Higher-Level Constructs in the READY Event Notification System

Robert E. Gruber[*], **Balachander Krishnamurthy** and **Euthimios Panagos**
{gruber, bala, thimios}@research.att.com
AT&T Labs — Research

1 Introduction

There is growing interest in the use of general-purpose event notification services as "middleware" for gluing together independently-developed distributed applications. This approach was first discussed by Oki *et.al.*in 1993 [10], where an early version of the (now) TIBCO Information Bus [13] was described. Since then, many other commercial products have been developed that accept events from event suppliers (publishers) and deliver them to event consumers (subscribers). Publish-subscribe mechanisms have also been retro-fitted to existing middleware products, including persistent message queue products and TP monitors. There is also standards work in this area: the OMG adopted a standard CORBA Event Service [11], and will soon adopt a standard for a Notification Service which is a feature-enriched version of the Event Service.

Despite all of these products and the standardization efforts, and despite the fact that event-driven computation is not a new idea, there has been relatively little work on high-level constructs for event services. The most basic event service has four kinds of entities (supplier, consumer, event, event service) and three basic functions (supply an event to the service, register interest in a kind of event[1], unregister interest). Existing commercial event services do not provide event models that are much richer than this most basic service. For example, there are typically no constructs that enable operations over multiple entities.

This paper describes high-level constructs (operating over multiple events, multiple consumers, *etc.*) provided by READY, an event notification system being developed at AT&T Labs [5]. Entities in the READY model include: consumer specifications that match over both single and compound event patterns; communication sessions that manage quality of service (QoS) and ordering properties for event delivery; grouping constructs for both specifications and sessions; event domains and boundary routers that bound the scope of event distribution and control the mapping of events across domains. The result is a more powerful, more flexible event service.

The rest of the paper is organized as follows. The first two sections describe constructs that (in combination) are only available in prototype event systems: extensible event types, simple and compound matching specifications, and event aggregation. The next three sections discuss constructs that are unique to READY, including grouping constructs and constructs for managing boundaries between event domains. A final section discusses the current status of READY as well as some general issues raised by this work.

We are currently exploring the merits of READY's high-level constructs by using READY as a core component of a unified messaging system [2]. Examples presented here are taken from this domain.

2 Event Types

This section describes READY event types and event encoding issues, where the two key properties are extensibility and self-describing encoding. Extensibility is accomplished in two ways: subtypes describe extensions that are used uniformly across many events, while arbitrary additional fields can be added to individual event instances.

In READY an event type definition specifies a set of required and optional fields, where each field contains a field name, type identifier, and value. The field type specifies the encoding of the value. Field types include scalar values, such as `string`, `real`, `integer`, set types, and event types. Event fields are self-describing: the name and type of a field can be determined from the field itself. READY types can have subtypes. A subtype declaration simply adds additional required or optional field specifications to those of its parent type.

If event type E has required field R of type T, all E events must have an R field with type T. If event type E has an optional field O of type T, an E event does not need to have an O field; however, if O *does* appear in an E event, it must be of type T. Any event encoding, regardless of its type, can

*Contact Author: Robert Gruber, 180 Park Avenue, Room B285, Florham Park, NJ 07932, Tel: +1 973 360-8667

[1] Registration for interest in events typically involves specifying a subject or type, perhaps with a wildcard; some services allow more complex tests over the values of several event fields.

have *extra fields*. Access to such fields always requires run-time type checking.

An event encoding must include a field with name EType whose value identifies the type of the event. READY can be configured such that it assumes a correct encoding of the required fields for this event type immediately follows the EType field. It can also be configured to parse and perform sanity checks on each field. Event type identifiers are conceptually encoded as strings (the encoding can be optimized).

Compound type identifiers such as Mail.Msg.New can be used. For our messaging framework, we use the convention that the structure of type identifiers matches the type hierarchy. For example, Mail.Msg is a subtype of type Mail, and Mail.Msg.New is a subtype of Mail.Msg. This convention can make it easier to integrate a READY-based system with publish-subscribe systems: simple matching expressions over a compound EType mimic the matching over compound subject names found in many publish-subscribe systems.

3 Specifications

A READY specification consists of an optional label, an event pattern and an optional action, written in the following syntax:

label: [*event_pattern*] **do** {*action*}

A simple specification uses a pattern that matches single events. Simple specifications are also called event filters. A compound specification uses a pattern that describes an interesting combination of events. A pattern often corresponds to a high-level event, *i.e.,* an event that subsumes the set of lower-level events that matched the pattern. Thus, compound matching adds an abstraction mechanism to the event service.

