A Case Study on Event Dissemination in an Active Overlay Network Environment

Sérgio Duarte, J. Legatheaux Martins, Henrique J. Domingos, Nuno Preguiça
DI/FCT/UNL
Quinta da Torre, 2829-516 Caparica - Portugal
+351-212948300
{smd, jalm, hj, nmp}@di.fct.unl.pt

ABSTRACT
In this paper, we describe a case study of the design and development of a group-conferencing tool suite, built on top of an overlay network based event dissemination framework, which is extensible via quality of service template plug-ins. We explain, for each of the tools, how the framework built-in conveniences were explored to create simple but effective distributed solutions, backed by the appropriate quality of service templates, whose design we also discuss.

Keywords
Case study, event dissemination, quality of service (QoS), multicasting, overlay networks, active networks.

1. INTRODUCTION
Distributed application design is closely tied to the problem of the quality of service offered by the support communication channels. In general, for a given problem, a too weak quality of service tends to put an excessive burden on the application, which has to overcome the communication infrastructure shortfalls on its own. On the other hand, an excessive quality of service is wasteful because it normally comes with a matching price tag somewhere. Ideally, one should strive for a balanced compromise between the two, aiming at simpler applications backed by communication support with the "right" quality of service. This has been recognized in many fields of distributed computing and, naturally, also in the more specific context of messaging middleware and event systems [1][2][6].

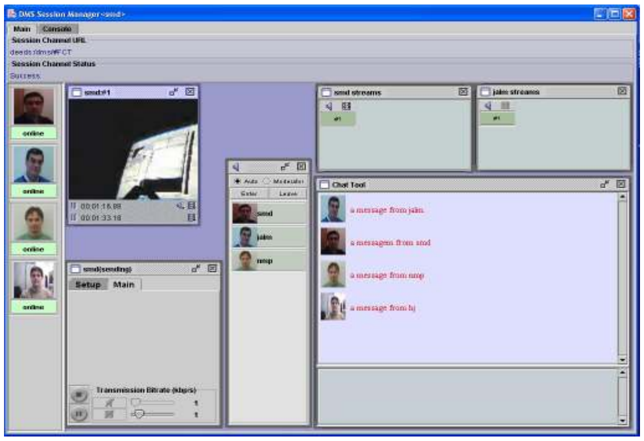
Our work in the context of distributed event dissemination tackles this precise challenge of designing a flexible, generic event dissemination framework, capable of providing the means to easily and incrementally build communication support channels with just the "right" quality service needed in each situation. We have addressed this problem by creating a solution based on pluggable QoS templates that leverages its overlay-network oriented architecture to achieve those goals. We want to show that this may prove to be a viable alternative to the "one size fits all" approach.

In this paper, we intend to describe the experience gained from the development of a group-conference tool suite built on top of a framework that advocates principles that go deliberately against rigid, "one size fits all" approaches in the context of distributed event dissemination.

2. CASE STUDY APPLICATION
The case study JAVA application is a barebones group-conference tool suite, comprising videoconference, moderator and chat tools. It allows a user to join a named group session, monitor the status of other users and engage in chat or videoconference activities. A moderator tool is included to help the audio coordination of videoconference sessions involving multiple participants.

The objective of this case study is to test the claim that an expected positive impact on application development supported by data dissemination with the "right" quality of service (QoS) is achievable and viable in an event dissemination framework extensible via specific QoS template plug-ins.

In broad terms, the application developed consists of a desktop where the individual tools are launched and manipulated. A sample screen capture is shown in Figure 1. The desktop provides an updated view of the status of the users enlisted in the current session. Videoconference activities, within a session, are achieved using complementary sender and a receiver tools and involve encoding, *multicasting* and presenting RTP [3] A/V streams. An optional moderator tool allows informal dialog coordination, by enabling and muting the appropriate audio streams, according to the evolving state of a global queue of enrolled participants. A chat tool makes up the last of the desktop components.



a message from jalm
a messagem from smd
a message from nmp
a message from hj

Figure 1 - Sample screen capture of an ongoing session.

To test the aforementioned claim, the entire communication requirements posed by this tool suite have been strictly fulfilled by the amenities of the event dissemination framework, by developing framework plug-ins with the appropriate QoS classes, as required by each application component. Therefore, we must highlight that this case study focuses on the problem of flexible event transportation and sidelines other key aspects of event dissemination such as filtering. In doing so, we intentionally stressed the event transport facet of the framework by evaluating its feasibility in dealing with a scenario with communication needs closer to the data multicasting problem.

3. DEVELOPMENT FRAMEWORK

The tool suite is built on top of a JAVA-based event dissemination platform named DEEDS. DEEDS has been designed to be as flexible and adaptable as possible and aimed at a broad range of applications and execution scenarios. The guiding goals of the framework are the extensibility and configurability of existing features, as a way of satisfying the requirements of large-scale, heterogeneity and mobility in specific contexts.

DEEDS advocates a general-purpose solution in the sense that it can be easily adapted to particular problems, or greatly eases the creation of custom solutions using existing features as guiding blueprints. A small set of simple and intuitive concepts have been deliberately used to foster an incremental approach towards problem solving that capitalizes on existing experience.

