are modeled as events to which other objects in the program react. Delivery of such *local events* can be assumed to be well ordered, very fast, predictable, and reliable. Further, the entity that is interested in the event can be assumed to always want to know about the event as soon as the event has occurred.

The same style of programming is useful in distributed systems, where the object reacting to an event is in a different JVM, perhaps on a different physical machine, from the one on which the event occurred. Just as in the single-JVM case, the logic of such programs is often reactive, with actions occurring in response to some change in state that has occurred elsewhere.

A distributed event system has a different set of characteristics and requirements than a single-address-space event system. Notifications of events from remote objects may arrive in different orders on different clients, or may not arrive at all. The time it takes for a notification to arrive may be long (in comparison to the time for computation at either the object that generated the notification or the object interested in the notification). There may be occasions in which the object wishing the event notification does not wish to have that notification as soon as possible, but only on some schedule determined by the recipient. There may even be times when the object that registered interest in the event is not the object to which a notification of the event should be sent.

Unlike the single-address-space notion of an event, a distributed event cannot be guaranteed to be delivered in a timely fashion. Because of the possibilities of network delays or failures, the notification of an event may be delayed indefinitely and even lost in the case of a distributed system.

Indeed, there are times in a distributed system when the object of a notification may actively desire that the notification be delayed. In systems that allow object activation (such as is allowed by Java Remote Method Invocation (RMI) in the Java 2 SDK, v1.2.2), an object might wish to be able to find out whether an event occurred but not want that notification to cause an activation of the object if it is otherwise quiescent. In such cases, the object receiving the event might wish the notification to be delayed until the object requests notification delivery, or until the object has been activated for some other reason.

Central to the notion of a distributed notification is the ability to place a third-party object between the object that generates the notification and the party that ultimately wishes to receive the notification. Such third parties, which can be strung together in arbitrary ways, allow ways of off-loading notifications from objects, implementing various delivery guarantees, storing of notifications until needed or desired by a recipient, and the filtering and rerouting of notifications. In a distributed system in which full applications are made up of components assembled to produce an overall application, the third party may be more than a filter or storage spot for a notification; in such systems it is possible that the third party is the final intended destination of the notification.

Goals and Requirements

The requirements of this set of interfaces are to:

- Specify an interface that can be used to send a notification of the occurrence of the event
- Specify the information that must be contained in such a notification

In addition, the fact that the interfaces are designed to be used by objects in different virtual machines, perhaps separated by a network, imposes other requirements, including:

- Allowing various degrees of assurance on delivery of a notification
- Support for different policies of scheduling notification
- Explicitly allowing the interposition of objects that will collect, hold, filter, and forward notifications

6.1.5 Transaction

Transactions are a fundamental tool for many kinds of computing. A transaction allows a set of operations to be grouped in such a way that they either all succeed or all fail; further, the operations in the set appear from outside the transaction to occur simultaneously. Transactional behaviors are especially important in distributed computing, where they provide a means for enforcing consistency over a set of operations on one or more remote participants. If all the participants are members of a transaction, one response to a remote failure is to abort the transaction, thereby ensuring that no partial results are written.

Traditional transaction systems often center around transaction processing monitors that ensure that the correct implementation of transactional semantics is provided by all of the participants in a transaction. Our approach to transactional semantics is somewhat different. Within our system we leave it to the individual objects that take part in a transaction to implement the transactional semantics in the way that is best for that kind of object. What the system primarily provides is the coordination mechanism that those objects can use to communicate the information necessary for the set of objects to agree on the transaction. The goal of this system is to provide the *minimal* set of protocols and interfaces that *allow* objects to implement transaction semantics rather than the *maximal* set of interfaces, protocols, and policies that *ensure* the correctness of any possible transaction semantics. So the completion protocol is separate from the semantics of particular transactions.

The two-phase commit protocol defines the communication patterns that allow distributed objects and resources to wrap a set of operations in such a way that they appear to be a single operation. The protocol requires a manager that will enable consistent resolution of the operations by a guarantee that all participants will eventually know whether they should commit the operations (roll forward) or abort them (roll backward). A participant can be any object that supports the participant contract by implementing the appropriate interface. Participants are not limited to databases or other persistent storage services.

Clients and servers will also need to depend on specific transaction semantics. The default transaction semantics for participants is also defined in this document.

The two-phase commit protocol presented here, while common in many traditional transaction systems, has the potential to be used in more than just traditional transaction processing applications. Since the semantics of the individual operations and the mechanisms that are used to ensure various properties of the meta-operation joined by the protocol are left up to the individual objects, variations of the usual properties required by transaction processing systems are possible using this protocol, as long as those variances can be resolved by this protocol. A group of objects could use the protocol, for example, as part of a process allowing synchronization of data that have been allowed to drift for efficiency reasons. While this use is not generally considered to be a classical use of transactions, the protocol defined here could be used for this purpose. Some variations will not be possible under these protocols, requiring subinterfaces and subclasses of the ones provided or entirely new interfaces and classes.

Because of the possibility of application to situations that are beyond the usual use of transactions, calling the two-phase commit protocol a transaction mechanism is somewhat misleading. However, since the most common use of such a protocol is in a transactional setting, and because we do define a particular set of default transaction semantics, we will follow the usual naming conventions used in such systems rather than attempting to invent a new, parallel vocabulary.

The classes and interfaces defined by this specification are in the packages net.jini.core.transaction and net.jini.core.transaction.server. In this document you will usually see these types used without a package prefix; as each type is defined, the package it is in is specified.

Model and Terms

A transaction is created and overseen by a *manager*. Each manager implements the interface TransactionManager. Each *transaction* is represented by a long identifier that is unique with respect to the transaction's manager.

Semantics are represented by *semantic* transaction objects, such as the ones that represent the default semantics for services. Even though the manager needs to know only how to complete transactions, clients and participants need to share a common view of the semantics of the transaction. Therefore clients typically create, pass, and operate on semantic objects that contain the transaction identifier instead of using the transaction's identifier directly, and transactable services typically accept parameters of a particular semantic type, such as the Transaction interface used for the default semantics.

As shown in the next figure, a *client* asks the manager to create a transaction, typically by using a semantic factory class such as TransactionFactory to create a semantic object. The semantic object created is then passed as a parameter when performing operations on a service. If the service is to accept this transaction and govern its operations thereby, it must *join* the transaction as a *participant*. Participants in a transaction must implement the TransactionParticipant interface. Particular operations associated with a given transaction are said to be *performed under* that transaction. The client that created the transaction might or might not be a participant in the transaction.

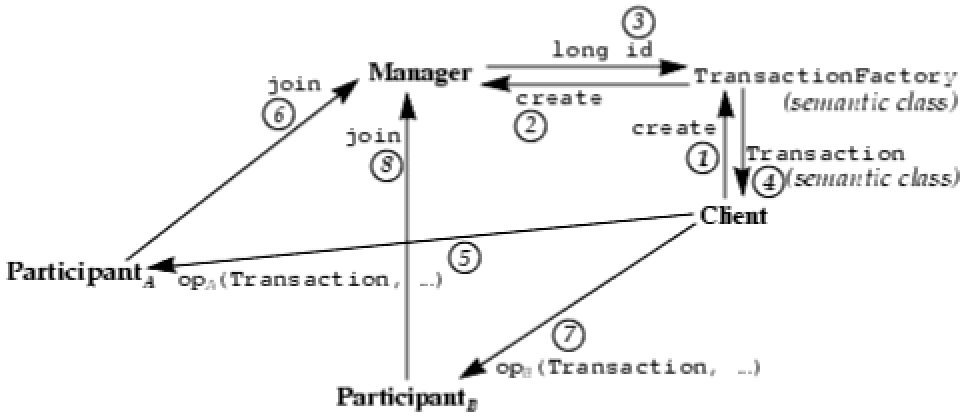


Figure 9: Transaction Creation and Use

A transaction *completes* when any entity either *commits* or *aborts* the transaction. If a transaction commits successfully, then all operations performed under that transaction will complete. Aborting a transaction means that all operations performed under that transaction will appear never to have happened.

