EVENT DISSEMINATION SERVICE FOR PERVASIVE COMPUTING

Sasu Tarkoma[1]

Abstract

Event-based computing is a generic enabler for the next generation mobile services and applications that need to meet user requirements irrespective of time and location. The event paradigm and publish/subscribe systems allow clients to asynchronously receive information that matches their interests. We outline an event architecture for mobile computing that addresses two key requirements: terminal mobility and user mobility. The system consists of access servers, event channels and a mechanism for locating event channels. The architecture uses filter covering and merging for supporting high accuracy in event delivery, reducing communication cost, and improving event processing on terminals and servers. Experimental results based on the merging system are also examined.

1. Introduction

Pervasive computing creates new possibilities for applications and services; however, it also presents new requirements for software that need to be taken into account in applications and in the service infrastructure. In order to support the development and deployment of intelligent applications a number of fundamental enabling middleware services are needed [4]. Two important services are event monitoring and event notification, which are vital for supporting adaptation in applications. Event monitoring and notification are used to realize a number of pervasive applications, such as smart rooms, sensor networks, presence applications, and device tracking and management.

Most research on event systems has focused on event dissemination in the fixed-network, where clients are usually stationary and have reliable, low-latency, and high bandwidth communication links. Recently, mobility and wireless communication have been an active topic in many research projects working with event systems, such as Siena [1], JEcho [2], and Rebeca [6]. These systems have focused on fixed-network infrastructure supporting mobile entities. Ad hoc environments are an emerging research area.

Pervasive computing creates new requirements for event systems, such as mobility support, context-awareness, interoperability, and support for heterogeneous devices. The proposed event framework addresses these issues by providing a Web Services-based fixed-network event routing infrastructure that supports user and terminal mobility. User mobility occurs when a user becomes disconnected or changes the

[1] Helsinki Institute for Information Technology, sasu.tarkoma@hiit.fi

device. Terminal mobility occurs when a terminal moves to a new location and connects to a new access point. Mobility transparency is a key requirement for the system, and the middleware system should hide the complexity of subscription management due to mobility. The hypothesis is that efficient mobility support may be provided using an explicit-join rendezvous scheme [5]. The proposed scheme uses event channels as meeting points in the server cloud. In addition, covering relations and filter merging are central in minimizing the size of routing tables at the event routers or servers.

The proposed event framework differs from existing work in several ways. The handover protocol uses subscription covering to prevent unnecessary update messages. The filtering system is based on disjunctive attribute filters, which are more expressive than single predicate or conjunctive attribute filters. Features such as client-side filtering and sessions are not supported in many systems. Client-side filtering is important for pervasive environments in order to prevent unnecessary signaling over wireless networks. The system groups client subscriptions into sessions, which are manageable units and may be shared between applications.

Features such as mobility support and merging are not visible to pervasive applications, but they are used in making the system more efficient by delivering information where it is needed and doing it using the highest possible precision. API features such as support for client-side filtering, and session sharing are interesting for pervasive applications. For example: smart sensors may reduce unnecessary network signaling by retrieving or receiving merged sets of filters and sending only matching events to the network.

The rest of the paper is organized as follows. In Section 2 we present the system architecture. In Section 3 we examine filter merging and in Section 4 we present the current status and future work.

2. System Architecture

The architecture is based on the notion of a domain of event servers that provide the event service for a number of wireless and mobile clients and for other entities that use events. The architectural components of the system are:

- event channels (EC), which are rendezvous points for subscribers and publishers,
- resolution servers (RS), which are responsible for the event channels,
- sessions that consist of zero or more subscriptions and buffered notifications,
- access servers (AS), which maintain connections with client systems, and
- event domains are collections of access and resolution servers.

The architecture aims to meet the requirements of mobile users by supporting bounded delivery times and disconnected operation. User mobility is supported by the session concept that stores undelivered notifications at access servers. A handover protocol is used to transfer sessions between access servers, which facilitates client

mobility and may also be used in load balancing of sessions. Access servers associate notifications with client subscriptions so that client systems do not necessarily have to filter incoming notifications. This is especially useful for low-end client systems, such as mobile phones and lightweight PDAs. Resolution servers are responsible for event channels that contain a more generic set of filters, and the access servers maintain the full set of filters. Server-to-server communication may take an advantage of IP-multicast, but if it is not available the application level multicast is realized using point-to-point messaging.

The motivation for the work stems from the high cost of mobility in a hop-by-hop routed event infrastructure. The cost for mobility support in terms of exchanged messages is high, because the source and target servers need to synchronize and update the event routing topology, which may be arbitrary [5]. When non-covered or merged subscriptions are propagated servers can no longer identify the source server of a subscription, because this information is lost in the covering or merging process, and servers need to use flooding to find the route to the source server [6].

The system supports two notification models: ordered delivery with logging, and decentralized delivery with causal ordering. In ordered delivery events are routed through the event channel, which provides total ordering of events within an event domain and event history and logging features. In decentralized delivery the event channel is used to synchronize the subscription status of the access servers and they forward events to other access servers. Within a single event domain, filtering is done in three phases: first on client systems, which is not mandatory, then at the resolution servers (event channels) or access servers (decentralized delivery), and in the last phase on the destination access servers.