3.1 Event Dissemination Model

DEEDS implements the well-known *publish/subscribe* paradigm, enhanced with a *feedback* operation allowing event consumers and event sources to engage in one-to-one event exchange dialogs. These operations are supported over *active event channels* that designate named instances of particular QoS templates. These QoS templates correspond to system-level plug-ins that execute in the nodes of the event dissemination overlay network and provide the routing logic needed to direct the event stream produced by the *publish* and *feedback* operations.

The event dissemination model offered is also protocol transparent, meaning that there are no references to specific communication protocols at either the application level or within the QoS templates themselves. Actual protocol bindings are relegated to the deployment phase and subjected to the administrative policies of each particular site.

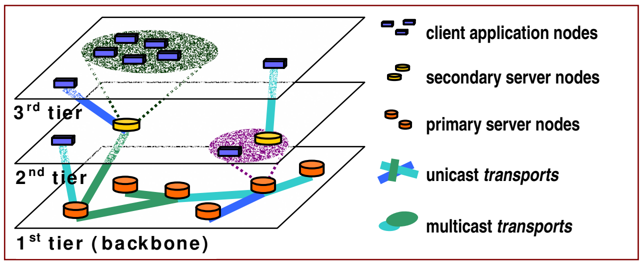


Figure 2 – DEEDS' Overlay network architecture, showing the tree node types linked by various kinds of *transports*.

3.2 Platform Architecture

The event dissemination model summarized above is matched by a distributed architecture designed with large-scale and heterogeneity support in mind. A three-tier overlay network of nodes makes up the core of the event dissemination infrastructure, as shown in Figure 2. The first tier of this logical network is known as the *backbone* and its server nodes typically handle the more demanding routing operations. The second level is made of a mix of secondary server nodes and client nodes (applications), while solely client nodes compose the third tier. In every node, regardless of type but, with different contexts, instances of the QoS template plug-ins are executed to handle event forwarding.

Routing of events and exchange of control messages between nodes is forwarded over *transports,* which are wrappers that abstract the actual communication links connecting the involved processes. Use of a heterogeneous mix of *transports* to form the overlay network is allowed, thus it is possible to use TCP, UDP, IP Multicast, HTTP or other protocol based *transport* at the same time to accommodate different administrative policies.

The dissemination architecture also comprises a data repository, known as the *system registry*, where static-persistent configuration and dynamically collected volatile data is kept in the form of attribute-value pairs. Portions of the registry have a local scope and deal, essentially, with information about each node particular configuration and runtime status. The remaining of the registry is globally reachable (on demand) and is replicated (lazily) among all server nodes. This is the place where DEEDS stores persistent information that is relevant to every node, such as the event channel directory that lists the names of known channels and their bindings to the QoS templates.

3.2.1 Node architecture

The primary job of a DEEDS node is to provide the execution environment for the event channel QoS template instances. Event routing within a QoS plug-in typically involves accepting incoming events and control messages, updating the state of the node, and sending event and control messages to other nodes over the appropriate *transports*. A node, depending on its type, also runs a number of background *services*. These services exchange information with their counterparts on other nodes to perform housekeeping functions and provide a monitored view of the status of the dissemination network. One of these services, for instance, is responsible for maintaining the node's *system registry* replica. An explanation of the most relevant services comes next.

3.2.1.1 Backbone Monitoring Services

These services are two intertwined, complementary processes that only run on the first tier, backbone nodes. Their purpose is to monitor the overlay network and assemble a structured view of the overlay network backbone.

One of the two is the *Hello* service, which continuously probes the list of currently known (backbone) nodes, one by one, to determine which are active and to obtain an estimate of their distance. A scheduler within the service assigns higher priorities to nearby or "critical" nodes, so that the allotted bandwidth is not wasted on probing irrelevant nodes that are too distant in terms of latency or spanning tree hops. The *Linkstate* service completes the pair; its task is to efficiently deliver the data gathered by the *Hello* service to the other backbone nodes and collect theirs, so that a global perception of state of the backbone is achieved. To attain this, each node periodically publishes its "hello data" in a dedicated special broadcast event channel. The data is encoded in such a way that, with a modest increase in size, also carries the node's current assessment of the "best" backbone spanning tree. Embedding a spanning tree in each of these messages allows the broadcasting to be achieved by *source routing* the message to next nodes in the tree path. This scheme is advantageous because no special coordination among the nodes is required to avoid cycles or to detect duplicates; it permits the *Linkstate* service to rely on itself to improve recursively its own routing information. As a result, the global view of the backbone these services provide makes it possible to obtain good spanning trees directly with graph theory algorithms. The minimum spanning tree (MST) algorithm is one of them but, although simple and lightweight, it tends to produce deep, meandering trees, which is not desirable. Instead, we prefer to use a spanning tree derived from a spanner graph algorithm, which adds shortcuts to the MST so that the distance between any two nodes in the spanner does not exceed by a given factor their direct distance. The depth of the resulting spanning trees can be finely controlled using the spanner factor, while keeping the tree cost effective.