Committing a transaction requires each participant to *vote*, where a vote is either *prepared* (ready to commit), *not changed* (read-only), or *aborted* (the transaction should be aborted). If all participants vote "prepared" or "not changed," the transaction manager will tell each "prepared" participant to *roll forward*, thus committing the changes. Participants that voted "not changed" need do nothing more. If the transaction is ever aborted, the participants are told to *roll back* any changes made under the transaction.

6.1.6 Lookup Service

The Jini™ lookup service is a fundamental part of the federation infrastructure for a *djinn*, the group of devices, resources, and users that are joined by the Jini technology infrastructure. The *lookup service* provides a central registry of services available within the djinn. This lookup service is a primary means for programs to find services within the djinn, and is the foundation for providing user interfaces through which users and administrators can discover and interact with services in the djinn.

Although the primary purpose of this specification is to define the interface to the djinn's central service registry, the interfaces defined here can readily be used in other service registries.

The Lookup Service Model

The lookup service maintains a flat collection of *service items*. Each service item represents an instance of a service available within the djinn. The item contains the RMI stub (if the service is implemented as a remote object) or other object (if the service makes use of a local proxy) that programs use to access the service, and an extensible collection of attributes that describe the service or provide secondary interfaces to the service.

When a new service is created (for example, when a new device is added to the djinn), the service registers itself with the djinn's lookup service, providing an initial collection of attributes. For example, a printer might include attributes indicating speed (in pages per minute), resolution (in dots per inch), and whether duplex printing is supported. Among the attributes might be an indicator that the service is new and needs to be configured.

An administrator uses the event mechanism of the lookup service to receive notifications as new services are registered. To configure the service, the administrator might look for an attribute that provides an applet for this purpose. The administrator might also use an applet to add new attributes, such as the physical location of the service and a common name for it; the service would receive these attribute change requests from the applet and respond by making the changes at the lookup service.

Programs (including other services) that need a particular type of service can use the lookup service to find an instance. A match can be made based on the specific data types for the Java™ programming language implemented by the service as well as the specific attributes attached to the service. For example, a program that needs to make use of transactions might look for a service that supports the type net.jini.core.transaction.server.TransactionManager and might further qualify the match by desired location.

Although the collection of service items is flat, a wide variety of hierarchical views can be imposed on the collection by aggregating items according to service types and attributes. The lookup service provides a set of methods to enable incremental exploration of the collection, and a variety of user interfaces can be built by using these methods, allowing users and administrators to browse. Once an appropriate service is found, the user might interact with the service by loading a user interface applet, attached as another attribute on the item.

If a service encounters some problem that needs administrative attention, such as a printer running out of toner, the service can add an attribute that indicates what the problem is. Administrators again use the event mechanism to receive notification of such problems.

6.2 Jabber vs. Proxy Communication

For the implementation of the message communication we took into consideration 2 approaches: Jabber framework and Proxy communication.

6.2.1 Jabber

Jabber is a set of streaming XML protocols and technologies that enable any two entities on the Internet to exchange messages, presence, and other structured information in close to real time. The first Jabber application is an instant messaging (IM) network that offers functionality similar to legacy IM services such as AIM, ICQ, MSN, and

Yahoo. However, Jabber is more than just IM, and Jabber technologies offer several key advantages:

- Open -- the Jabber protocols are free, open, public, and easily understandable; in addition, multiple implementations exist for clients, servers, components, and code libraries.
- Standard -- the Internet Engineering Task Force (IETF) has formalized the core XML streaming protocols as an approved instant messaging and presence technology under the name of XMPP, and the XMPP specifications are moving forward rapidly within the IETF's standards process.
- Proven -- the first Jabber technologies were developed by Jeremie Miller in 1998 and are now quite stable; hundreds of developers are working on Jabber technologies, there are tens of thousands of Jabber servers running on the Internet today, and millions of people use Jabber for IM.
- Decentralized -- the architecture of the Jabber network is similar to email; as a result, anyone can run their own Jabber server, enabling individuals and organizations to take control of their IM experience.
- Secure -- any Jabber server may be isolated from the public Jabber network (e.g., on a company intranet), and robust security using SASL and TLS has been built into the core XMPP specifications.
- Extensible -- using the power of XML namespaces, anyone can build custom functionality on top of the core protocols; to maintain interoperability, common extensions are managed by the Jabber Software Foundation.
- Flexible -- Jabber applications beyond IM include network management, content syndication, collaboration tools, file sharing, gaming, and remote systems monitoring.
- Diverse -- a wide range of companies and open-source projects use the Jabber protocols to build and deploy real-time applications and services; you will never get "locked in" when you use Jabber technologies.

Jabber uses a client-server communication. We prefer OpenIm, on open source implementation for the server because of its key features, useful for our algorithms: stability, modularity - component oriented using Apache Avalon Merlin manager (integration with LDAP or DB can be easily done via users-manager and storage API), most of classical IM functions are supported: message, presence, roster, subscription, vCard, offline storage, oob (lan file transfer), browse and search, server to server communications, secure connections via SSL, message logger and recorder (for statistic usage or supervision).

6.2.2 Proxy Service

This service intermediates communication between MonALISA Service and its clients. It registers as a Jini client being, in this way, found by clients. It also finds farms in given lookup services and connects with them. Clients send request messages to the known proxy, which forwards them to the specified farm.

This service was introduced because of the following reasons:

- it limits the number of TCP connections to farms. Without this proxy, every client starts its TCP connection with every found farm. With a big number of clients, a farm could be overloaded. But having a number of proxy services, the number of farm clients is much greater
- the number of messages between farms and clients decreases. For example, without this proxy, every client received from every farm the same filter messages, but on its TCP connection. Using the proxy service, this kind of messages are transmitted only between the farms and the proxy service and then spread by it to all known clients interested in those filters.
- the MonALISA service can now run behind a firewall without any problem. If the proxy cannot connect to the found farm, then the farm initiates the TCP connection with the proxy announcing its presence.

These proxy services run on different machines and register with known lookup services. The client finds these services and, getting the proxys attributes, makes a decision on which to choose. After choosing one, the communication with farms is intermediated by this one.

If the connection with the chosen proxy has died, the client tries to find another one and initiates a new dialog with farms through the new one.

6.2.3 Solution

Although Jabber offers many enhancements and it already has the infrastructure for message communication we chose to use the proxies for the communication in order to use the already existing TCP connections between the proxy and the farms. Each Jabber server would connect through TCP to each farm and thus the number of connections to the farms for a wide large netowork would increase dramatically.

We also studied the possibility of installing a Jabber Server on each Proxy but this would end up in one connection to the proxy, so the clients would be hidden and not directly accesible to the jabber server as the protocols ensures. In this case we should have changed the source code of OpenIm in order to make the clients transparent to the Jabber Server using just one connection to the Proxy; this would imply a greater number of messages in the network, suplimentary traffic over the network, thus not a reliable solution.

Instead we use the proxy for routing the message. To achieve this we had to modify the classes assuring the communication protocol between farms and proxies in MonALISA and introduce our own type of message: *monMessageMulticast*, which is an implementation of the *monMessage* interface.

