DEEDS

A Distributed and Extensible Event Dissemination Service

Abstract

In this paper, we present ongoing research on the development of a general-purpose event dissemination service that integrates support for stationary and mobile systems in a common seamless framework. We describe the features of an event dissemination model based on a *publish/subscribe/feedbac*k paradigm over *active event channels,* which incorporate specific routing logic to offer custom quality of service guaranties and programmable forms of subscriber addressing. A matching architecture that tackles large-scale and heterogeneity concerns is also outlined. Finally, we illustrate the potential of the framework in the context of a few multi-user collaborative settings.

1 Motivation

Distributed event-driven programming is a recognized paradigm, used in many different areas of application development, such as distributed GUIs, multimedia and CSCW applications, surveillance and monitoring software, shared virtual reality environments and multi-player games.

The usefulness and widespread use of events arises from the very simplicity of the concept and its convenience in expressing interactions with varying degrees of coupling between the involved peers. As such, events are a particularly interesting tool for gluing together assorted, heterogeneous components into complex applications [1].

These characteristics explain why events play a decisive role in the field of Distributed Systems, as is shown by the fact that Corba[2], JavaBeans[3], DCOM[4], Jini[5], and every other major platform for distribution support, includes some sort of event exchange mechanism. Yet, in spite of promoting the use of events, when it comes to event dissemination in scenarios involving large-scale, mobility or heterogeneity, these systems offer solutions that are invariably limited. In the generality of cases, realistic uses are restricted to local area environments, where delivery of events is fast and rather predictable. This widespread inadequacy to less forgiving environments often leads to custom adaptations and had-hoc approaches, instead of more demanding, fully re-usable solutions. This is not a short-lived problem or setback. The promise of a wired world of pervasive nomadic computing devices in a not too distant future, is only going to demand a bigger role for mobile and large-scale platforms and highlight the value of specific and adequate support.

Heterogeneity is also on the rise. The profusion of Internet-ready appliances and Internet-based services, we see today, is turning the Internet into an increasingly diverse medium, moving it away from the original model. Furthermore, uncoordinated administrative barriers are constraining more and more the free flow of information. That, and the enormous differences in capability of the computing devices available today, are important obstacles to models that follow “one size fits all” approaches.

The combined effects of these trends, hint that the engineering of distributed large-scale applications is not getting any easier, especially if one tries to avoid the way of the least common denominator (well portrayed by mass adoption of HTTP as the panacea to every connectivity problem). Based on these assumptions, we believe that an event dissemination middleware that puts special emphasis on its extensibility and in the programmability of every aspect related to the processing and transport of events is a good answer to many of the various problems posed by heterogeneous, mobile and large-scale environments.

In the next few sections, we describe, DEEDS, our ongoing efforts in the development of a system with the goals listed above. First, we discuss the key features of the proposed event dissemination

model and of its support architecture. Next, we summarize the programming model and illustrate possible uses of the system in specific contexts and application scenarios. We, then, conclude the document by presenting related work in the area, followed by some final remarks.

2 Overview of DEEDS

DEEDS is a JAVA-based event dissemination platform designed to be as flexible and adaptable as possible, aimed at a broad range of applications and execution scenarios. The guiding principles of the framework are the extensibility and tailorability of existing features as a way of satisfying the requirements of large-scale, heterogeneity and mobility in specific contexts.

The solution advocated is general-purpose in the sense that it can be readily 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 is deliberately used to foster an incremental approach towards problem solving that capitalizes on existing experience.

DEEDS follows the well-known *publish/subscribe* paradigm with the exception that a *feedback* operation, which allows event consumers reach back the event publishers, is given equal importance in the overall model. A fully featured *feedback* operation is provided because it is reasonable to assume that, in many situations, event-receivers will want to talk to the publishers of incoming events, on an individual basis, using strictly *unicast* semantics. Given the fact that administrative barriers (firewalls) and protocol heterogeneity pose the same obstacles to *unicast* as for *multicast* traffic, it makes sense to enroll both in a common programmable framework. Ultimately, we can offer solutions in which *unicast* event streams can change transports along the way or that, for instance, avoid choking a publisher with *unicast* traffic by selectively discarding it closer to its source.