The information received through this service is also used to gather knowledge about fresh backbone nodes. Finally, the spanning tree advertised by the node with the lowest identifier is taken as the official one and used to produce multicast and unicast routing tables that, in turn, can be employed to drive the event routing in other QoS templates plug-ins, such as the one used by the *system registry* management service summarized next.

3.2.1.2 Registry Management Service

This node service manages the global, replicated portion of the *system registry*. The service runs on every node but, since client nodes only keep a volatile cache of the *system registry*, the operation of the service in these nodes is somewhat restricted.

The service updates the registry in two different ways. There is a low bandwidth proactive replication process that periodically multicasts registry items in a dedicated event channel. But, more often, updates to the registry are the result of lookups that cannot be resolved locally and are sent to other nodes in the form of queries. Both processes rely on a tailored event channel QoS template to send and receive information. This event channel can both multicast registry items and queries away from a source or unicast replies towards a destination, one single hop at a time in both cases, querying and feeding system registries along the way.

3.3 Programming Model

DEEDS programming model is expressed in the JAVA programming language and assumes execution in a standard JAVA environment. The programming library consists of a set of user-level programming interfaces intended for the development of applications. And, a set of system-level classes for system enhancement, which allow the creation of additional node support *services*, novel QoS template plug-ins and *transport* classes.

A flexible concept of event is used, representing a reasonably small, self-contained notification, composed by a pair of items: a main payload, in the form of an arbitrary "serializable" JAVA object; and an *envelope* object, whose particular class may be specific to each event channel type (represented by its supporting QoS template). Both event components are optional, which means that empty events are allowed. Data overlap between the two is not restricted in any way but is wasteful and should be avoided.

The role of *envelope* objects can be seen as a way of passing arbitrary control information to the event dissemination infrastructure to avoid the need to scrutinize the main event payload for that same purpose at a greater cost. For instance, the envelope can be a rough description of the main event payload, to assist QoS templates in optimizing event dissemination based on aggressive event filtering practices. Or, more simply, an envelope can be an expiration deadline to allow the QoS template of the event channel to automatically discard late events before reaching some of its subscribers and, thus, free network resources earlier.

The counterpart of the *envelope* is the *criteria* object used in subscription operations. These are generic event filters operating over envelope types that are used to check the envelopes of incoming events to select those to be delivered to the application. Together, *envelopes* and *criteria* form the basis of the event filtering capabilities of the framework.

The event model also includes the notion of *receipt* objects, whose purpose is to aggregate and return system-generated information associated with an event, such as event-source identifiers, sequence numbers and subscription "handbacks". These *receipts* cannot be fabricated and are important for the *feedback* operation because they identify the event source targeted by the operation.

3.3.1 Application Programming Interfaces

The basis of the programming interfaces is the EventChannel class, which provides the access points to the event dissemination operations according to the *publish/subscribe/feedback* model. References to these objects are obtained by performing a *lookup* operation on a global event channel directory. The only parameter required is the string name of the desired event channel. Creation of a new event channel is accomplished with the *clone* operation, which takes the intended name for the new channel and the name of the QoS template plug-in, in which the new channel will be based upon. The use of "*clone*" for the operation name is meant as way of emphasizing the idea that the new event channel will be a copy or clone of a prototype channel already present and accepted into the system.

Having obtained a reference to an *EventChannel* object, the application can follow the expected programming pattern of the *publish/subscribe* paradigm. The specifics being that the *publish* operation requires an *envelope* and an object (the main payload*)* and returns a *receipt*. To be notified an application performs *subscribe* operations, specifying *criteria* objects to filter out undesired events based on their *envelopes*. The *feedback* operation fits in the model to allow a notified application to engage into a one-to-one dialog with a specific event source; it differs from the *publish* operation by requiring a *receipt* of a previously received event as an extra argument.

The following code excerpt exemplifies the use of these main programming interfaces in two basic *publisher* and *subscriber* applications. For clarity and brevity, only partial argument lists are shown.

```
import deeds.api.*;
public class Publisher implements EventFeedbackSubscriber {
    EventChannel c ;
    public Publisher() {
        Deeds.Directory().clone( "QoStemplate", "channel_name");
        c = Deeds.Directory().lookup("channel_name");
        c.subscribeFeedback( criteria, ..., this);
        while(...)  c.publish( envelope, payload );
        c.unsubscribe(...);
    }
    void nofifyFeedback( Receipt r, Envelope e, MarshalledEvent m ) {
        Object  payload = m.getEvent();
        ...
        c.feedback( r, envelope, payload2) ;
    }
}
```

```
import deeds.api.*;
public class Subscriber implementsEventSubscriber,EventFeedbackSubscriber{
    EventChannel c ;
    public Subscriber() {
        c = Deeds.Directory().lookup("channel_name");
        c.subscribe( criteria, ..., this) ;
        c.subscribeFeedback( criteria2, ..., this);
    }
    void nofify( Receipt r, Envelope e, MarshalledEvent m ) {
        Object  payload = m.getEvent() ;
        ...
        c.feedback( r, envelope, payload2) ;
    }
    void nofifyFeedback( Receipt r, Envelope e, MarshalledEvent m ) {
        Object  payload = m.getEvent();
        ...
        c.feedback( r, envelope, payload2) ;
    }
}
```

3.3.2 QoS Template Development

Extending the framework capabilities is in great part tied to the development of new QoS template plug-ins. In their essence, event channel templates implement a particular routing protocol across the overlay network to deliver events to interested parties. A QoS template must deal with two separate streams of events, the multi-point stream that is produced by *publish*-operations, and the (optional) unicast stream consisting of *feedback* events. To achieve this purpose, the plug-in can also format any appropriate control messages it needs and exchange them with other nodes.