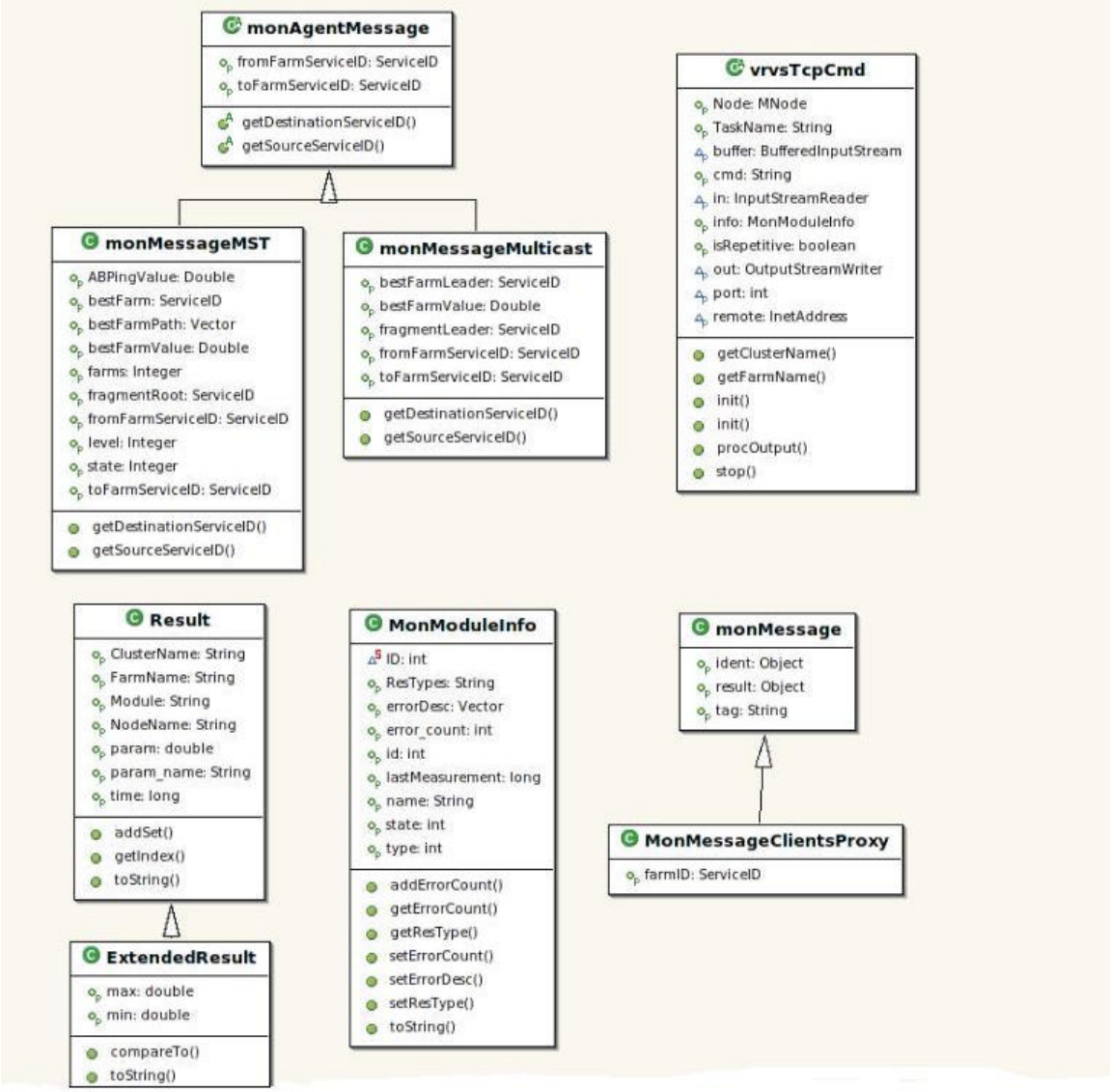
6.3 Implementation Aspects

The implementation of the presented algorithms was focused on two main directions: communication architecture and the development of the K-SPH algorithm.

6.3.1 Message Communication Arhitecture

During this stage we were concerned with the integration of the message communication of the algorithm in the existing MonALISA framework, namely to make possible the communication between farms, as in the existing stage the communication is possible only between farms and proxy.

To achieve this we developed specific types of messages (monMessageMulticast) derived from the existent interfaces and enriched the proxy implementation with new message listeners in order to route corectly the new types of messages.



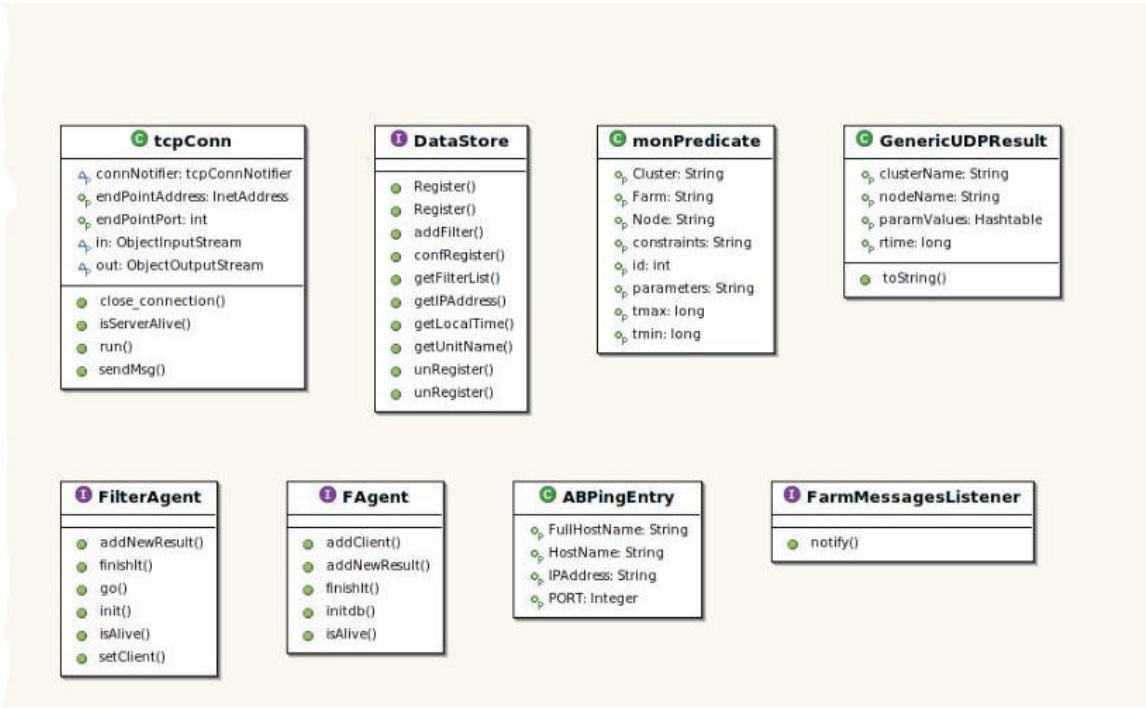


Figure 10: Components of `lia.Monitor.monitor` package which needed updates.

We also added a pool of messages in the DataCache, which are then routed using the appropiate listeners.

6.3.2 Multicast Algorithm Implementation

The development of the K-SPH algorithm in Java follows the steps presented in section 4: discovery and connection. First we need a synchronizing step in order to wait for all the farms to be online and start the agorithm.

We use two major listeners:

- for the messages inside a fragment
- for the messages outside a fragemnt (interfragment communication)

and take the appropiate decisions considering the type of message arrived as mentioned in the algorithm description.

The algorithm ends when no best farm to connect to is found, thus we have only one fragment which is the multicast tree. The farms are anounced by an appropiate message about the algorithm's termination.