The action part may specify local actions, notification (delivery of events to consumers), or both. Local actions are actions that the event service can perform "internally," without notifying a consumer. For example, a local action can create a new event and supply it to the event service. Other local actions include: adding and removing specifications; suspending or resuming specifications or specification groups (discussed below). The default action is notification of the event or events that matched the event pattern.

READY's matching language is based on the language used for Yeast [9]. An example of a compound specification from our unified messaging framework:

```
% A match occurs when the same user sends two
% messages in 30 secs and the subject of the
% 2nd message contains the subject of the 1st

SameSubj: [E1:Mail.Msg.New, E2:Mail.Msg.New;
    (E1, E2) within 30s;
    E1.From == E2.From &&
    E2.Subj ~ "*"+E1.Subj+"*"]
```

Once you introduce compound matching, it is useful to also introduce *aggregation*. As part of the specification's actions, it should be possible to combine the set of events that matched the patter, resulting in one or more aggregate events. As with the original matching events, the aggregate events can be supplied to the event service and/or consumers can be notified.

When considering support for aggregation for the READY language, we made the follow observation. An event is a set of named fields; the set of events that match a compound specification is a set of sets of named fields; aggregation involves flattening the set of sets back to a simple set of named fields: fields need to be removed, kept as-is, or combined to produce new fields. Since two events can have a field with the same name, there must be a way to rename a field at the time flattening occurs, *e.g.,* so that both fields can be retained as-is. It is not hard to invent set operators that select or remove named fields, rename fields, and so forth; this is what we did. (An escape hatch to a general-purpose language is also provided.) Here is an example of aggregation that uses some of these operators:

```
% E3 = fields named a and b from E1 and
% fields named a and b from E2, renaming
% E2's a and b to x and y.

E3 = new Ex.Type {E1|(a,b) + E2|(a->x,b->y)}
```

To the best of our knowledge, only READY and two other research-prototype event services have compound matching and aggregation. While their language constructs differ, the matching and aggregation capabilities of READY and the event service developed by the OPERA group at the University of Cambridge [1, 7] have similar power. A more limited form of compound matching is provided by the TAO real-time event service [6]: repetitions of the same kind of event can be detected.

4 Specification Groups

Specifications can be grouped to form specification groups, and operations normally applied to a single specification can be applied to groups. For example, consider the *suspend* and *resume* operators, which are used to disable and enable matching over a specification. By providing groups and these operators, READY enables rapid changes to the set of active specifications. One operation with a single argument (the name of a group) has a large impact. If groups and the suspend semantics were not available, a consumer would need to use many individual remove or add operations to suspend or resume a large set of specifications.

5 Sessions

A client of READY interacts with READY using three kinds of sessions: *admin sessions, supplier sessions,* and *consumer sessions*. Admin sessions have methods to create supplier and consumer sessions, and to support all READY operations that are not specific to a particular supplier or consumer session.

Supplier sessions are used to supply events to the service. Before a supplier session S can be used for this purpose, the supplier must register a description of the kinds of events that it will supply using S. READY is free to discard events supplied to session S that do not fit the description. The normal case is to register the set of event types that can be supplied; event types can be added or removed over time. Additional constraints can be added, *e.g.,* , a supplier could indicate that the priority field for all events it supplies will never be greater than 10.

Consumer sessions are used for registering specifications and for setting delivery properties for notifications from these specifications. Important delivery properties include quality of service (QoS) and ordering constraints. Two examples: best-effort delivery ordered by timestamp; reliable delivery ordered by priority. (Events that reach consumer within some time window are ordered with respect to each other according to ordering constraints.)

A consumer session can be *disconnected* from the event service. Conceptually, the service is responsible for retaining notifications until the session is *reconnected*, at which point the notifications are delivered. A combination of factors, such as the QoS specification for the consumer session and time-to-live fields in events, determines how long the service actually retains notifications for a disconnected session.

6 Session Groups

A consumer or supplier session can join consumer groups or supplier groups, respectively. The operations described above for individual consumer or supplier sessions can be applied to groups instead. When a specification (or specification group) is associated with a consumer group, notifications from this specification go to all sessions in the consumer group. The delivery properties of all members of a session can be set via the group. Although not (yet) part of the READY design, it seems clear that concepts from group communication systems [12, 3] can be applied to session groups. For example, one could optionally specify that all consumer sessions in a session group must receive notifications in a consistent order.

7 Event Domain Boundaries

There must be a way to manage interactions between different event domains (different administrative domains, different application frameworks, and so on). In the normal case, the network infrastructure of each domain is disjoint. A boundary event router can sit at a junction between two or more named domains and manage the flow of events across the boundary. The types that are used, the conventions adopted for use of particular fields within an event type, *etc.*, can differ across domains, thus complex mapping capabilities are needed at this boundary.