Unless the desired QoS is very basic, design of a new plug-in can be a complex task. To make their development easier it is possible to capitalize on useful information already available in the node. This information is made accessible through the *system registry* and is presented in the form of *dynamic objects* that other processes keep updated and store in named *containers*. Containers keep track of changes in the information they store and notify interested parties. This scheme allows QoS plug-ins to synchronize their state (a privately computed routing table, for example) in reaction to changes in the *containers* they monitor. The framework already provides a number of these *containers* such as, a list of known backbone nodes and the *transports* available to reach them, a list of local subscribers for each event channel, a current view of the overlay network links, a low-cost spanning tree covering the backbone nodes and the associated broadcast and unicast routing tables. These resources are a great help in the programming of new plug-ins, as will be shown in the following sections, where we describe the ones that were developed for the purpose of the group conference tool suite. A source example is also provided in the appendix at the end of this paper.

4. CASE STUDY DEVELOPMENT

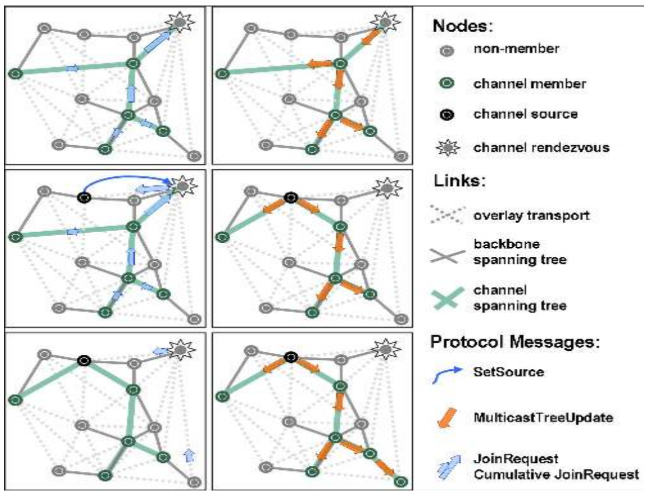
In this section we describe the most relevant aspects that guided the development of each of the applications that make up this case study. We recall that the challenge we have undertaken has been to show that a combination of the right quality of service in communication can lead to simple (peer-oriented) applications that address elaborate problems. Furthermore, we want to verify that, with an acceptable effort, the desired QoS is feasible within plug-in model of the DEEDS framework. In any way, we want to advocate here that this is the best way to solve these problems but that it is a good, promising way; a viable alternative to more popular approaches such as the centralized client-server model.

4.1.1 Video-Conference Tools

The videoconference tools are more precisely described as being two separate programs, the *transmitter* that captures, encodes and transmits the a/v streams and the *receiver* that decodes and presents them. For obvious reasons, we used the Java Media Framework [4] to create these programs. It allows a JAVA application to easily capture, encode or transcode audio and video streams in a number of standard formats. One feature of JMF that is particularly pertinent for this case study is its ability to deal with RTP encoded media streams. RTP [3] (and its companion protocol RTCP) is a IETF sponsored transport protocol, specifically designed for transmitting real-time data, such as audio, video over multicast or unicast network services. What makes RTP so attractive is that it has been made independent of the underlying transport and network layers, which enables us to encode RTP streams and multicast them over our event dissemination overlay network.

The core effort in delivering RTP streams over DEEDS rested in the creation of the appropriate *RTPConnectors* adaptors according to the JMF specifications, which are the actual objects used internally to have a media source send out the RTP and RTCP packets and gather reception statistics reports (RTCP packets) from its listeners. Implementing these connectors in DEEDS was no trouble at all, and merely consisted in having the connector *publish* the RTP and RTCP packet stream in a given event channel and use the *feedback* operation to report back the RTCP packets to the source.

The greater undertaking in the development of these tools was the selection of the best event channel type for the task and implementing the corresponding QoS template plug-in. Given the nature of the problem, the desired event channel type had to offer a light-weight multicast service with as low as possible latency and jitter. In this particular case, reliability is not an issue and dropping a few packets is tolerable. Moreover, a simpler single-source multicast routing protocol solution can be adopted provided each sender uses its own channel, which is actually desirable in this case. With these characteristics in mind, we implemented a *SingleSourceUnreliableMulticast* plug-in.



