DEEDS – a Distributed and Extensible Event Dissemination Service

Sérgio Duarte, José Legatheaux Martins,
Henrique J. Domingos and Nuno Preguiça
{smd, jalm, hj, nmp}@ di.fct.unl.pt

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Abstract

In this paper, we present our ongoing research on the development of a complete event dissemination solution aimed at a broad range of distributed environments, including both stationary and mobile systems. We detail an event dissemination model based on a *publish/subscribe/feedbac*k paradigm over active event channels that incorporate specific routing logic to achieve protocol transparency, custom quality of service guarantees and multiple delivery semantics. The main features of a matching programming model are also described, including those targeted at mobile application development.

1 Motivation

Event-driven programming is a recognized and well-established paradigm, present in a wide variety of computation scenarios. Asynchronous interactions, based on events, follow naturally in distributed computing and provide a particularly convenient and speedy way of engineering complex distributed applications out of assorted, off the shelf, possibly heterogeneous software components. Nevertheless, this convenience is somewhat offset by a relative lack of event dissemination tools capable of overcoming the challenges of large-scale and heterogeneous settings. Mass adoption of mobile and wireless computing habits and, further penetration of the Internet in every aspect of everyday life, will only enhance the need for systematic solutions rather than the ad-hoc approaches one has grown accustomed to witness when dealing with these environments.

Delivery of events in local area domains is usually well ordered, fast, predicable and rather reliable [2][3][4][5] [9]. These attributes mean that, when confined to local and short scale environments, the task of event dissemination maps well to a single or a small set of communication protocols. Naturally, under these rather benevolent conditions, research on event related studies has neglected, to some extent, the actual transport of events in favor of higher-level constructs, such as event algebras and elaborate content-based subscription languages [9][10][8][3][5]. However, as soon as either *large scale* or *mobility* is put into the mix, the job of event propagation takes serious proportions.

Mobility, on its part, introduces the newness of permanent disconnection and intermittent unreliable services with poor or variable quality of service warranties [12]. These are not exceptional conditions; they have to be accepted as normal modes of operation that require explicit support [13]. Large-scale event dissemination, on the other hand, faces obstacles mainly in the form of scalability concerns, heterogeneous transport protocols and administrative traffic barriers. No simple re-wiring of the communication protocol stack is capable of addressing the problem of scale.

In our view, these realities dictate that sensible approaches to the problem of efficient event dissemination, involving scale and mobility, should pay closer attention to the actual propagation of events, focusing on explicit support for heterogeneity and present it in a consistent manner that discourages non-reusable, ad hoc solutions to specific needs. It is along those lines, we shall present DEEDS, our ongoing work on the design of a comprehensive event dissemination architecture with the ambitious goal of providing seamless support for stationary and mobile users, regardless of the scale and heterogeneity of their distribution.

In the next three sections, we describe the main design and architectural features of DEEDS and the proposed programming model. In the remaining two, we compare our ongoing efforts to related work and issue some concluding remarks.

2 Overview of DEEDS

DEEDS is a JAVA-based, broad event dissemination solution developed with flexibility and extensibility in mind and intended to fit the needs of a wide range of applications and execution scenarios. It

features explicit support for mobile applications in the form of a series of specific abstractions, found across the event dissemination model and programming interfaces.

The event dissemination model advocated in DEEDS is based on a *publish/subscribe/feedback* paradigm over *active event channels*. These represent the enhancement of the notion of *event channel* with techniques inspired from the field of *active networks* [1].

Event channels are logical entities, with names that applications use as rendezvous points. They serve as intermediaries between event sources (*publishers*) and event consumers (*subscribers*), supporting asynchronous, temporally and space decoupled interactions between them. Furthermore, event channels are also useful as a structuring tool, organizing the event flow by aggregating related events.

In DEEDS, besides the name, a few other attributes characterize an *active event channel*. The most important of those is the publicized *quality of service* (QoS), represented in terms of abstract qualities such as reliability, type of latency, semantics of delivery, failure model, and so on. It is important to stress that quality of service is not negotiable, but a permanent aspect of each event channel individually. Meeting the advertised quality of service is, indeed, a problem, especially if one considers the sheer number of possible combinations of the QoS attributes and each site's local conditions. It is precisely to handle the problem that the *active* trait of DEEDS' event channels has been adopted. Under this *activation* scheme, each event channel is fitted with specific plug-in code, whose job is to interface with the underlying infrastructure and deliver the promised quality of service. These plug-ins, dubbed *system routing assistants*, can be made specific to a site's particular conditions to match available resources and administration policies. *System routing assistants* are system-level objects that end-user applications are un-ware of. For this reason, they can be (administratively) replaced at any time (for instance if an improved version is made available or if conditions change and a different kind is required). As such, their use constitutes the answer to DEEDS' pursued extensibility and tailorability.