DEEDS also expands the familiar notion of event channels to provide event dissemination operations over *active event channels*, a concept that borrows ideas from the domain of *active networks[6]*, as is explained further on.

Traditionally, event channels symbolize logical addresses that serve as meeting points and, often, take the role of mediators between publishers and subscribers so that decoupled interactions (in space and time) can happen among them. The name given to an event channel frequently hints on the subject of the events it carries. Used in this fashion, they provide structure to the event flow.

The *active event channels* found in DEEDS share those attributes but the model stipulates, as well, that each channel, individually, has an intrinsic *quality of service* (QoS) attached to it. That QoS is expressed by a combination of abstract qualities such as reliability, persistency capabilities, type of subscriber addressing (*unicast*, *multicast*, *anycast*, etc), ordering, latency, failure model and so on. One key aspect is that the QoS of a channel is not contracted or negotiated; it is a fixed characteristic, established programmatically, whose trustworthiness relies on a rational parameterization and deployment of the event channel architecture.

The QoS attribute is, without doubt, the central piece of the event dissemination model and in order to provide the maximum possible generality a toned down form of *active networking* has been adopted.

Active networking[6] doctrine states that transmitted-data carries along the code (or reference to the code) that will control its forwarding along the transmission path. That code, specified in a packet-by-packet basis, is meant to lodge itself in network routers and execute there the routing logic that creates custom-made communication protocols. This is a bold departure from traditional practice in computer networking and for that reason alone faces considerable resistance but, in truth, some criticism from security and performance concerns is not uncalled-for. Still, we believe that with some adaptations and in the context of event dissemination within a confined network infrastructure, the advantages of an active networking inspired solution can easily offset its drawbacks. To that end, we made changes consisting in having all events flowing in the same event

channel share a common routing logic, encased in special objects called *system routing assistants*. This agrees well with the intention of having a specific QoS for each event channel but has the additional benefits of simplifying the task of deploying the routing logic efficiently and of allowing for safer operation of the network routers, which are freed of the dangers of executing arbitrary code. Moreover, because system routing assistants are system-level objects that applications do not know about, they can be replaced transparently when an improved version comes along.

The event dissemination model defended is also protocol transparent from the application perspective. This allows the same application to be deployed in a wider range of networking environments. However, it comes at the expense of a more complicated *system routing assistant* design, which must be capable of coaxing the protocols available at various network nodes into delivering the channel's advertised QoS (assuming a set of minimum requirements is met). Despite the added complexity, protocol transparency is very desirable because it allows for interesting scenarios such as having an event channel efficiently service a cluster of applications in a local-area domain using IP multicast and, at the same time, use TCP or HTTP to reach a remote domain, where a different protocol arrangement may be in place.

Given the various limitations inherent to mobile computers [7][8], when compared to their stationary counterparts, it is not feasible to handle them in the same manner. Specifically, variable and intermittent connectivity and long periods of voluntary disconnection that mobile devices are expected to experience are too disruptive to have these take part in a global cooperative event dissemination effort. For this reason, our model calls for a backbone of stationary, well-equipped servers to form the core of the event dissemination network. Lesser systems, such as desktop machines and mobile computers in particular, are relieved to secondary roles and must connect to the fringes of the network to gain access. This arrangement puts much less strain on *system routing assistant* development (because it is reasonable to expect that the conditions in the backbone will be rather stable and change slowly over time) and clears the way for a framework dedicated to the needs of mobile systems.

Catering to the requirements of mobile applications is achieved through the *application routing assistant* framework. This framework aims to standardise the way mobile applications deal with event filtering, prioritisation, digestion and any other fine-grained event management procedures that are meant to adapt the standard QoS of an event channel to the specific needs of an application, according to the local network resources available at the time. The foundation of this framework consists of *application routing assistants* objects that are supplied by the applications with the purpose of steering the event flow between them and the backbone dissemination network. Two classes of these objects exist. One manages the *outflow* of events towards the network and for this reason its objects stay close to their parent applications. The other deals with the *inflow* of events towards the mobile application. Accordingly, objects of this sort are sent to a harbouring backbone server. This latter kind of *application routing assistant* is particularly interesting because the servers where they are anchored serve as *home bases* that are impervious to the fluctuating conditions of mobile links, allowing them to act on behalf of their poorly connected or offline parent applications.