The plug-in implements its multicast routing protocol capitalizing heavily on network state data already provided by the normal operation of the framework. It essentially creates a tree of backbone nodes, see picture above, rooted at the node where the event source is connected and spanning the nodes with registered subscribers. A special rendezvous node selected independently for each channel, by mapping the channel id to a node id and finding the best match in the list of backbone nodes, acts as a temporary root. A node joins the multicast tree, in response to changes in its registrations *container*, by sending a *JoinRequest* control packet towards the root of the tree. These requests travel towards the root one hop at time (except the first time when they have to reach the root via the rendezvous node). Each node merges all the requests it receives from lower level nodes into a larger compound request. As a result, the root is not flooded with many single requests but receives just a few larger ones. When the root detects a new node after merging together all the requests (or when it is time to refresh the tree) the channel's multicast tree is updated. The new tree is obtained by finding the minimum spanning tree covering the root and the subscribed nodes, according to the current state of the backbone. It is then propagated down to all nodes, by having each node send it to its children and so on, according to the topology conveyed in the updated tree. A node knows that it has joined the multicast group when it receives a tree update that includes it; to leave the multicast tree it sends *LeaveRequest* packets directly to the root packet until it gets a confirmation; the root in turn updates the tree in response.

4.1.2 The Desktop

The desktop is the main application that glues everything together. Its purpose is more than just to be a background where the tools are launched and manipulated. It has the important role of managing the group session by monitoring the status of its participants and providing the necessary binding information that turns the isolated tool instances into a closely coupled group.

The desktop relies on a dedicated event channel for its operation. The name of this event channel identifies the session that the user is joining. The remaining tools rely also on this name to complete binding information by appending appropriate suffixes to derive their own event channels' names.

During the course of its operation, the desktop uses its event channel to publish a periodic heartbeat that informs other desktops in the same session about the presence of this participant. The desktop collects these heartbeats (including its own) to keep a list of the session's participants. This list is presented graphically on the left side of the desktop, showing both online participants and offline ones. A participant is considered offline if the last time its heartbeat has been heard exceeds a preset amount of time.

The type of event channel required for the correct operation of the desktop in the terms described differs from the one used in the video conferencing tool in the fact that it has a clear a multi-source requirement. An unreliable type can be used and has been developed but we later decided to replace it with a reliable version. The difference being that a reliable event channel allows for a tighter tolerance in heart beat timings because with a reliable event channel one only has to consider delayed heart beats, whereas with an unreliable one, lost heart beats must take into consideration and, therefore, one can only reasonably conclude that a participant is offline if a certain number of consecutive heart beats failed to arrive.

The two QoS template types were developed anyway, basically because it makes sense to produce the reliable version after the unreliable one. Moreover, the *UnreliableMulticast* QoS template is essentially an extension of the single-source version developed earlier. The changes made consisted in also having the nodes with sources join the multicast tree, in addition to the nodes with subscribers, and always choosing the rendezvous node as the root of the multicast tree. The *JoinRequest* handling and related multicast tree updating was kept the same. The only additional modification required was about the routing of the actual events. They no longer travel down the tree, as before, but at each node are sent away from their point of origin along the branches of the multicast tree (now interpreted as a graph).

This multicast routing algorithm will perform poorly if the number of nodes that are exclusively a source of events is much larger than the receiver nodes. However, this does apply in the case of the desktop application because every node is always both a source and a subscriber.

The *ReliableMulticast* template that was eventually used in the desktop application solves the problem of lost packets with a small fixed-sized packet queue, at each node of the multicast tree, one for each source. Holes in queue are filled by sending a negative acknowledgement packet, listing a certain number of missing packets, one hop towards the source. Every so often, a node is also required to send a packet, one hop towards the source, acknowledging the last event in sequence it received. At each level of the tree (in respect to the source in question) these ACK packets are aggregated into larger compound ones to avoid the problem known as ACK implosion. The source advances the queue in step with the lowest sequence numbered ACK received and drops any node that fails to advance its sequence number for too long.

4.1.3 Moderator Tool

The purpose of this tool is to help coordinate an ongoing videoconference session by muting the audio streams of selected participants, while keeping the video going. This tool is rather simple in its approach; it manages a queue of enrolled participants, monitoring changes to the queue and only allowing the participant at the head of the queue to talk, keeping the others silent. The actual tool consists of a simple graphics interface that shows the state of the queue, with its enrolled participants, and allows a participant to enter or leave the queue. No fault-tolerance features have been implemented but, given its overall informal nature, this problem would addressed by allowing anyone to remove a silent participant from the queue.

To keep it simple and peer-oriented, all instances of the tool behave in the same way, none having a special role. Changes to the queue are made by publishing *enter* or *leave* events to an event channel that every moderator tool (in the same session) subscribes, with the sanity of this whole process resting in the event channel's ability to keep all the queues consistent. The actual muting and enabling of the audio streams is done indirectly by publishing appropriate events to another event channel shared with the all the tools running on the same desktop, video-conference ones included. This is an event channel that only spans one particular desktop and is a clone of the built-in *LocalLoop* QoS template.