DEEDS' also provides support for protocol heterogeneity, meaning that the quality of service of an event channel is not tied to a particular communication protocol. Contrary to that, event channels are meant to be, as much as possible, protocol transparent, so that it is up to the *system routing assistants* to pick up the transports available on a site that are considered more appropriate to meet their objective, as documented in their specifications. Therefore, it is perfectly possible to have a multicast-based protocol chosen to deliver events in a local area network and have, instead, a unicast protocol such as *http* deliver them to a remote location, behind a firewall, where a completely different scheme may be in place.

Providing specific support for mobile clients is another concern that has driven the design of DEEDS. To that end, the dissemination model incorporates provisions for event persistency, by allowing event channels to advertise an event playback capability. In this model, persistency is an attribute of event channels, not of individual events. Therefore, a *volatile* channel will discard events as soon as they are delivered, while a *persistent* one will have at least a part of its history available for playback.

Mobility is further addressed in the event dissemination model by extending the basic *publish/subscribe /feedback* paradigm with a specific framework that allows applications to fine-tune the way events are transported. The basis of this framework consists of special plug-in objects, dubbed *application routing assistants*, which applications supply when they publish events or subscribe a channel. These objects are expected to monitor network conditions and behave according to the needs of their parent application, for instance by filtering out low priority events when bandwidth is scarce, performing event digests or temporarily storing events for disconnected and offline applications. Operations such as these are common practice when dealing with mobile clients. DEEDS also supports these practices, but the paradigm that has been adopted goes further than that, and actively promotes a standard way of implementing those procedures that is much tidier and manageable than plain ad-hoc alternatives.

The event dissemination model, discussed so far, is powered by a distributed architecture, engineered towards the demands of scale. The following section portrays this support architecture in some detail and, in particular, describes how the servers self-organize to create the event dissemination network.

3 Architecture Overview

The core of the DEEDS' event dissemination architecture consists of a collection of stationary servers interconnected by various types of transport-level network (redundant) connections. The objective is to form a virtual (*backbone*) network, where servers act as routers and, the transport connections between them correspond to the links that define the network topology. Under this design, the *system routing assistants*, presented previously, provide the

routers' processing logic that directs the forwarding of events from the publishers to the subscribers.

To address comfortably the requirements of scale, the server network is organized into *domains*, encompassing servers that share a reasonable degree of system-administration coordination. Therefore, servers belonging to the same domain should experience a relatively homogenous view of the world, especially in what regards to transport availability and configuration of system routing assistants. This is not an absolute requirement but eases significantly the development of system routing assistants, and promotes a more efficient event dissemination overall.

Secondary servers, running on desktop or mobile personal computers, make up the remaining of the event dissemination network. Their primary job is to interface with the aforementioned backbone network and, in doing so, provide connectivity to client applications. These servers have limited routing responsibilities, relying on the services of a designated backbone server for all inter-server communication. Consequently, in contrast to what happens with backbone servers, the location and identity of secondary servers is not proactively advertised and has a limited scope of visibility, usually restricted to the designated primary server.

Stationary servers, regardless of their primary or secondary status, also serve as anchor points for remote *application routing assistants*. This capability is meant to provide mobile clients with a *home base* location, where to migrate portions of application code that will process events in their absence or that will adapt the event flow to match the observed conditions of the mobile link.

A system-wide replicated data cache, known as the system *registry*, is maintained by every server in the dissemination network. The registry gathers all information pertaining to server and network operation, from persistent static configuration data to volatile soft state, generated during server operation. The scope of replication is determined by the individual nature of each of the cache items and is encoded in a *radius of interest* tag, ranging from strictly local to fully global. Dedicated event channels are used to refresh registry entries and keep overall consistency, following a policy that prioritizes updates to keep the bandwidth overhead within the set limits.

Primary servers are also required to run a large-scale *discovery service,* whose purpose is to maintain a database with the location and identities of the servers of a given domain. The information collected (and advertised) through this service consists of candidate network entry points, which servers can probe when they want to join (or rejoin) the event dissemination network. Specifically, the discovery service provides the minimum *join information* necessary to establish a basic event channel that allows a server to bootstrap its registry services and properly integrate itself in the network. By denying the join information to a candidate server, the discovery service also incorporates the security policies that control admittance to the network.