3 Architecture Overview

The DEEDS' event dissemination model presented in the previous section is complemented by a distributed architecture with a strong commitment to large-scale support.

Expanding on what has already been hinted before, the core of event dissemination infrastructure consists of a federation of interconnected servers that forms a logical network within the network. Depending on availability and local administrative policies, each server is linked to several others using a variety of redundant transport-level connections (*transports*) such as: TCP, UDP, IP Multicast, HTTP, EMail, etc. The *system routing assistants*, described earlier, execute in each server/router to provide the routing logic that drives the forwarding of events among the participants

of the logical network. Secondary servers, operating in lesser machines, also take part in this network but are given limited routing responsibilities and are mostly required to link-up with a primary server to provide connectivity to their client applications. All non-mobile servers can also serve as anchor points (*home bases*) for the *application routing assistant* plug-ins.

Besides supplying the computing resources for *system routing assistants,* each server also supports several elementary services that use the networking infrastructure in a recursive manner to gather information about the event dissemination network as a whole. Examples of these services are: the "Discovery Service" that periodically probes network entry-points (URLs) to enable servers join the network federation; and the "Hello Service" that collects round-trip delay information to others servers. Data obtained and processed by these and other simple services is made available to *system routing assistants* and other services to assist in the creation of more elaborate event channels, which in turn feed more complex services, in accordance to the professed incremental (recursive) principle.

The final key element of the architecture consists of an object repository, designated as the *system registry*, where static configuration and dynamically collected data is kept. The system registry acts mostly as a *cache* that, depending on scope, partially replicates data from other servers of the federation. Items that should be visible globally, such as the event channel directory and the code of routing assistants are (usually) found in every server registry but others are only loaded on demand. In both cases, special purpose event channels are used to keep the various registries consistent and, in particular, to perform expanding searches in other registries to resolve cache faults.

4 Programming Model

DEEDS programming model is expressed in the JAVA programming language and assumes execution in a standard JAVA environment. The framework is composed by user-level programming interfaces, used in the development of event-based applications and *application routing assistant* objects. Furthermore, system-level interfaces are available for system enhancement and administration, which include facilities for the creation of additional server support services and new *system routing assistant* and *transport* types.

Events in DEEDS are small, self-contained notifications, composed by a pair of items: an arbitrary serializable JAVA object and an *envelope* object, whose particular class is specific to each event channel type. Envelope objects supply control information to the event dissemination infrastructure as required by the particular QoS of the event channel in question. Their use allows for interesting and varied types of event channels. For instance, an envelope consisting in an expiration deadline could be used to have an event channel automatically discard late events before reaching some its subscribers, freeing network resources earlier. In a different arrangement, an event channel that advertises a persistency capability might require its envelope to include a rough mapping of contents of each event to a relational database notation. A similar type of envelope could be used to create event channels that operate according to some sophisticated content-based subscription policy. Overall, the use of *envelopes* (and their counterparts in *subscription* operations) makes publish/subscribe operations very powerful and greatly adds to the flexibility of the *system routing assistant* framework.

The programming model also includes the notion of *Receipt* objects, which report system-generated information that is not found on event envelopes, such as event-source identifiers. Receipts, for instance, are used to designate the target of event feedback operations.

The code excerpt found ahead, exemplifies the use of the main programming interfaces in two basic *publisher* and *subscriber* applications.

```
import deeds.api.core.* ;

class Publisher implements Runnable, EventFeedbackSubscriber {

    EventChannel channel ;

    public Publisher() {
        new Thread( this ) .start() ;
    }

    public void run() {

        channel = DeedsSystem.getChannelDirectory().lookup("/aChannel") ;

        channel.subscribeFeedback( new SubscriptionControl(...), this, ...) ;

        while(...) {

            ...
            channel.publish( new Envelope(...), new someEvent()) ;
            ...
        }

        channel.unsubscribe(...) ;
    }

    public void nofifyFeedback( Receipt receipt, MarshalledEvent mev ) {

        Object o = mev.getEvent() ;
        Envelope e = receipt.getEnvelope() ;
        ...
    }

    static public void main( String[] args ) {
        new Publisher() ;
    }
}
```

```
import deeds.api.core.* ;

class Subscriber implements Runnable, EventSubscriber {

    EventChannel channel ;

    public Subscriber() {

        channel = DeedsSystem.getChannelDirectory().lookup("/aChannel");

        channel.subscribe( new SubscriptionControl(...), this, ...) ;
    }

    public void nofify( Receipt receipt, MarshalledEvent mev ) {

        Object o = mev.getEvent() ;
        Envelope e = receipt.getEvent() ;
        ...

        channel.feedback( receipt, new Envelope(...), new feedbackEvent()) ;
    }

    static public void main( String[] args ) {
        new Subscriber() ;
    }
}
```