This simple approach to the moderator tool was thought viable on the assumption that a suitable QoS template could be developed easily enough to not completely offset what would be gained in the first place. Specifically, the moderator tool required a multi-source reliable multicast event channel, with the additional need for a consistent ordering of events for all subscribers. Our bet was that it would be possible to adapt one of the existing QoS templates and, with a modest effort, turn it into what was necessary. It turned out that it was, indeed, a rather simple task to extend the existing *ReliableMulticast* template into a *TotalOrder ReliableMulticast* version that also guaranties that every node receives events in exactly the same order. Basically, the adaptation consisted in having the rendezvous node serve as a sequencer and establish the globally perceived ordering of the events, by embedding in the event stream a new control message stream relating the source sequence number of each event to the total order of the channel. The reliability mechanism already used in the event stream also applies to these new control messages thus avoiding any gaps in the total order sequence numbers. In each node, events are delivered to the application when both the next in sequence mapping message and the corresponding event have arrived. This solution to the problem is not novel but we feel that it adds additional proof to the extensibility claims of the framework.

4.1.4 Chat Tool

This tool allows the users in a session to engage in a written dialog. It follows a similar approach to the one used in the moderator by having all the instances of the tool share exactly the same role. Consequently, the chat tool also shares with the moderator tool the same QoS requirements for its event channel, thus allowing us to re-use the same QoS template plug-in already developed for the moderator tool. As a result, the chat too is very small and simple. Basically, it only needs to publish the text input by the user into a dedicated event channel that every chat tool also subscribes to receive what the other users are saying. When a new event arrives, a log of the messages received is converted to HTML code to be presented, taking advantage of JAVA support for this format. To dress up the chat tool, and by taking further advantage of the HTML rendering capabilities of the JAVA environment, we opted for presenting each message side by side with the icon image associated with its author. The real motivation was that with only a replacement of the default protocol handler of the JAVA environment we managed to use the *system registry* as the URL source for those images and exploit and evaluate its location independent addressing, load on demand and caching capabilities.

Our next step to improve this chat tool has been to get it to replay the history of the messages exchanged in previous sessions. To keep the changes in line with the overall philosophy, we would like to accomplish this without modifying the application code. Specifically, the goal is to replace the event channel type, currently in use, with a new type also advertising a persistency quality. Such a channel type would replay past events to a new subscriber before catching up with the rest of the group. Again, we feel this is quite achievable by extending an existing QoS template and developing a persistency support service.

5. TRIAL EXPERIMENTS

The group conference tool suite described in the preceding sections has been tested on a limited scenario in terms of the number of backbone nodes used. For practical reasons, the evaluation of the correctness of the QoS template plug-ins in scenarios involving a realistic number of backbone nodes was done by simulation only. For this task, we used the framework's built-in simulator to run the unchanged QoS templates in random networks with up to 100 backbone nodes during several hours of virtual time. To stress the routing algorithms and to rapidly expose any errors, very aggressive (and unrealistic) packet loss rates of up to 50% were tested. The algorithms behaved as expected, delivering the promised QoS. Actual performance data was not gathered at this time because the goal of the current line of work is not the design of overlay routing algorithms per se but to prove that the framework's proclaimed extensibility and programmability lives up to expectations. In this respect, we were pleased to confirm that the DEEDS framework does, indeed, support the coding and adaptation of elaborate routing algorithms in a natural and straightforward manner.

Testing of the actual group conference tool suite has involved, so far, a LAN DEEDS environment setup with the following characteristics. The dissemination network consisted in just two backbone nodes to which the desktop applications were connected directly; therefore, no secondary servers (second tier nodes) were used. Heterogeneous mixes of TCP, UDP and IP multicast *transports* were employed to assemble the network. Specific transport bindings were setup for each event channel, depending on the template involved. Reliable channels were set to use TCP across the entire network, while unreliable ones were set to use UDP between the two backbone nodes (with a 200 ms imposed delay) and IP multicast among the clients of the same backbone node and itself. It is worth mentioning that the choice of specific protocol bindings is a node configuration procedure that is meant to reflect local administrative practices of a particular site. Although, choices of protocol bindings can and will affect QoS template performance, the templates themselves cannot programmatically specify or enforce a particular configuration.

Informal testing with the network configuration described above, conducted with a group of up to four participants, has shown that the tools behave in an acceptable way despite their prototypal nature. In particular, the more demanding videoconference tool showed that the overhead inherent to the framework is not too impairing. Conferencing using audio alone worked particularly well but video suffered a noticeable frame drop. A more careful analysis of the problem revealed that video alone worked fine and that the problem was more apparent when audio and video were used together. This led us to think the problem was in the tool itself and not in the actual dissemination process. This suspicion was confirmed when the same tests conducted over pure IP multicast exposed the same problem.

6. FUTURE WORK

Results obtained from this case study have been very encouraging and strengthened our motivation to continue the validation of the DEEDS' event dissemination model and architecture. To that end, we will next evaluate how key problems, such as, efficient routing based on aggressive filtering policies or content-based routing problems, can be solved using the framework. We would like to incorporate any results from these efforts to expand the usefulness of the *system registry* in application design beyond that already tried in the chat tool. The other major undertaking still required is to evaluate the impact of enhancing the framework with security related features. More specifically, we intend to incorporate signed code techniques to the load on demand procedure of QoS template plug-ins and introduce other cryptography elements to protect the overlay network from outside interference and eavesdropping.