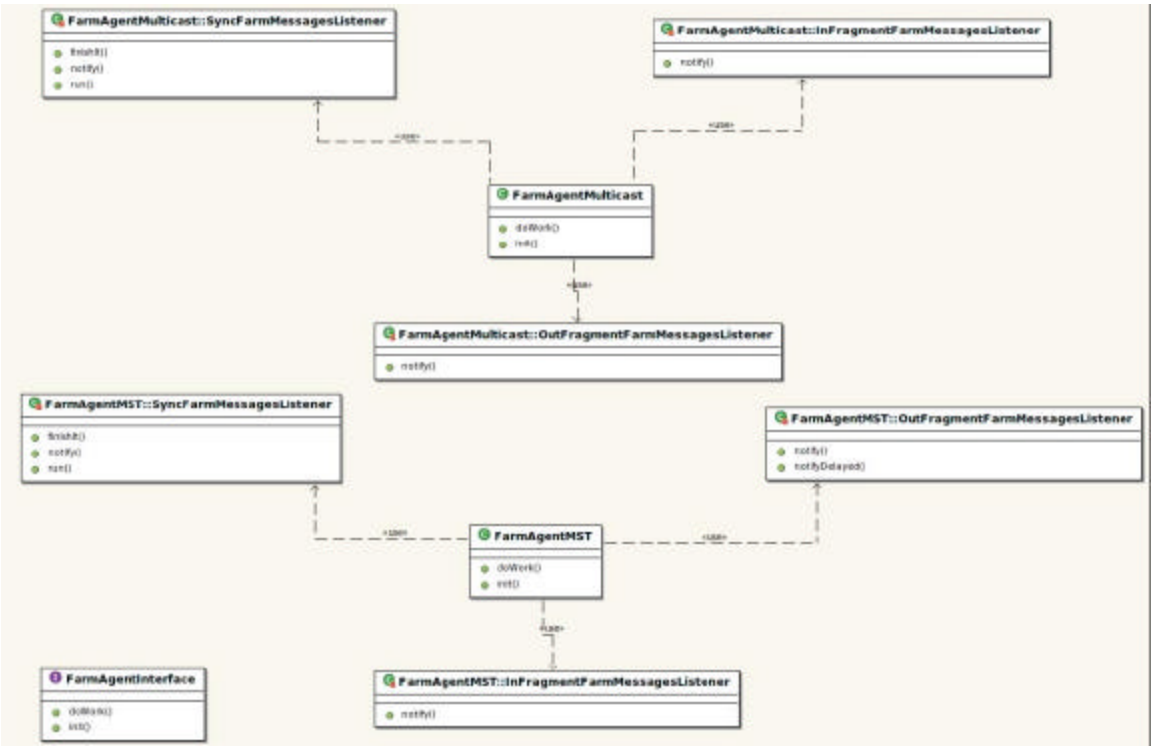


Figure 11: Comparative dependecies representations of the multicast and D MST implementations.

7. Conclusions

We compare our algorithms by execution against the baseline algorithm, the pruned minimum spanning-tree heuristic, which is the basis of many algorithms for finding multicast trees. We analyze the competitiveness (the ratio of the sum of the heuristic tree's edge weights to that of the best solution found) of the heuristic algorithms upon the pruned MST.

To evaluate the distributed heuristics presented in the Section 4.1, we implemented the algorithms and performed extensive executions on different generated test networks used by MonALISA. We choose the distributed MST algorithm as our baseline algorithm to compare the results. This algorithm was used to produce a minimum spanning tree of the network graph, which was then pruned to obtain a Steiner tree. We chose this MST algorithm as our baseline algorithm because the majority of previous distributed algorithms reviewed find multicast trees are based on finding minimal spanning trees. This algorithm differs from the heuristic algorithm, distributed K-SPH , in the fact that all the network nodes must participate in the execution of the

algorithm in the former, while only the multicast members and nodes in the vicinity of the multicast tree being set up execute the algorithm in the latter.

This section summarizes the execution results and compares the algorithms in terms of their convergence time, competitiveness, and the number of messages exchanged.

7.1 Evaluation Methodology

Each algorithm was run on a total of test networks. The test networks have 10% or 30% of its nodes in the multicast group because multicast applications running on such a WAN are likely to involve only a minority of nodes in the network.

We performed different executions on each generated graph by varying the multicast group size in diffenret ways.The nodes in a multicast group were chosen randomly in each case. The random numbers were chosen from a uniform distribution.

7.2 Metrics Tested

The metrics we use for comparison are the competitiveness, convergence time, and messages passed. Competitiveness is the ratio of heuristic tree cost C_t to that of the best solution C_{best} found by any heuristic. To determine the best solution, we considered solutions produced by the distributed heuristics described in this diploma paper as well as serial heuristics described in other articles. We use the best heuristic solution found for each test network rather than an optimal solution because explicit algorithms to find optimal solutions are prohibitively expensive on large networks.

The convergence time was found by measuring the elapsed time in the simulated network from the start of simulation to the time at which the last message reaches its destination. Since message-passing delays are likely to dominate over processing delays on the convergence time of the algorithm in a wide-area network, we considered only the former in computing the simulation time.

We used the distance between two nodes as the delay to pass a message between them. Messages passed is the total number of messages passed between nodes before convergence.

7.3 Simulation Results

Having described the algorithms and the simulation environment, we now turn to the results of our simulations.

The distributed versions of K-SPH provide inferior solutions compared to their centralized versions because of the lack of global topology information in each node in the former. However, the degradation in the competitiveness was small in our test networks. In fact, the competitiveness produced by distributed K-SPH was often superior to that of centralized SPH.

When comparing the competitiveness, heuristics K-SPH consistently outperformed the pruned MST heuristic, in both centralized and distributed cases. This result is consistent with the known theoretical upper bounds on the heuristics. It has been shown that cost of a solution produced by K-SPH is within twice the cost of an optimal solution. In contrast, the ratio between a solution produced by pruning a minimum spanning tree and an optimal solution can be as large as the number of non-multicast nodes. In our case, the cost of pruned MST solutions was rarely worse than twice that of the best solution found, but was often significantly worse than that produced by shortest path heuristics.

We found that 90% of the solutions produced by distributed K-SPH were within 4% of the best in terms of their cost. In comparison, when the best 90% of the solutions produced by the pruned MST algorithm were considered, some of the solutions had costs as high as 50% more than that of the optimal algorithm. Thus, if competitiveness is the most important criterion in the choice of the algorithm, distributed K-SPH is the heuristic of choice.

Heuristic K-SPH also enjoys the advantage that it doesn't require the participation of all the nodes in the network. Only the nodes in the multicast tree and within its neighborhood need to participate in the execution of the algorithm. The pruned minimum spanning tree algorithm, on the other hand, requires participation from every node of the network, a condition difficult to satisfy in practice in a large wide-area network. However, limiting the execution of the algorithm to a subset of the network nodes results in a substantial increase in the number of messages generated in our algorithms, with a corresponding increase in convergence time. This convergence time may be reduced by streamlining our algorithms.

Viewed from the perspective of convergence-time, however, the pruned MST heuristic enjoys an advantage over shortest path heuristics K-SPH. The convergence time for the solutions produced by pruned MST algorithm fell well within a much narrower range as compared to the results for distributed K-SPH.

Distributed K-SPH allows multiple fragments of the tree to combine in parallel. This allows distributed K-SPH to provide lower convergence times without increasing the number of messages substantially. Even though the convergence times for distributed K-SPH are higher than those of the pruned MST algorithm by as much as 10 times, we believe that the former can be brought down by careful optimization of distributed K-SPH.

7.4 Concluding Remarks

In this paper we introduced the distributed heuristics based on shortest path Steiner hueristics, and evaluated their performance relative to to a baseline pruned minimum spanning-tree heuristic. The primary advantage of our distributed algorithms over previous algorithms is that they require participation from only the nodes in the multicast tree and within their neighborhood.

The heuristics developed are an improvement over existing distributed Steiner heuristics based on the minimum spanning tree for two reasons: they produce solutions of superior quality in most cases and requires the participation of only a subset of network nodes. Our results show that the competitiveness of the solutions produced by our algorithms were, on the average, at least 25 percent better in comparison to those produced by the pruned spanning-tree approach. In addition, the competitiveness found by our algorithms in almost all cases was within 10% of the best solution found by any of the Steiner heuristics considered, including both centralized or distributed algorithms.

8. Future Work

We are working on optimizig the current implementation of the algorithm, especially trying to accelerate the convergence time and reduce the number of message exchanged. We also continue to add a monitoring tool for the multicast and a specifing logging interface in order to make the results more visible and reusable.

The goal of the project is to integrate the multicast and MST agents in a more generic agent communication platform developed for the future versions of MonALISA.

Also, we are working on a distributed version of the presented genetic algorithm.

I would like to thank Mr. Valentin Cristea and Mr. Iosif Legrand who coordinated scientificaly this work and also Mihaela Toarta, Ramiro Voicu, and Catalin Carstoiu for their continous support along the development of this diploma project.

Bibliography

[1] A. Ballardie, Core Based Trees (CBT) Multicast Routing Architecture, RFC 2201, September 1997.

[2] K. Bharath-Kumar and J. M. Jaftie, "Routing to Multiple Destinations in Computer Networks:' LEEE Trans. Cornrnurn.,vol.COM-31, pp. 343-351, 1983.

[3] R. Cohen, "Smooth Intentional Rerouting and its Applications in ATM Networks," IEEE INFOCOM, 1994, pp. 1490-1497.