4 Programming Model

DEEDS' programming model is divided in two distinct levels, with very different purposes and capabilities. The top one, the user-level, is geared towards the development of event-ware applications. The other, the system-level, focuses on system enhancement and administration. In both cases, the programming interfaces are expressed in the JAVA programming language and assume execution in a standard JAVA environment.

The definition of event used by DEEDS is quite liberal. An event is a small, self-contained notification consisting of a serializable JAVA object paired with a 64-bit integer identifier. Additional information, usually associated with event, such as source identifiers or sequence numbers, is managed automatically by the system runtime and exposed through separate *Receipt* objects.

4.1 Core User-Level Programming

The user-level programming interfaces, naturally, reflect the *publish/subscribe/feedback* paradigm adopted. Typically, applications perform *lookup* operations in an *event channel directory*, with a string name as parameter.

```
EventChannel channel;
EventChannelDirectory directory;

directory = DeedsSystem.getEventChannelDirectory();
channel = directory.lookup( "/channels/apps/App1" ) ;
```

A successful lookup allows the application to *publish* events, *subscribe* the channel, or both.

The publish operation is very straightforward; the application only needs to provide the event data and, in return, accept a receipt object.

```
Receipt r = channel.publish( 0x1L, new myEvent() );
```

To receive events the application must perform a *subscribe* operation. To that end, it supplies the object that will be notified for each event individu-

ally. A 64-bit bit mask is also included to coarsely filter undesired events based on their identifier. An additional parameter identifies the subscription.

```
channel.subscribe( mask, new EventSubscriber() {
    public void notify( Receipt receipt, MarshalledEvent mev ) {
    ...
    }
},...);
```

For performance reasons, the event is kept wrapped in serialized form until accessed; this way expensive un-marshalling operations are avoided whenever the application discards an event judging by its identifier or by the contents of the accompanying receipt.

The DEEDS event dissemination model also lets an application send events back to the source of a previously received event. This is achieved through the *event feedback mechanism*, which, unlike publishing, is strictly a *unicast* operation that targets just one receiver.

The *feedback* operation is similar to a *publish* operation but requires the receipt of the event for which a feedback event is being sent.

```
Receipt r = channel.feedback( receipt, 0x8L, new myEvent2() );
```

The receipt is needed to designate the destination of the feedback event. Receipts cannot be fabricated with the purpose of feedback and regular ones are refused if the original publisher did not subscribe feedback events. The rest of the feedback-related operations have very similar counterpart versions of the interfaces presented here.

To cease receiving events, an application must *unsubscribe* the event channel. The subscription identifier object provided in the subscribe operation is the only argument required.

```
channel.unsubscribe( subscriptionID );
```

Assuming the event channels involved have already been created, these operations are, basically, what is needed to develop event-based applications. Concerning the bulk of an application, the programming style advocated does not involve much more. For this to be realistic, applications must trust the event channels to deliver the QoS they advertise. This is an important point, DEEDS expects applications to be developed with a particular QoS in mind, matching an existing event channel profile that is feasible in the target execution environment. Therefore, event channel creation and deployment are particularly sensitive procedures, depending on sound system administration practices. For this reason, user-level applications are restricted to *cloning* pre-existing (template) event channels.

```
directory.clone("/templates/reliable","/channels/apps/App1",...,...);
```

4.2 User-Level Plug-ins

As stated earlier, the event flow micro-management between applications and the dissemination infrastructure is performed using a specific programming framework - the *routing assistant framework*. In a nutshell, the framework aims to standardize most, if not all, application procedures regarding event filtering, event prioritization, event digestion, event-mailbox management, and other amendments imposed by connectivity limitations. This is achieved by supplying *routing assistant* objects as additional parameters for the *publish* and *subscribe* operations, effectively installing application-defined behavior deep in the event dissemination infrastructure. Due to the sensitivity of these operations, routing assistants must be entirely resolvable from a resource bundle of classes, previously registered in the separate administrative procedure. Privileging the class resource bundle registration procedure ensures some security against malicious routing assistants.

Due to space restrictions, we cannot detail the application routing assistant programming interfaces. Nevertheless, a rough description of their makeup and expected behavior is deemed necessary.

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 decision is to control which events enter the queue, by discarding unwanted events, as they arrive, based on the their receipts and 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 the information, route, delay or discard the event. The routing assistant is also allowed to inject or replace events in the queue. 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.