For clarity and brevity, only partial argument lists are shown. In addition to that, in a real-life scenario, the applications would have to supply "Envelope" and "SubscriptionControl" objects of the class specified by the event channel in question. (If *application routing assistants* were to be used, they would be specified in extended versions of the methods presented.)

4.1 User-Level Plug-ins

The micro-management of the event flow between applications and the dissemination infrastructure is performed using a specific programming model - the *routing assistant framework*. In short, the purpose of this framework is to normalize most, if not all, application procedures regarding event filtering, event prioritisation, event digestion, event-mailbox management, and other adaptations imposed by connectivity limitations. This is achieved by supplying *application routing assistant* objects as additional parameters for *publish* and *subscribe* operations, effectively installing application-defined behaviour into the event dissemination infrastructure. Under this scheme the main body of applications is focused on interpreting events and leaves much of the rest to be handled by application routing assistants objects.

Applications routing assistants are, basically, a pipeline or queue in which events flow from the network towards the parent application or in the opposite direction, depending on their type. Their first job is to decide which events enter the queue, as they arrive, by analysing the event's *receipts*, *envelopes* or contents. Events that do reach the event queue are sorted according to a previously negotiated policy. Next, they are presented again to the routing assistant for dispatching to one or more of the available transports. During this phase, the routing assistant can query the system about network conditions and the properties of the various transports and, based on that information, route, delay or discard the event. Routing assistants also have the chance of injecting new events into the queue or replace existing ones. Moreover, the event queue, the routing assistant itself, and a data-storage scratch pad can be flagged as persistent to further expand the possible uses of the framework.

4.2 System-level Plug-ins

New classes of event channels are added to the system using special plug-ins, known as *system routing assistants,* whose creation involves two equally important steps. One, as expected, deals with the programming aspects of the design of a routing assistant object. The other is subtler and consists in the precise documentation of the new channel specifications in terms of the advertised QoS and its execution requirements. This is vital information about a new channel class because the QoS will be a key reference to application programmers and the execution requirements serve as guidelines to system administrators when deploying this type of event channel in particular environments.

System routing assistants are rather simple constructs if one just takes into account that they are only expected to dispatch the events the underlying system presents them to. Specifically, they must manage two separate streams of events: the multi-point stream that is produced by *publish*-operations, and the *unicast* stream consisting of *feedback* events. If desired, it is possible to delegate to a different system routing assistant class the task of handling one of those streams. This is the likely behaviour for the *unicast* stream, which most routing assistants will probably pass un-altered to the system-provided default *unicast* router.

Unless, the QoS involved is very basic, in reality, system routing assistants are going to be complex objects. Therefore, to ease their development and capitalize on already available resources (for instance, a pre-computed routing table), they may rely on the system object registry to gather information or to obtain references to support services. These are 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 system routing assistants to synchronize their state (its own particular routing table, for example) in reaction to changes in the *containers* they monitor. Since the type and amount of information that can be made available through *containers* is not limited in any way and can be extended at any time, we believe these simple programming resources are an adequate way of adapting the overall system to the needs of present and future event channels developers. Nevertheless, issues, such as security and resource consumption, that are particularly important in active networking contexts[6] still need be fully analysed but we expect our future work in the area will not be hindered by the present model decisions.

5 Application Scenarios

In this section, we describe possible uses for our event dissemination service, in settings where its programmable features particularly standout. We hope these will demonstrate the versatility of the *system routing assistant* and *application routing assistant* frameworks.