[4] T. H. Cormen, C. E. Leiserson and R. L. Rivest, Introduction to Algoritlvns, MIT Press, 1990.

[5] Y. K. Dafat and R. M. Metcatfe, "Reverse Path Forwarding of Broadcast Packets," Coorrnurrications of the ACM, vol. 21, no. 12, pp. 1040-1048, 1978.

[6] S. Deering, et. al., Protocol Independent Multicast Version 2, Dense Mode Specification, work in progress.

[7] J. M. S. Dear, "Multicasting in The Asynchronous Transfer Mode Environment," Computer Laboratory Technical Report, no. 298, University Cambridge, 1993.

[8] D. Estrin, et. at., Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification, RFC 2117, June 1997.

[9] M. IMASE and B. M. Waxmarr, "Dynamic Steiner Tree Problem," SIAM

[10] R. M. Karp, "Reducibility among Combinatoriat Problems," in Miller and Thatcher (Eds.), Complexity of Computer Computations, Plenum Press, New York, 1972, pp. 85-103.

[11] L. Kou, G. Markowsky, and L. Berman, "A Fast Algorithm for Steiner Trees," Acts Informarica, vol. 15, pp. 141-145, 1981.

[12] H. Mehlhorn, "A Fsster Approximation Algorithm for the Steiner Problem in Graphs," Infom. Process. Lett., vol. 27, no. 3, pp. 125–128, 1988.

[13] J. Plesnik, "A Bound for the Steiner Tree Problem in Graphs," Math. Slovaca, vol. 31, no. 2, pp. 155–163, 1981.

[14] R. C. Prim, "Shortest Connection Networks and Some Genertilzations," Bell Syst. Tech. J. 36, pp. 1389-1401, 1957.

[15] T. Pusateri, Distance Vector Multicast Routing Protocol, work in progress.

[16] V. J, Rayward-Smith, "The Computation of Nearly Minimal Steiner Trees in Graphs," Int. J Math. Ed. Sci. Tech., vol. 14, no. 1, pp. 15–23, 1983.

[17] V. J, Rayward-Smith and A. Clare, "On Finding Steiner Vertices," Networks, vol. 16, pp. 283–294, 1986.

[18] H. Takahashi and A. Matsuyama, "An Approximate Solution for the Steiner Problem in Graphs," Math. Japonic, vol. 24, no. 6, pp. 573-577, 1980.

[19] H. Tode, Y. Sakai, M. Yamamoto, H. Okada and Y. Tezuka, "MulticastRouting Algorithm for Nodat Load Balancing," IEEE INFOCOM, 1992, pp. 2086-2095.
[20] M. Waxman, "Routing of Mukipoint Connections," IEEE Journal on Selected Area in Communications, vol. 6, no. 9, pp. 1617–1622, December 1988
[21] M. Waxman, "Performance Evahration of Multipoint Routing Algorithms," IEEE INFOCOM, 1993, pp. 980–986.
[22] P. Winter, "Steiner Problem in Networks: A Survey," Networks, vol. 17, pp. 129-167, 1987.
[23] J. Disc. Math, vol. 4, no. 3, pp. 369–384, August, 1991.
[24] E. N. Gilbert, "Random Graphs," The Annals of Mathematical Statistics, vol. 30, pp. 1141–1444, 1959.
[25] J. Kadtie, "Minimizing Packet Copies in Multicast Routing by Exploiting Geographic Spread," ACM SIGCOMM Computer Communication Review, vol 24, pp. 47–63, 1994.
[26] J. Kadirire and G. Knight, "Comparison of Dynamic Multicast Routing Algorithms for Wide-Area Packet Switched (Asynchronous Transfer Mode) Networks" IEEE INFOCOM, 1995, pp. 212-219.
[27] R. M. Karp, "Reducibility among Combinatoriat Problems," in Miller and Thatcher (Eds.), Complexity of Computer Computations, Plenum Press, New York, 1972, pp. 85-103.
[28] L. Kou, G. Markowsky, and L. Berman, "A Fast Algorithm for Steiner Trees," Acts Informarica, vol. 15, pp. 141-145, 1981.
[29] H. Mehlhorn, "A Faster Approximation Algorithm for the Steiner Problem in Graphs," Infom. Process. Lett., vol. 27, no. 3, pp. 125–128, 1988.
[30] J. Plesnik, "A Bound for the Steiner Tree Problem in Graphs," Math. Slovaca, vol. 31, no. 2, pp. 155–163, 1981.
[31] R. C. Prim, "Shortest Connection Networks and Some Genertilzations," Bell Syst. Tech. J. 36, pp. 1389-1401, 1957.
[32] T. Pusateri, Distance Vector Multicast Routing Protocol, work in progress.
[33] V. J, Rayward-Smith, "The Computation of Nearly Minimal Steiner Trees in Graphs," Int. J Math. Ed. Sci. Tech., vol. 14, no. 1, pp. 15–23, 1983.
[34] V. J, Rayward-Smith and A. Clare, "On Finding Steiner Vertices," Networks, vol. 16, pp. 283–294, 1986.
[35] H. Takahashi and A. Matsuyama, "An Approximate Solution for the Steiner Problem in Graphs," Math. Japonic, vol. 24, no. 6, pp. 573-577, 1980.
[36] H. Tode, Y. Sakai, M. Yamamoto, H. Okada and Y. Tezuka, "Multicast Routing Algorithm for Nodat Load Balancing," IEEE INFOCOM, 1992, pp. 2086-2095.
[37] B. M. Waxman, "Routing of Mukipoint Connections," IEEE Journal on Selected Area in Communications, vol. 6, no. 9, pp. 1617–1622, December 1988
[38] B. M. Waxman, "Performance Evahration of Multipoint Routing Algorithms," IEEE INFOCOM, 1993, pp. 980–986.

[39] P. Winter, “Steiner Problem in Networks: A Survey,” Networks, vol. 17, pp. 129-167, 1987.

[40] K. Bharath-Kumar and Jaffe. ``Routing to multiple destinations in computer networks," IEEE Transactions on Communications, vol. COM-31, no. 3, pp. 343--351, Mar. 1983.

[41] G. Chen, M. Houle, and M. Kuo. ``The Steiner problem in distributed computing systems," Information Sciences, vol. 74, no. 1-2, pp. 73--96, Oct. 1993.

[42] R. Gallager, P. Humblet, and P. Spira. ``A distributed algorithm for minimum-weight spanning trees," ACM Transactions on Programming Languages and Systems, vol. 5, no. 1, pp. 66--77, Jan. 1983.

[43] P. Humblet. ``Another adaptive distributed shortest path algorithm," IEEE/ACM Transactions on Communications, vol. 39, no. 6, pp. 995--1003, Jun. 1991.

[44] F. Hwang and D. Richards. ``Steiner tree problems," Networks, vol. 22, pp. 55--89, 1992.

[45] V. Kompella, J. Pasquale, and G. Polyzos. ``Two distributed algorithms for the constrained Steiner tree problem," in Proc. Comput. Commun. and Netw., San Diego, CA, Jun. 1993.

[46] M. Smith and P. Winter. ``Path-distance heuristics for the Steiner problem in undirected networks," Algorithmica, vol. 7, no. 2-3, pp. 309--327, 1992.

[47] H. Takahashi and A. Matsuyama. ``An approximate solution for the Steiner problem in graphs," Math. Japonica, vol. 24, no. 6, pp. 573--577, 1980.

[48] S. Voss. ``Steiner's problem in graphs: Heuristic methods," Discrete Applied Mathematics, vol. 40, pp. 45--72, 1992.

[49] P. Winter. ``Steiner problem in networks: A survey," Networks, vol. 17, no. 2, pp. 129--167, 1987.