Oki *et.al.* [10] observe that boundary routers can perform such mappings by having the router consume an event at one side of the boundary and supply another event on the other side. Mappings occur outside of the event service (and would be coded in a general purpose programming language). We chose to extend the power of our specification language to support boundary mapping specifications. In addition to providing a language for writing mappings, this approach provides support for dynamic control over the boundary router: as with normal specifications, it is easy to add, remove, suspend, or resume mapping specifications (and groups of mapping specifications).

Two extensions to the specification language are required. First, event types in mapping specifications must be augmented with domain names. Second, for an aggregation function or other local action that supplies a new event, a set of one or more domains for the supplied event must be specified. The common case is to specify one-to-one mappings, but compound matching and event aggregation can be used for more complex boundary cases.

An advanced feature of boundary routers is the ability to restrict network bandwidth used up by mapped events. With this feature, boundary routers turn out to be especially useful for carrying out a safe trial of a new service that relies on events generated by existing "released" services. A boundary mapping from trial event to released event can perform sanity checks to ensure that events will not bring down the released services. Further, bandwidth restrictions on mapped trial events shield the released services from serious performance degradation due, *e.g.,* to a bug in a trial service that generates too many events.

8 Discussion

Compared to current event service products, we present an event service with a richer set of entity kinds (including grouping entities) and a more powerful set of operations over these entities. Not all event-driven applications can take advantage of this richer event model, but there are many application, from many application domains, that can. Our goal is to make this model available in a general-purpose event notification service.

Event services are an important component of many application frameworks [4, 8]. Frameworks specify a consensus[2] on common component types, event types, and behavior patterns for the particular application domain (how do components invoke each other directly, when do components supply key events, *etc.*). By providing high-level constructs that describe expected patterns of behavior, more of a framework's specification can be captured and shared by applications. (Compound event specifications are a step in the right direction, but much more is needed.)

A discussion of implementation issues requires another paper, but it is important to note here that introducing high-level constructs does not imply that the service will be inefficient. In fact, the opposite is true: high-level declarations enable optimizations that are not otherwise available. Here are several examples. High-level supplier descriptions, combined with high-level consumer specifications, enable low-level event-routing optimizations. For specifications, common sub-expression elimination can be performed across all registered matching specifications, while common filters (capturing what is *not* interesting to any consumer) can be moved "upstream" towards suppliers.

9 Current Status

We are in the process of implementing READY on top of a CORBA Event Service and plan to implement on top of a CORBA Notification Service in the future. A CORBA service is just one choice for the underlying event routing layer; our API hides details of the underlying system. Since no single event service implementation will support (or have the best implementation of) the full range of QoS and event ordering properties we would like to support, we also plan to explore the use of multiple underlying systems.

References

[1] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using events to build distributed applications. In *Second International Workshop on Services in Distributed and Networked Environments*, pages 148–155. IEEE Computer Society Technical Committee on Distributed Processing, IEEE Computer Society Press, June 1995.

[2] A. Biliris, R. Gruber, G. Hjalmtysson, H.V. Jagadish, M.A. Jones, M.F. McGroary, E. Panagos, M. Rabinovich, A.W. Robinson, S. Spear, D. Srivastave, and D. Vista. Hermod: A distributed database infrastructure for electronic messaging. Submitted for publication, 1998.

[3] D. Dolev and D. Malki. The transis approach to high availability cluster communication. *CACM*, 39(4), April 1996.

[4] M. Fayad and D. Schmidt. Object-oriented application frameworks. *CACM*, 40(10), October 1997.

[5] R. E. Gruber, B. Krishnamurthy, and E. Panagos. Ready: A notification service for atlas. Technical report, AT&T Labs - Research, 180 Park Avenue, Florham Park, NJ 07932, September 1997.

[6] T.H. Harrison, D.L. Levine, and D.C. Schmidt. The design and performance of a real-time corba object event service. In *Proceeding of the OOPSLA '97 Conference, Atlanta, Georgia*, October 1997.

[7] R. Hayton. Oasis: An open architecture for secure interworking services. Technical report, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, March 1996.

[8] R. Johnson. Frameworks = (components + patterns). *CACM*, 40(10), October 1997.

[9] B. Krishnamurthy and D. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10), October 1995.

[10] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus — an architecture for extensible distributed systems. In *14th ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, December 1993.

[11] Object Management Group (OMG). Event service specification. ftp://www.omg.org/pub/docs/formal/97-07-13.pdf.

[12] R. Van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *CACM*, 39(4), April 1996.

[13] Tibco. Tib/rendezvous. http://www.rv.tibco.com/rvwhitepaper.html.

[2] Since this consensus will evolve over time, it is important to use an event service that supports event type evolution.