7. RELATED WORK

The lack of Internet-wide, reliable "native" multicasting support has fuelled the search for several middleware solutions to the information dissemination problem. Horus[1] and iBus[2] are two paradigmatic middleware messaging systems that addressed the problem of group-oriented communication with customizable QoS guaranties. In these systems, QoS is offered by layered protocol composition, by means of extensible protocol stacks. The chosen communication model is strongly biased towards peer-to-peer computing between end applications, without or with very limited intervention of support servers. Our work differs greatly both in scope and approach. We advocate a solution that includes support for large-scale scenarios, whereas theirs is essentially targeted at LAN environments. We also address the problem of QoS in a radical different way; preferring non end-to-end oriented protocols according to principles inspired from *active-networking* [5] research but adapted to the specifics of overlay networking.

The problem of QoS handling in the specific context of publish/subscribe systems has also been discussed in [6]. In this work, QoS based delivery of events is exposed at the programming language level using a framework of "asynchronous collections" that offers familiar object-oriented programming abstractions for handling information, such as bags, sets, arrays, lists, sorted sets, etc. Little information is given about the underlying architecture.

Siena[6], Elvin[8] and Gryphon[9] are noteworthy examples of elaborate event systems, based on content-based subscription. In these systems, event consumers subscribe from a global pool of structured events by providing sophisticated filter expressions, which must be evaluated against incoming events to determine those of interest. In [8], Elvin is described as a non-scalable, centralized solution but, on the plus side, offers support for client disconnection. Both Siena and Gryphon address scalability issues by migrating subscription expressions over decentralized multi-server architectures. These platforms pursue, mainly, optimized content-based solutions based on a fixed set of routing protocols. Being a framework, DEEDS lacks most of the specific event algebra processing engines of these systems but, on the other hand, its extensibility offers a larger potential for the support of a broader range of scenarios. It also puts a greater emphasis on the dissemination component of distributed event systems.

[10][11][12] are systems that also tackle the problem of multicasting in overlay-network environments, each offering a specific multicast routing algorithm and a fixed protocol for the self-organization of the overlay network. They differ mainly in those respects to our offering, because ours has been designed from the ground up to be extended with new routing algorithms via pluggable templates.

Finally, discussion on group-oriented meeting tools can be found at [13][14], which are important references in their field. These systems are particularly good examples of the pragmatic tendency of choosing centralized client/server solutions whenever that is acceptable. Our work has hinted that fully distributed, more fault-tolerant solutions can be viable alternatives to that model.

8. CONCLUSIONS

The design and implementation of this case study has been very helpful in our work on the development of DEEDS, a programmable and extensible event dissemination framework. It has strengthened our belief in the soundness of our goals and in the design decisions made so far.

It confirms the viability of the programming model advocated in the framework, which claims that simple, yet, effective event-aware distributed applications can be built on top of an overlay network communication infrastructure, provided the most natural or straightforward requirements in quality of service are met. This conviction comes from the fact that elaborate routing protocols, offering diverse types of QoS, were developed, readily, and in the form of pluggable and re-usable extensions to the dissemination framework, perfectly in line with our expectations.

This case study also provided evidence that DEEDS offers enough built-in conveniences to make it is relatively easy to adapt existing documented routing algorithms into its overlay networking environment. Is has also shown that the creation of new QoS plug-in templates can follow an incremental approach from previously developed ones. The active networking inspired plug-in model represents great versatility because it encourages the use of tweaked variants of the same plug-in as a form of optimization for specific requisites, instead of having to settle with an overall best one.

Another area of framework design that confirmed its value was the adoption of a *protocol agnostic* approach to the programming model. It showed that there are obvious advantages in supporting protocol heterogeneity in a independent manner to the programming of new applications and template plug-ins. Allowing the choice of actual bindings between event channels and underlying communication protocols to be left to the deployment phase proves to be sensible, because it can be changed at any time and so can be better adapted to what is available in each particular circumstance at a given time. Overall, it was made clear that the adoption of protocol heterogeneity will offer a more diverse and richer realm of deployment possibilities.

Finally, we feel the results obtained so far encourage us to continue the validation process of the dissemination framework by tackling other areas of the distributed event dissemination problem along the lines exposed in the future work section above.

9. REFERENCES

[1] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. Communications of the ACM, 39(4):76--83, April 1996.

[2] M. Altherr, M. Erzberger and S. Maffeis. "iBus - A Software Bus Middleware for the Java Platform". In International Workshop on Reliable Middleware Systems, p. 43-53, October 1999.

[3] Schulzrinne, A., Casner, S., "RTP: A Transport Protocol for REAL-Time Applications", Internet Engineering Task Force, Internet Draft, Oct. 20, 1993.

[4] Sun Microsystems. "Java Media Framework 2.0 API Guide". http://java.sun.com/jmf. 1999.

[5] J M. Smith, et al. "Activating Networks: A Progress Report". IEEE Computer, Vol. 32, No. 4, p. 32-41, April 1999.

[6] P. Eugster, R. Guerraoui, J. Sventek. “Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction”. ECOOP, pp. 252-276, 2000

[7] A. Carzaniga, D. S. Rosenblum and A. Wolf. “Achieving Scalability and Expressiveness in an Internet-scale Event Notification Service”. In Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC-00), July 2000.