4.3 System-level Plug-ins

New classes of event channels are added using special plug-ins, known as *system routing assistants,* in a two two-step procedure. One deals with the aspects of the actual programming of the routing assistant object. The other, equally important, involves the documentation of the specifications of the new channel. A precise description of the event channel is important because it is intended to serve two purposes. First, it exposes the *quality of service* that user-level programmers will use as reference. Second, it enumerates the event channel's execution requirements, which serve as guidelines to system administrators when deploying the event channel.

The base programming interfaces of a system routing assistant are rather simple, even though an event channel with an elaborate quality of service will very likely be a complex piece of software. Specifically, the system only expects the routing assistant to be able to dispatch the events it presents it to. Two separate streams of events are involved, a multi-point stream produced by publish operations and an optional unicast stream of feedback events.

```
public interface SystemRoutingAssistant{

   public GUID getChannel();

   public boolean isUnicastRouter();

   public void mroute( EventEnvelope ee ) throws Exception;

   public void uroute( EventEnvelope ee ) throws Exception;
}
```

Events received from remote servers are decoded into *EventEnvelope* objects and then passed to the appropriate routing assistant for further processing. Non-standard envelopes (custom message types) are supported by leaving the interpretation of the remaining envelope data to the system routing assistant.

To ease their development and capitalize on already available programming resources, routing assistants can also rely on the system object registry to gather information or to obtain references to external "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.

```
Container c ;
c = (Container)Registry.getValue("/Containers/Transports");
c.addContainerListener( new ContainerListener {
    public void handleContainerChanges( Container c ) {
    ...
    }
});
```

This scheme allows system routing assistants to synchronize their state (a routing table, for example) in reaction to changes in the containers they monitor. Since the information made available through this process is not limited in any way and can be extended at any time, we find these simple programming interfaces a convenient way of adapting the overall system to the needs of present and future event channels developers. Still, active network issues such as security and resource consumption[1] have been clearly downplayed but we believe future work in the area will not be hindered by the present model decisions.

5 Related Work

Considerable work in information dissemination systems has been produced in recent years, originating from both academic sources and the software industry. The broad scope of available solutions prohibits an exhaustive discussion, so we focus our attention on representative platforms that pursue similar objectives to our own interests.

TIB/Rendezvous[4] is a messaging middleware that follows a subject-based publish/subscribe model over a hierarchical namespace. It offers a fixed set of QoS guaranties, such as "reliable delivery", peer-to-peer "certified delivery" and centralized "transactionally guarantied delivery". Its industrial and closed nature does not compare well in terms of extensibility.

Smartsockets[6] and iBus[2] are two other event-channel based industrial solutions, which have evolved into supporting Sun's JMS standardized but cumbersome messaging API [7]. iBus is a primarily peer-to-peer solution but a recent update introduced some protocol heterogeneity support through the use of bridging/tunneling. In any case, an event channel's QoS is tied to the protocol stack in use. Smartsockets promises scalability using a mesh of servers but unlike DEEDS, the event routers are not programmable and implement a rigid routing algorithm.

Elvin[9], Siena[8], Gryphon[10] are examples of content-based subscription solutions. These approaches require the use of structured events, whose content must be interpreted by the dissemination infrastructure to select the interested receivers. In these systems, event consumers subscribe from a global pool of events by providing elaborate filter expressions, which must be evaluated against incoming events. Elvin is, currently, a non-scalable, centralized solution but does offer support for disconnection. Both Siena and Gryphon address scalability issues by migrating subscription expressions

over decentralized multi-server architectures. Still, it is not clear how these authors address the implications of the heavy processing requirements associated with massive filter evaluation. DEEDS also provides content-based subscriptions, albeit very simple ones, but avoids heavy processing loads using, instead, binary masks over the integer event identifier. Masks can be efficiently merged and propagated but are much coarser and do require programming discipline.

INS[14] is a resource and service discovery system that integrates name resolution with message routing in a dynamic and mobile 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.

Salamander[11] is a wide-area data dissemination platform, geared towards broadcasting of scientific data-streams by a tree of servers. Application plug-ins can be added along the distribution path to allow data degradation and filtering.

6 Concluding Remarks

We will end this discussion by mentioning the present status of this work and some of the directions that will guide our future work.

Current efforts of the team are focused on the completion of a prototype that will, hopefully, validate and demonstrate the guiding principles of the platform. Although preliminary results are encouraging, we expect the insights yet to be gained from a full-featured prototype and the modeling of sample, real-life applications will allow us to refine the pursued concepts and address weaker spots, such as security.

7 References

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

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

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

[5] 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

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

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

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.