5.1 The Job-Offer Network

The various regional branches of a state-run unemployment-fighting agency have embraced the digital revolution and want to offer the public with a specialized information channel of job opportunities. The agency wants job seekers to be able to search their databases for available offers and allow potential employers to advertise new job positions. The organization wants to keep each branch autonomous and under a de-centralized management. The only centralized resource is the front-end Web site of the agency, where software downloads are available and a simple “ticker” applet is shown with some of the most recent job offers.

An outline solution for this scenario calls for a dedicated server in each branch, supporting a number of specialized event channels.

- A best-effort, volatile multicast event channel is used to publish new job opportunities, as they are made available. These events will be posted regularly with a decreasing frequency. This

event channel feeds the agency web-site ticker and a similar one found on the client application that gives the public access to the system. The agency's statistical department also subscribes this event channel to conduct studies about the matter. Servers of the larger branches can listen to this channel, as well, to keep a cache of the offers available on other regions, in an effort to promote quicker searches.

- A reliable, "anycast-like" event channel is used to perform expanding distributed searches on the various databases. The client application includes a form that allows the job seeker to express his/her preferences and constraints (which are expected to reflect regional preferences) and encode them into an inquiry event. Special routing logic within the channel's *system routing assistant* analyses each inquiry and based on it may decide to direct the search to a more appropriate server, terminate it or expand it to neighbouring branches, depending on the volume of results already produced.
- A reliable "*geocast"* channel is used to route new job opportunities to the appropriate institution regional branch as employers post them.

Further refinements in the client application may give the user the option of installing an *application routing assistant* in a selected server, which will act as his/her agent while he/she is attending other matters. The assistant will monitor the event flow to perform a digest or select promising offers. If a particularly attractive job opportunity appears it might decide to page the job seeker immediately, using a SMS message or by sending an Email to a close-monitored account.

5.2 The Taxi Company

A taxi company operating in a big city wants to streamline its operations. Management is studying ways of increasing the efficiency of its fleet by minimizing both the time that is spent searching for new customers and the distance travelled to service booked runs. At the same time, they also want to implement a monitoring program to address the demands of their workforce for better security conditions during night shifts. To support the new enhanced operations each vehicle has been equipped with simple mobile computing devices and GPS systems. The company has secured a private unidirectional broadcast data channel and contracted the services of a wireless ISP to handle unicast communications.

A possible solution for this scenario could be developed around the following sketch:

- A reliable multicast event channel is used to collect new taxi runs, published there as events. These may have originated from various sources, such as the company's web site, WAP services, phone operators or pre-arranged bookings. A scheduler application subscribes this event channel, processes the runs and sends them to the fleet.
- An unreliable multicast event channel, supported by the contracted broadcast data channel, is used by the company's scheduler application to deliver runs to the fleet continuously. The vehicle's onboard computer subscribes this channel but filters each run so that, depending on the car's current position, only presents promising ones to the driver. Idle drivers or those close to the end of their runs compete to grab a new run. The event feedback mechanism of the event channel (configured to use the wireless unicast transport) is used to send (reliably) an event back to the company server requesting the run. At the company's headquarters, the scheduler application analyses the feedback events and uses the position data (also included in the event) to designate the winner of the run among the interested, according to some metric. A new event is sent to the multicast channel to inform which taxi got the run. This event is repeated a few times to cater for possible loss.
- Each taxi publishes periodically its current position to a multicast event channel (supported by the wireless unicast connection). A human-attended monitoring application subscribes this channel (and the previous one) and alerts the operator when a taxi veers off the expected route

or when it stops for too long, and so on. The human operator can then decide to contact the vehicle's driver and, if the need arises, to remotely request an audio/video transmission of the car's interior to better assess the situation.

- A persistent best-effort event channel is used to support audio/video streams coming from the taxi fleet.

5.3 Shared Whiteboard

We re-visit the traditional shared whiteboard scenario to illustrate a few additional uses of the *application routing assistant* framework. We will consider a shared whiteboard application with advanced multimedia capabilities, as a way of maximizing *awareness* dissemination and improve the collaborative potential of the program. For the sake of the argument, we will assume that the actions of the various users of the system are encoded into a steady flow of events that in ideal conditions produces observable effects at interactive rates.