[8] B. Segall, D. Arnold. “Elvin has left the building: A publish/subscribe notification service with quenching”. In Proceedings of AUUG97, Brisbane, 1997.

[9] G. Banavar et al. “An efficient multicast protocol for content-based publish-subscribe systems. In the 19th IEEE International Conference on Distributed Systems (ICDCS’99), May 1999

[10] Yang-Hua Chu, Sanjay Rao, and Hui Zhang. “A case for end system multicast”. In Proceedings of ACM Sigmetrics, Santa Clara, CA, 2000

[11] J. Jannotti, D. Gifford, K Johnson, M.Kaashoek and J. O’Toole Jr.”Overcast: Reliable Multicasting with an Overlay Network”. In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, pp 197-212. USENIX Association October 2000.

[12] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. “Almi: An application level multicast infrastructure.” In Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS), pages 49--60, 2001.

[13] K. Watanabe et. al. “Distributed Multiparty Desktop Conferencing System: MERMAID”. In Proceedings of the Conference on Computer-Supported Cooperative Work, Los Angeles, CA, September 1990.

[14] T. Crowley, P. Milazzo, E. Baker, H. Forsdick, and R. Tomlinson. “MMConf: An infrastructure for building shared multimedia applications”. In Proc. CSCW'90, pages 637--650, Los Angeles, CA, October 1990.

10. APPENDIX

The sample below represents the main JAVA class that implements the *UnreliableMulticast* QoS template plug-in. This refers to the template code that executes in the context of the backbone (1st tier). Two other simpler classes provide the routing logic for the 2nd and 3rd tier of the overlay network. Most of group membership management, spanning tree calculation is done in the NodeGroup class, not shown here. The actual sendTo methods are found in the base class. These methods take a node id or a collection of node ids and forward events by selecting the appropriate *transports.* That selection is based on information kept in a *container* that tracks changes in primary node data.

```
package deeds.sys.templates.unreliable.e;
//…removed list of imports.
public class p_UnreliableMulticast extends ControlPacketRouter {

    public p_UnreliableMulticast(GUID channel) {
        super( channel ) ;
    }
    public void init() {
        super.init() ;
        Container.monitor( "p-" + this.channel(), new ContainerListener() {
            public void handleContainerChanges(Container c) {
                sc = (SubscriptionContainer) c.item("SubscriptionContainer") ;
                isMember = ! sc.isEmpty() || isRendezVousNode() ;
            }
        }) ;
        members = new NodeGroup( links ) ;
        lauchRefreshMembershipsTask() ;
    }
    // routes the actual published events
    public void pRoute( pDataEnvelope de ) throws Exception {
        if( de.isLocalEvent() || de.isMinorEvent() ) isSource = true ;
        sendTo( members.children( de.src.major() ), de ) ;
        loq.send( de ) ;
    }
    // routes the actual feedbacked events
    public void fRoute( fDataEnvelope de ) throws Exception {
        Object node = de.dst.major() ;
        if( node.equals( thisNode ) ) loq.send( de ) ;
        else sendTo( node, de ) ;
    }
    void cRoute( JoinGroupRequest r ) {
        if( members.addAll( r.members() ) ) {
            if( isRendezVousNode() ) {
                sendTo( members.root(),
                    new MulticastTreeUpdate( channel(), members.freshTree() ) ) ;
            }
            else {
                cDataEnvelope nr = new JoinGroupRequest( channel(), members.nids());
                sendTo( members.parentOrDefault( rendezVousNode() ), nr ) ;
            }
        }
    }
    void cRoute( LeaveGroupRequest r ) {
        boolean changed = members.remove( r.src() ) ;
        if( isRendezVousNode() ) {
            if( changed ) {
                sendTo( members.root(),
                    new MulticastTreeUpdate( channel(), members.freshTree() ) ) ;
            }
            sendTo( r.src(), new LeaveGroupAck( channel(), r.src() ) ) ;
        }
        else sendTo( members.parentOrDefault( rendezVousNode() ), r ) ;
    }
    void cRoute( LeaveGroupAck a ) {
        if( a.matches( thisNode ) ) joinedGroup = false ;
    }
    void cRoute( MulticastTreeUpdate u ) {
        joinedGroup = u.contains( thisNode ) ;
        members.updateTree( thisNode, u.ste ) ;
        sendTo( members.children(), u ) ;
    }

    private void lauchRefreshMembershipsTask () {
        new PeriodicTask( 0, 60000 ) {
            public void run() {
                if( isMember || isSource ) {
                    isMember = true ;
                    members.add( thisNode ) ;
                    cRoute( new JoinGroupRequest( channel(), members.nids() ) ) ;
                }
                else
                    If( joinedGroup ) cRoute( new LeaveGroupRequest( channel() ) ) ;
            }
        } ;
    }
    private NodeGroup members ;
    private boolean isSource = false ;
    private boolean isMember = false ;
    private boolean joinedGroup = false ;
    private SubscriptionContainer sc = null ;
    private NetworkLinks links = (NetworkLinks) Singleton.get("NetworkLinks"
```