[50] H.B. Newman, I.C. Legrand, J.J. Bunn, “A Distributed Agent-based Architecture for Dynamic Services” CHEP 2001, Beijing, Sept 2001,

[51] Julian Bunn and Harvey Newman Data Intensive Grids for High Energy Physics Grid Computing: Making the Global Infrastructure a Reality, edited by Fran Berman, Geoffrey Fox and Tony Hey, March 2003 by Wiley

[52] Nancy A. Lynch, Distributed Algorithms, Morgan Kauffman Publishers, 1996

[53] Michael T. Goodrich, Roberto Tamassia, Algorithm Design, John Wiley & Sons, 2001

[54] H.B. Newman, I.C. Legrand A Self-Organizing Neural Network for Job Scheduling in Distributed Systems CMS NOTE 2001/009, January 8, 2001

Appendix

Appendix 1: monAgentMessage.java

```
/*
 * Created on Jun 11, 2004
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
package lia.Monitor.monitor;

import net.jini.core.lookup.ServiceID;
/**
 * @author ceainegru
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
public abstract class monAgentMessage implements java.io.Serializable{
	public ServiceID fromFarmServiceID=null;
	public ServiceID toFarmServiceID=null;
	public abstract ServiceID getDestinationServiceID();
	public abstract ServiceID getSourceServiceID();
	public monAgentMessage(){
	}

}
```

Appendix 2: monMessageMulticast.java

```
/*
 * Created on Jun 16, 2004
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
package lia.Monitor.monitor;

import net.jini.core.lookup.ServiceID;
import lia.Monitor.monitor.monAgentMessage;



/**
 * @author ceainegru
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
public class monMessageMulticast extends monAgentMessage implements
java.io.Serializable{
            public ServiceID fromFarmServiceID;
            public ServiceID toFarmServiceID;
            public ServiceID fragmentLeader;
            public ServiceID bestFarmLeader;
            public Double bestFarmValue;



            //constructori
            public monMessageMulticast(){


            }


            public monMessageMulticast(ServiceID fromFarmServiceID, ServiceID
toFarmServiceID,ServiceID bestFarmLeader,Double bestFarmValue){
                        this.fromFarmServiceID = fromFarmServiceID;
```

```
                        this.toFarmServiceID = toFarmServiceID;
                        this.bestFarmLeader=bestFarmLeader;
                        this.bestFarmValue=bestFarmValue;
            }

            public monMessageMulticast(ServiceID fromFarmServiceID, ServiceID
toFarmServiceID){
                  this.fromFarmServiceID = fromFarmServiceID;
                  this.toFarmServiceID = toFarmServiceID;
            }

            public monMessageMulticast(ServiceID fromFarmServiceID,ServiceID
toFarmServiceID,ServiceID fragmentLeader){
                  this.fromFarmServiceID=fromFarmServiceID;
                  this.toFarmServiceID=toFarmServiceID;
                  this.fragmentLeader=fragmentLeader;
            }



            public ServiceID getDestinationServiceID(){
                  return toFarmServiceID;
            }

            public ServiceID getSourceServiceID(){
                  return fromFarmServiceID;
            }

            public ServiceID getFragmentLeader(){
                  return fragmentLeader;
            }

            public ServiceID getBestFarmLeader(){
                  return bestFarmLeader;
            }

            public double getBestFarmValue(){
                  return bestFarmValue.doubleValue();
            }}
```