A rough summary of the application's event channel layout is as follows:

- Best-effort multicast event channels are used to transmit separately encoded audio and video RTP streams. These channels' routing logic is such that it discards late frames, based on event temporal attributes; and efficiently filters the streams close to their sources according to the profile of subscriptions received. For instance, a user experiencing poor performance could unsubscribe the video portion of the signal keeping the audio feed or completely shutoff the transmission for that particular source. Under such scenario, the system routing assistant will prune the multicast tree accordingly to conserve backbone bandwidth.
- A reliable/FIFO multicast event channel is used to disseminate events encoding the operations performed on the whiteboard.
- A best-effort unreliable multicast event channel is used to disseminate the various forms of awareness information, such as the movements of mouse pointers of each user.

Although this application scenario is rather feasible in a stationary environment with ample networking resources, it would not be so in a setting involving freely roaming mobile computers. Therefore, we propose the use of *application routing assistants* to perform the necessary adaptations to a mobile environment, keeping the bulk of the application essentially the same.

Events leaving a mobile computer would need the following adaptations:

- The quality of the multimedia transmissions would have to be degraded to match the available bandwidth. Therefore, the assistant would probably stop sending information if very little or no connectivity was available at the moment. Under better conditions it could try lowering the video resolution or select a slower refresh rate. For audio, coarser bit rates would be used to encode the signal. Pushing the argument to the extreme it could also transcribe voice into text.
- The application routing assistant, dealing with the whiteboard operations channel, could degrade the data rate by eliminating intermediate operations or stack them into more compact ones. In times of offline operation or intermittent connectivity the more relevant events could be routed through a *transport* offering some sort of temporary storage, for instance one based on email.
- The awareness channel assistant would probably stop the flow of awareness information whenever its host computer is not docked or is disconnected from the network.

The inflow application routing assistants would behave according to similar principles as the ones presented above. They would have to degrade the rich flow of events coming from the network backbone into a coarser one to be funnelled through the mobile link. In the case of voluntary disconnected operation of the parent application they could be expected to perform digests of the collaborative work sessions that occurred during the period. These event digests could be stored until the application reconnects or be routed through an email based *transport*.

6 Related Work

Growing interest in the engineering of distributed applications using the familiar event-programming paradigm is reflected in the emergence, in recent years, of several event and data dissemination platforms. The following few represent a sample of the related work available in the area.

The absence of widespread, reliable "native" multicasting support has fuelled the quest for several middleware solutions to the information dissemination problem. iBus[9] is one of such solutions. Originally, it allowed Java applications to interact over a multicast channel abstraction, using a simple publish/subscribe model. Since then, commercial pressures lead it to evolve into supporting Sun Microsystems' bulky JMS API standard[10]. iBus supports protocol composition, by means of extensible protocol stacks, offering various qualities of service such as reliable multicast, virtual synchrony, encryption, among others. The preferred communication model is peer-to-peer, in which messages are volatile and flow directly from application to application, without intervention of support servers. A recent update of this system added support for persistent messages but requires funneling to a dedicated centralized hub. Some protocol heterogeneity and large-scale support is also being introduced with the help of bridging techniques.

TIB/Rendezvous[11] is an industrial messaging middleware that follows a subject-based publish/subscribe model over a hierarchical namespace. It offers a fixed set of QoS guaranties, such as a "reliable delivery", supported by temporary storage that holds messages for up to 60 seconds; a peer-to-peer "certified delivery", backed by the publisher runtime environment that buffers and retransmits information until all subscribers acknowledge reception; and a "transactionally guarantied delivery" supported by a centralized database solution. This platform lacks the overall consistency of DEEDS and seems excessively bound by the practical constraints of its commercial nature and target market, found mainly in the area of finances.

Smartsockets[12] is another industrial event channel dissemination solution that also uses Sun's JMS programming API [10]. Its most interesting feature lies in the promise of scalability using a network of servers. However, unlike DEEDS, the servers are not programmable and implement a rigid routing algorithm.

Cobea[12] is one of the several CORBA-based event architectures in existence. This one, in particular, uses a publish/register/notify paradigm that supports server-side event filtering based on pattern matching templates. This is actually a content-based subscription solution. Despite the use of filtering, the centralized client-server solution implemented does not really answer the problem of scalability.