The decentralized notification mechanism does not rely on a centralized dispatcher, which makes it less prone to problems related with scalability and network partitions. This approach synchronizes subscriptions by using a proxy channel, which resides on the access servers. The event channel has the merged subscriptions for the access servers, and upon subscriptions/unsubscriptions it updates the proxy channels. The event channel has a global high-level view on the subscriptions and advertisements in the channel, and may further optimize and merge the set of filters. For example using advertisement semantics subscriptions need to be sent only to those servers that have matching advertisements. Access servers forward events to other access servers based on the proxy channel's routing table. The proxy channel contains the filters for other access servers, which are needed in order to make the forwarding decision.

Event channels partition the subscription space into orthogonal or near orthogonal sets of subscriptions. Event channels are located using a built-in directory service, which maps event types to channels. A load balancer component is used to relocate channels and assign channels to resolution servers. A key property of the system is that the lookup cost for an event channel is constant or near constant. This means that subscription management operations may be performed efficiently, and the number of hops required by the operations is bounded both within an event domain and between event domains. Event channels may also be connected on a higher level, for example to form hierarchies, which may affect the publication cost of events, but this does not affect the routing table update cost for a single channel. Access servers update event channels only when a subscription is not already covered by an existing subscription, in addition the event channel update protocol uses either perfect or imperfect merging.

Filter merging removes unnecessary redundancy, and reduces the routing table sizes and processing overhead of event channels. Access servers join the event domain by sending a subscription message to an event channel. An access server may leave the domain by unsubscribing all subscriptions.

2.1. Federation

Federation of event domains may be accomplished by connecting event channels of the same type in different domains. A basic assumption is that in order for an event to be delivered to another domain an event channel corresponding to the event type must exist in the foreign domain. There are several ways to implement communication between event channels. The basic method is to multicast updates between all channels. Another method is to map or hash the event channel name over a set of backbone servers to find a rendezvous server. Each event channel of the same type in different domains updates the merged set of subscriptions to this rendezvous server. The server delivers this information to other channels, and therefore the channels have the knowledge of what events should be forwarded and where. Federated resolution servers may also calculate a shortest-distance spanning tree for event channels of the same type. In this case the event channels would form a more traditional routed event infrastructure with a higher cost for mobility.

2.2. Handover Procedure for Client Mobility

The mobility protocol supports both client-initiated and server-initiated handovers. There are two variants of the protocol: handover within a domain and handover between domains. With client-initiated terminal mobility the protocol proceeds in both cases as follows: the target server of mobility initiates the protocol and contacts the source server and client subscriptions are sent to the target access server. If the target server has already subscribed a covering set of subscriptions the event channels are not updated unless the source server has no subscribers for the relocated subscriptions. Buffering needs to be done both at the source server and at the target server in order to avoid false negatives. Handover between domains differs from domain specific operation, because relevant event channels in the two domains, source and target, may need to be updated. In the final phase of the handover the client session (containing buffered notifications) is moved from the source to the target server and duplicates are removed. Initial results with the handover protocol within an event domain indicate that the handover procedure may benefit from support for filter covering in scenarios where subscriptions are saturated and subscribed by other clients both at the source and target access servers.

3. Filter Merging

We define a notification to be a set of 3-tuples: $N = \{t_1,t_2,..,t_m\}$, each tuple is defined by <name, type, value>. The set of elementary types is defined as: T = {String, Integer, Double, Boolean, Date}. A filter is a set of attribute filters, which are 3-tuples defined by <name, type, filter clause>. Each attribute filter must match a tuple in a notification for the filter to match a notification. The filter clause is a constraint in the disjunctive normal form constructed using elementary atomic predicates. Our system supports basic comparison and matching predicates for the different types. Notifications and filters are represented using XML.

We have developed a merging framework that uses filter covering and merging rules for predicates to remove redundancy. The covering algorithm takes into account also semi-structured events by supporting quantification over lists. The framework supports perfect and imperfect merging. Both merging approaches use the same principle for filters that have the same structure[2]: the conjuncts of each mergeable attribute filter are merged either using merging rules or combined using disjunctions if they are not covered by other conjuncts. A conjunct that is covered by another conjunct may be removed. The algorithms have polynomial time complexity.

Imperfect merging simply fuses the set of filters that have the same structure. Perfect merging differs from imperfect merging by a stricter mergeability condition: a filter F_1 may be merged with another filter F_2 only if it has at least n-1 identical attribute filters, where n is the number of attribute filters in F_1 and F_2 [3]. The disjunctive formulas guarantee that the merging of the distinctive attribute filter can be performed. Perfectly merged filters have a precision of one, and imperfectly merged filters have a precision in the range [0,1].