Appendix 3: FarmAgentMulticast.java

```
/*
 * Created on Jun 17, 2004
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
package lia.Monitor.FarmAgents;



import java.util.Hashtable;
import java.util.Vector;
import net.jini.core.lookup.ServiceID;
import net.jini.discovery.LookupDiscoveryManager;
import java.util.logging.Logger;
import java.util.logging.Level;
import java.util.Comparator;
import java.util.Collections;

import lia.Monitor.DataCache.ProxyWorker;
import lia.Monitor.monitor.FarmMessagesListener;
import lia.Monitor.JiniSerFarmMon.RegFarmMonitor;
import lia.Monitor.JiniSerFarmMon.MLLUSHelper;
import lia.Monitor.monitor.monPredicate;
import lia.Monitor.DataCache.Cache;
import lia.Monitor.monitor.ExtendedResult;
import lia.Monitor.monitor.monMessage;
import lia.Monitor.monitor.monMessageMulticast;




/**
 * @author ceainegru
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
```

```
 */
public class FarmAgentMulticast {
	/** Logger Name */
	private static final transient String COMPONENT ="lia.Monitor.FarmAgents";
	/** The Logger */
	private static final transient Logger logger = Logger.getLogger(COMPONENT);



	private  LookupDiscoveryManager  ldm ;


	//obiect folosit pentru a rezolva IP sau numele host-ul unei ferme cu ServiceID-ul
corespunzator aceesteia
	private MLLUSHelper resolver;


	//obiect ce contine pool-ul de mesaje care ajung pe proxy si care realizeaza
interogarile si
	//actualizarile bazei de date folosind predicatele cu care se inscriu clientii
	protected Cache dataStore;


	protected RegFarmMonitor host ;


	//ID-ul fermei curente
	protected ServiceID hostServiceID ;


	//vectorul fermelor cu care suntem conectati
	protected  Vector farmsVector;


	//leaderul fragmentului
	protected ServiceID fragmentLeader;
	//parintele nodului curent in fragment
	protected ServiceID parent;


	//worker folosit pentru comunicatia cu proxy
	protected ProxyWorker proxyWorker;


	//starea algoritmului
	protected int state;
	private final int NOT_STARTED=-1;
	private final int CONNECTING=1;
	private final int DISCOVERY=2;
```

```
        private final int FINISHED=3;

        //vector de sincronizare
        protected Vector sync;

        //legaturile cu fermele care fac parte din arborele de multicast
        protected Vector MulticastFarmList;
        protected Hashtable farms;
        //id-ul fermei din alt fragment cu care incercam unirea
        protected ExtendedResult bestFarmRes;
        protected ServiceID bestFarm;
        //delay-ul gasit pentru aceasta ferma
        protected double bestFarmValue;

        //contor al mesajelor primite de la copii
        private int count;

        //listener pentru mesajele de sincronizare
        protected SyncFarmMessagesListener syncListener;
        //listener pentru mesajele exterioare fragmentului
        protected OutFragmentFarmMessagesListener outListener;
        //listener pentru mesajele interioare fragmentului
        protected InFragmentFarmMessagesListener inListener;

        //constructor
        public FarmAgentMulticast(){
                farmsVector = new Vector();
                MulticastFarmList = new Vector();
                sync=new Vector();
                state=NOT_STARTED;
                farms=new Hashtable();
        }

        //functie care realizeaza procesarie initiale
        public  void  init(LookupDiscoveryManager  ldm,  RegFarmMonitor  host,  ServiceID
hostServiceID){
                        this.ldm=ldm;
                        this.host=host;
                        this.hostServiceID=hostServiceID;
                        this.dataStore=(Cache) host.dataStore;
```

```
                        outListener=new OutFragmentFarmMessagesListener();
                        inListener=new InFragmentFarmMessagesListener();
                        syncListener= new SyncFarmMessagesListener();

                        //demareaza thread-ul pentru sincronizare
                        syncListener.start();

                        dataStore.setInFragmentListener(inListener);
                        dataStore.setOutFragmentListener(outListener);
                        dataStore.setSyncListener(syncListener);
                }

        //functie ce realizeaza trimiterea unui mesaj monMessageMulticast de tipul
        //indicat de tag folosind
        //functiile de comunicatie definite de proxyWorker, mesajul fiind impachetat
        //intr-un mesaj generic de tip monMessage (care contine ca si obiect in campul
        //tag.ident mesajul monMessageMulticast propriu-zis)
        private void sendMessage(monMessageMulticast Mmsg,String tag){
                logger.log(Level.INFO,"send message "+tag+" to
"+Mmsg.getDestinationServiceID());
                monMessage msg=new monMessage(tag,Mmsg,null);
                proxyWorker.getProxyTCPClientWorker().sendMessage(msg);

        }



        //thread ce asigura sincronizarea initiala a fermelor. Aceasta este necesara
deoarece fermele apar
        //la momente diferite in retea iar algoritmul trebuie sa isi inceapa executia in
momentul cand toate
        //cunosc existenta celorlalte
        private class SyncFarmMessagesListener extends Thread implements
FarmMessagesListener{

                        private boolean stop=false;

                        public synchronized void notify(monMessage msg){
                                monMessageMulticast Mmsg=(monMessageMulticast)msg.ident;
                                if(msg.tag.compareTo("SYNC")!=0)
                                        logger.log(Level.INFO,"received "+msg.tag+"
from "+Mmsg.getSourceServiceID()+" fragmentLeader="+fragmentLeader);
```

```
                                //la primirea unui mesaj de tip SYNC...
                                if(msg.tag.equals("SYNC")){
                                        if(state==NOT_STARTED){
                                                                //daca ferma e in starea initiala
mesajul e adaugat in pool-ul de mesaje de tip sync
                                                                dataStore.addSyncFarmMessage(msg);

        //logger.log(Level.INFO,"Synced delayed "+MSTmsg.getSourceServiceID());
                                        }
                                        else
                                                if(!isSynced(Mmsg.getSourceServiceID())){


                                                                //altfel se trimite reply
pentru a confirma sincronizarea cu ferma ce a trimis initial sync
                                                                //si aceasta este adaugata in
vectorul de ferme cu care s-a sincronizate deja

        sendSync(Mmsg.getSourceServiceID());

        sync.add(Mmsg.getSourceServiceID());
                                                                //
        logger.log(Level.INFO,"Synced with "+MSTmsg.getSourceServiceID());
                                                }
                                }
                        }


                    //threadul ruleaza pana se realizeaza sincronizarea cu toate fermele ->
stop
                        public void run(){
                                                logger.log(Level.INFO,"Synchronizing process
started!");
                                                while(!stop){
                                                        for(int
i=0;i<dataStore.getSyncFarmMessagesNumber();i++){

        notify(dataStore.getSyncFarmMessage());
                                                        }
                                                        try{
                                                                Thread.sleep(1000);
                                                        }
                                                        catch(Exception e){
                                                                e.printStackTrace();
                                                        }
                                                }
```

```
                        }

                        public void finishIt(){
                                stop=true;
                                logger.log(Level.INFO,"Synchronizing process
succedeed!");
                        }
        }

        //functie ce rezolva ID-ul unui ferme dintr-un Extend Result folosind MLUSHelper
        private ServiceID getFarmServiceID(ExtendedResult res){
                return resolver.getServiceIDfromFarm(res.NodeName);
                }

        //comparator pentru ID-urile fermelor, util in algoritm in momentul unirii a doua
fragmente:
        //decizia de unire si noul lider al fragmentului format sunt liderul cu indexul
mai mic
        private int compareServiceIDs(ServiceID sid1,ServiceID sid2){
                return sid1.toString().compareTo(sid2.toString());
        }

        //functie ce testeaza daca ferma este sincronizata cu ferma pasata ca argument
(daca se gaseste
        //in vectorul sync)
        protected boolean isSynced(ServiceID sid){
                for(int i=0;i<sync.size();i++)
                        if(compareServiceIDs(sid,(ServiceID)sync.elementAt(i))==0)
                                return true;
                return false;
        }

        //functie ce interogheaza baza de date pentru rezultate despre starea conexiunilor
        //folosind ABPing in predicatul monPredicate;
        protected void getData(){
                        String[] params=new String[1];
                        params[0]=new String("RTime");
                        //se citesc datele din ultimul minut, care se stocheaza in vectorul
results
                        Vector results=dataStore.select(new monPredicate("*","ABPing","*",-
60000,-1,params,null));
                //      logger.log(Level.INFO,"ABPing no. of results="+results.size());
```

```
                        for(int i=results.size()-1;i>=0;i--){
                                //numele fermei e rezolvat din rezultatul interogarii
(ExtendedResult) in ServiceID
                                //folosind MLUSHelper si rezultatul adugat in vectorul de
ferme
                                ExtendedResult res=(ExtendedResult)results.elementAt(i);
                                if(resolver.getServiceIDfromFarm(res.NodeName)!=null)
                                        if(!farms.containsKey(res.NodeName)){
                                                farms.put(res.NodeName,res);
                                                farmsVector.add(res);
                                        }
                        }


                        //odata informat asupra valorilor legaturilor adiacente se trece in
etapa de DISCOVERY
                        state=DISCOVERY;
                        //si incepe sincronizarea cu celelalte ferme
                        sync();


                        for(int i=0;i<farmsVector.size();i++)
                                //rezultatele primite de la ABPing sunt sortate pentru a se
putea alege mereu bestPath
                                Collections.sort(farmsVector,new Comparator(){
                                        public int compare(Object o1,Object o2){
                                                ExtendedResult res1=(ExtendedResult)o1;
                                                ExtendedResult res2=(ExtendedResult)o2;
                                                if(res1.param[0]==res2.param[0])
                                                        return
res1.NodeName.compareTo(res2.NodeName);
                                                if(res1.param[0]>res2.param[0])
                                                        return 1;
                                                return -1;
                                        }
                        });
                //      logger.log(Level.INFO,"After    sorting    the    data    no.    of
results="+unusedFarmList.size());
                //      for(int i=0;i<unusedFarmList.size();i++)
                //
        logger.log(Level.INFO,((ExtendedResult)unusedFarmList.elementAt(i)).NodeName+"="+(
(ExtendedResult)unusedFarmList.elementAt(i)).param[0]);
                }


        //in  update()  liderul  isi  instiinteaza  copii  la  unirea  cu  alt  fragment  despre
```

```
schimbarea liderului
        //si deci a indexului fragemntului
        private void update(){
                state=DISCOVERY;
                count=0;

                bestFarm=null;//cea mai buna ferma gasita
                bestFarmValue=Double.MAX_VALUE;//valoarea delay-ului pentru aceasta

                logger.log(Level.INFO,"Entering Discovering Step..."+MulticastFarmList);
                for(int i=0;i<MulticastFarmList.size();i++){
                        logger.log(Level.INFO,"UPDATE "+MulticastFarmList.elementAt(i));
                        logger.log(Level.INFO,"PARENT "+parent);
                        if(parent!=null){

        if(compareServiceIDs(parent,(ServiceID)MulticastFarmList.elementAt(i))!=0){