Siena[14], Elvin[15] and Gryphon[16] are examples of elaborate content-based subscription solutions. These systems use structured events, whose contents have to be interpreted so that the appropriate subscribers are determined. In these systems, event consumers subscribe from a global pool of events by providing sophisticated filter expressions, which must be evaluated against incoming events. Elvin's, current incarnation, is a non-scalable, centralized solution but does offer support for disconnection. A distributed version is planned. Both Siena and Gryphon address scalability issues by migrating subscription expressions over decentralized multi-server architectures. These platforms pursue primarily the search for a highly optimised content-based solution, and therefore lack the general-purpose nature of DEEDS.

INS[17] is a resource and service discovery system that integrates name resolution with message routing in a dynamic network of computing devices. Two message delivery services are available: *intentional anycast* that targets an "optimal" destination name, and *intentional multicast*, which selects all destinations matching a given name. In both cases, communication is best-effort, without provisions for stronger guaranties, agreeing well with the fact that INS is better suited for service

discovery and binding rather than extended message exchanges. INS follows an approach that bears strong resemblances to the ones found on the three systems discussed above.

Salamander[18] is a wide-area data dissemination middleware, created to fulfil the needs in data dissemination of applications integrating a distributed laboratory. The publish/subscribe paradigm used is rich in features and offers key characteristics such as persistent database queries, resource announcement and discovery, adaptable quality of service, data persistency and support for client heterogeneity. This platform allows applications to interface with virtual distribution channels defined in an attribute data space supported by a tree of servers. Client applications connect to points of service (nodes) in the server tree to publish or consume data. Furthermore, application plug-in modules can be added along the distribution path to allow data modification (degradation) and filtering. On the surface, DEEDS shares several concepts with this system. However, they greatly differ on the granularity of the "events" that each handles. Salamander is focused on the dissemination and processing of large homogeneous data streams, such as those produced by the atmospheric monitoring stations that the project involves.

7 Concluding Remarks

We conclude this document by reiterating that DEEDS is an ongoing research effort. The team is still refining some of the less polished aspects of the framework, such as security and resource management. Nevertheless, preliminary results obtained from a demonstration prototype, whose development is under way, are positive and give us confidence that we are in a successful path. Our present priority is to conclude the prototype and then proceed with the modelling and development of paradigmatic real-life sample applications. We hope this process will confirm the soundness of the design decisions or, at least, allows us to address any shortcomings found.

8 References

[1] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, M. Spiteri, "Generic Support for Distributed Applications", In Journal "Computer", Volume 33, Number 3, p. 68-76, March 2000.

[2] Object Management Group. "Common Object Request Broker Architecture (CORBA)", http://www.omg.org

[3] Sun Microsystems, "Java Beans(TM) Manual", July 1997. http://java.sun.com/beans/

[4] Microsoft Corporation, "Microsoft COM Technologies", http://www.microsoft.com/com/tech/DCOM.asp

[5] J. Waldo, "The Jini Architecture for Network-centric Computing", CACM, July 1999, p. 76-82, http://www.sun.com/jini

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

[7] T. Imielinski, H. Korth. "Introduction to Mobile Computing. Mobile Computing - ed. T. Imielinski and H. Korth", Kluwer Academic Publisher, 1996.

[8] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, K. Walker. "Agile Application-Aware Adaptation for Mobility". In Proceedings of the 16th ACM Symposium on Operating Systems Principles, 1997.

[9] 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.

[10] M. Happner, R. Burridge and R. Sharma. "Java Message Service". Sun Microsystems Inc. October 1998.

[11] TIBCO, "TIB/Rendezvous White Paper". 1999. http://www.tibco.com.

[12] Talarian Corporation, "SmartSockets/SmartsocketsJMS Whitepapers". http://www.talarian.com.

[13] C. Ma and J. Bacon, "COBEA: A CORBA-Based Event Architecture". In proc. 4th Conference of Object-Oriented Technologies and Systems (COOTS-98), pp. 117 – 131, April 1998

[14] 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.

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

[16] 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.

[17] Adjie-Winoto W., Schartz E., Balakrishnan H., Lilley J., "The design and implementation of an intentional naming system (INS)". In Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99), December 1999.

[18] G. Malan, F. Jahanian and S. Subramanian. "Salamander: A Push-Subscribe Distribution Substrate for Internet Applications". In Proceedings of the USENIX Symposium on Internet Technologies and Systems, p. 171-181, December 1998.