Experimentation with the framework indicates that covering and merging may be used to reduce the size of propagated subscriptions, matching time, and signaling overhead. Merging performance depends on the nature of the constraints, their distributions and the structure of filters. Figure 1 gives a summary of the initial results for perfect and imperfect merging with simple integer, double, boolean and string constraints generated from 100 random strings and a number range of 1000. We used 10 schemas, 200 subscriptions, and 20 replications for these results using randomly generated subscriptions and randomly generated schemas using the uniform distribution. Perfect merging is especially useful for filters with a few attribute filters. Imperfect merging gives good performance even when the number of attribute filters grows, and fuses the filters given that they have the same structure with a cost in a number of false positives. For 200 subscriptions and 2 tuples the precision was 100%, for 3 tuples it was 92%-99%, and for 4 tuples it was 60%-82%. The benchmark is based on a number of predefined schemas. If the filter structure is random the merging schemes may not be able to perform merging.

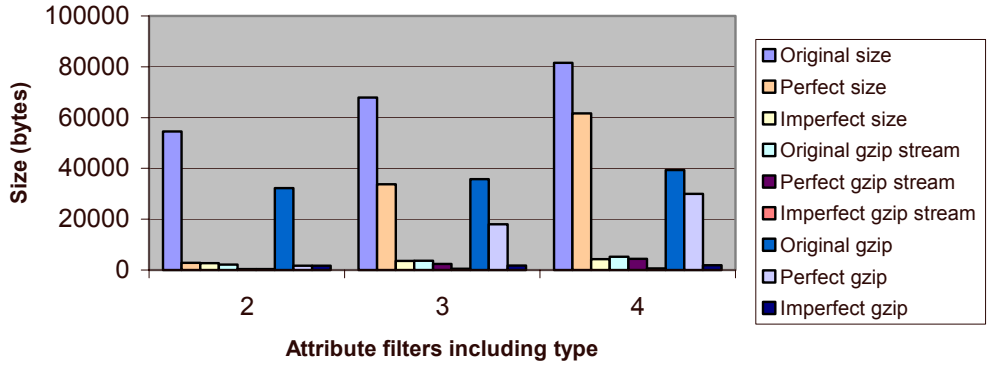

Figure 1. Initial results using perfect and imperfect merging using 2,3, and 4 tuples

[2] This means that the names and types of attribute filters are identical, but the constraints may differ.

Results with gzip compression for both filters and filter streams are also presented in Figure 1. The results show that gzip gives good compression ratio for both merged and unmerged filter streams. On the other hand, filter specific gzip compression is relatively inefficient when compared with streaming compression. The major benefit of covering and merging is not only shorter messages, but also more compact routing tables, and faster event matching and processing.

4. Current Status and Future Work

A prototype implementation of the proposed system has been developed using the Java language, building on existing technologies such as SOAP and Apache Axis. We have also developed a lightweight version of the client-side API, which features pub/sub and session management operations for J2ME MIDP[3]. The implementation has been used to create two demonstration applications: a mobile presence application and a context-sensitive ticker for mobile phones that allows transferring the end-point of a session to a dormant instance of the application. The presence application uses the event system to disseminate changes in users' presence information.

Currently, load balancing and federation support are under development. The framework supports scalability to a number of event servers, but wide-area scalability is an open issue. Dynamic filtering and merging in both client-server and server-server environments seem to be promising research topics—how to balance between the precision and size of filter sets using different algorithms. These mechanisms may also be used in ad hoc environments on devices that have enough processing power. The proposed event architecture is envisaged to be a supporting layer for a compound event detection system that supports the detection of complex event sequences in time. We also plan to compare different mobility models and load balancing techniques for session/channel relocation using simulation and the prototype implementation.

5. References

[1] CAPORUSCIO, M., INVERARDI, P., PELLICCIONE, P, Formal Analysis of Clients Mobility in the Siena Publish/Subscribe Middleware, Technical Report, Department of Computer Science, University of Colorado, October 2002.

[2] CHEN, Y., SCHWAN, K., ZHOU, D., Opportunistic Channels: Mobility-Aware Event Delivery, Middleware 2003, LNCS Vol. 2672, pp. 182-201, Springer-Verlag, 2003.

[3] MÜHL, G., Generic Constraints for Content-based Publish/Subscribe, in: C. Batini et al. (Eds.), CooPIS 2001, LNCS 2172, pp. 211-225, Springer-Verlag, 2001.

[4] RAATIKAINEN, K., CHRISTENSEN, H., NAKAJIMA, T., Application Requirements for Middleware for Mobile and Pervasive Systems, ACM SIGMOBILE Mobile Computing and Communications Review, Volume 6, Issue 4 (October 2002).

[5] TARKOMA, S., KANGASHARJU, J., RAATIKAINEN, K., Client Mobility in Rendezvous-Notify, 2nd Intl. Workshop on Distributed Event-Based Systems (DEBS'03).

[6] ZEIDLER, A., FIEGE, L., Mobility Support with Rebeca, Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS) Workshop on Mobile Computing Middleware.

[3] Java 2 Micro Edition, Mobile Information Device Profile