                                        monMessageMulticast                             up=new
monMessageMulticast(hostServiceID,(ServiceID)MulticastFarmList.elementAt(i),fragmentLeade
r);
                                        sendMessage(up,"UPDATE");
                                        count++;
                                }
                        }
                        else{
                                monMessageMulticast                                     up=new
monMessageMulticast(hostServiceID,(ServiceID)MulticastFarmList.elementAt(i),fragmentLeade
r);
                                sendMessage(up,"UPDATE");
                                count++;
                        }
                }
                discovery();
        }

        //in etapa de discovery fiecare nod cauta cea mai buna ferma din fragmentele
adiacente cu care sa se uneasca
        //folosind rezultatele furnizate de ABPing
        private void discovery(){

                        if(farmsVector.size()>0){
                                bestFarmRes=(ExtendedResult)farmsVector.remove(0);
```

```
        bestFarm=resolver.getServiceIDfromFarm(bestFarmRes.NodeName);
                                bestFarmValue=bestFarmRes.param[0];
                                logger.log(Level.INFO,"Best farm="+bestFarm);
                        }


                        best();
        }


        //se porneste agentul dupa ce se instantiaza rezolverul si proxyWorkerul
        public void doWork(){
                resolver=MLLUSHelper.getInstance();
                proxyWorker=dataStore.getProxyWorker();
                wakeup();
        }


        //se trimite parintelui cea mai buna ferma gasita
        private void best(){
                logger.log(Level.INFO,"Count="+count+" parent="+parent);
                if(count==0 && parent != null){
                        monMessageMulticast                                             mcast=new
monMessageMulticast(hostServiceID,parent,bestFarm,new Double(bestFarmValue));
                        sendMessage(mcast,"BEST");
                }
        }


        //la pronirea agentului se citesc datele si se intra direct in etapa de conectare
(se considera ca initial, prin citirea datelor
        //din baza de date s-a facut etapa de discovery)
        private void wakeup(){
                        fragmentLeader=hostServiceID;


                        getData();


                        logger.log(Level.INFO,"Starting MULTICAST...");



        bestFarm=getFarmServiceID((ExtendedResult)farmsVector.elementAt(0));
                        bestFarmValue=((ExtendedResult)farmsVector.elementAt(0)).param[0];
                        farmsVector.remove(0);


                        state=CONNECTING;
```

```
                        //algoritmul debuteaza cu trimiterea unui mesaj de tip connnect
catre un fragment preferat
                        monMessageMulticast                                          Mmsg=new
monMessageMulticast(hostServiceID,bestFarm,fragmentLeader);
                        sendMessage(Mmsg,"CONNECT");
                        logger.log(Level.INFO,"Entering Connection Step...");


                }


          //sincronizarea dureaza pana cand ferma ia cunostinta de toate celelalte ferme =
schimba mesaje
          //SYNC cu toate, deci pana cand vectorul sync ajunge la dimensiunea vectorului
de ferme
                private void sync(){
                        while(sync.size()!=farmsVector.size()){
                                for (int i=0;i<farmsVector.size();i++){
                                                ServiceID
crtFarmSID=getFarmServiceID((ExtendedResult)farmsVector.elementAt(i));
                                                if(!isSynced(crtFarmSID))
                                                        sendSync(crtFarmSID);
                                                try{
                                                                        Thread.sleep(500);
                                                }
                                                catch(Exception e){
                                                                        e.printStackTrace();
                                                }
                                }
                                try{
                                        Thread.sleep(1000);
                                }
                                catch(Exception e){
                                        e.printStackTrace();
                                }
                        }
                        syncListener.finishIt();
                }


        protected void sendSync(ServiceID sid){
                monMessageMulticast syncMmsg=new monMessageMulticast(hostServiceID,sid);
                sendMessage(syncMmsg,"SYNC");
```

```
	}


	//listenerul pentru mesajele externe fragmentului
	private class OutFragmentFarmMessagesListener  implements FarmMessagesListener{


		public synchronized void notify(monMessage msg){
			monMessageMulticast mcast=(monMessageMulticast)msg.ident;
			logger.log(Level.INFO,"received	message"+msg.tag+"	from "+mcast.getSourceServiceID()+" fragmentLeader="+fragmentLeader);


			if(msg.tag.compareTo("CONNECT")==0){
				//la primirea unui connect, conform algoritmului,daca nodul nu este leader trimite REJECT (doar liderul
				//ia decizii de unire)
				if(compareServiceIDs(hostServiceID,fragmentLeader)!=0){
					monMessageMulticast reject=new monMessageMulticast(hostServiceID,mcast.getSourceServiceID());
					sendMessage(reject,"REJECT");
				}
				else{
					//daca este lider dar nu este in starea de conectare raspunde cu BUSY
					if(state==DISCOVERY){
						monMessageMulticast busy=new monMessageMulticast(hostServiceID,mcast.getSourceServiceID());
						sendMessage(busy,"BUSY");
					}
					//daca cel ce a cerut connect nu este cel preferat de lider (in urma etapei de discovery) ii timite
					//de asemenea reject
					else{

	if(compareServiceIDs(bestFarm,mcast.getSourceServiceID())!=0){
							monMessageMulticast	reject=new monMessageMulticast(hostServiceID,mcast.getSourceServiceID());
							sendMessage(reject,"REJECT");
						}
						//altfel inseamna ca cei doi lideri se prefera reciproc deci se vor uni (ii confirma celuilalt printr-un accept)
						//si isi instiineaza copii despre aceasta (schimbarea frament liderului si implicit a indexului fragmentului)
						else{
							monMessageMulticast	accept=new
```

```
monMessageMulticast(hostServiceID,mcast.getSourceServiceID());
                                                    sendMessage(accept,"ACCEPT");
                                                    MulticastFarmList.add(bestFarm);
                                                    logger.log(Level.INFO,"Add         in
MulticastTree "+bestFarm);

       if(compareServiceIDs(hostServiceID,mcast.getSourceServiceID())<0)
                                                                update();
                                                }
                                        }
                                }
                        }
                        else
                        if(msg.tag.compareTo("REJECT")==0){
                                //la primirea unui reject nodul readauga ferma preferata in
lista de ferme pentru a o include in etapa
                                //urmatore de discovery in care intra din nou
                                farmsVector.add(bestFarmRes);
                                discovery();
                        }
                        else
                        if(msg.tag.compareTo("ACCEPT")==0){
                                //la primirea unui accept compara id-urile propriu si al
frgamentului preferat pentru a vedea care
                                //devine lider (cel cu id-ul mai mic)

       if(compareServiceIDs(hostServiceID,mcast.getSourceServiceID())>0){
                                        parent=mcast.getSourceServiceID();
                                }
                        }
                        else
                        //la primirea mesajului de busy continua etapa de conectare
trimitand un mesaj de connect urmatoare ferme
                        //preferae din vectorul sortat de preferinte
                        if(msg.tag.compareTo("BUSY")==0){
                                monMessageMulticast                             connect=new
monMessageMulticast(hostServiceID,mcast.getSourceServiceID());
                                sendMessage(connect,"CONNECT");
                        }


                }
```

```
        }

        //listener pentru mesajele din interiorul fragmentului
        private class InFragmentFarmMessagesListener  implements FarmMessagesListener{

                public synchronized void notify(monMessage msg){
                        monMessageMulticast mcast=(monMessageMulticast)msg.ident;
                        logger.log(Level.INFO,"received     message     "+msg.tag+"     from
="+mcast.getSourceServiceID()+" fragmentLeader ="+mcast.getFragmentLeader());

                        if(msg.tag.compareTo("UPDATE")==0){
                                //la prmirea unui mesaj de update se actualizeaza liderul
fragemntului si parintele curent
                                fragmentLeader=mcast.getFragmentLeader();
                                parent=mcast.getSourceServiceID();
                                update();
                        }
                        else
                        if(msg.tag.compareTo("BEST")==0){
                                count--;

                                //la   primirea   unei   ferme   preferae   pentru   unire   se
actualizeaza daca este cazul
                                //ferma actuala preferata
                                if(mcast.getBestFarmValue()<bestFarmValue){

                                        bestFarm=mcast.getBestFarmLeader();
                                        bestFarmValue=mcast.getBestFarmValue();
                                }

                                //daca s-au primit mesaje de la toti fii
                                if(count==0){
                                        //se reintra in etapa de conectare
                                        state=CONNECTING;
                                        //daca nu este radacina, trimite mesaj mai sus cu
ferma preferata
                                        if(parent!=null){
                                                monMessageMulticast                  best=new
monMessageMulticast(hostServiceID,parent,bestFarm,new Double(bestFarmValue));
                                                sendMessage(best,"BEST");
                                        }
```

```
                                        //altfel este radacina si ia decizia de unire
                                        else{
                                                //daca nu s-a mai gasit o cea mai buna ferma
inseamna ca algoritmul s-a incheiat si se anunta toate nodurile
                                                if(bestFarm==null){

        logger.log(Level.INFO,"MCast="+MulticastFarmList);
                                                        for(int
i=0;i<MulticastFarmList.size();i++){
                                                                monMessageMulticast
finish=new monMessageMulticast(hostServiceID,(ServiceID)MulticastFarmList.elementAt(i));
                                                                sendMessage(finish,"FINISH");
                                                        }


                                                }
                                                else{
                                                        //trimite connect fermei preferate in
urma etapei de discovery
                                                        monMessageMulticast       connect=new
monMessageMulticast(hostServiceID,bestFarm);
                                                        sendMessage(connect,"CONNECT");
                                                }
                                        }
                                }
                        }


                        else


                        //s-a primit finish, algorimtul s-a incheiat, mesajul este trimis
mai departe memebrilor arborelui de multicast
                        if(msg.tag.equals("FINISH")){
                                logger.log(Level.INFO,"MCAST="+MulticastFarmList);
                                for(int i=0;i<MulticastFarmList.size();i++){

        if(compareServiceIDs(parent,(ServiceID)MulticastFarmList.elementAt(i))!=0){
                                        monMessageMulticast               finish=new
monMessageMulticast(hostServiceID,(ServiceID)MulticastFarmList.elementAt(i));
                                        sendMessage(finish,"FINISH");
                                }
                                }


                        }
```

```
            }
        }


}
```