The Architecture of the READY Event Notification Service

R. E. Gruber, B. Krishnamurthy and E. Panagos
AT&T Labs – Research
{*gruber, bala, thimios*}*@research.att.com*

Abstract

In this paper, we present the architecture and implementation of READY, an event notification service that provides efficient, decoupled, asynchronous event notifications. READY supports consumer specifications that match over both single and compound event patterns, communication sessions that manage quality of service (QoS) for event delivery, grouping constructs for sessions and specifications, event zones and boundary routers that bound the scope of event distribution and control the mapping of events across zones.

1 Introduction

An event notification service is a key enabling technology for building new event-based services on distributed platforms; it provides the middleware for gluing together independently-developed distributed applications. Though event-driven computation is a relatively old idea, there has been little work on *high-level* constructs for event services. The most basic event service has four kinds of entities (supplier, consumer, event, and event service) and three basic functions (supply an event to the service, register interest in a kind of event, and 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 (events, consumers, *etc.*).

Event services are an important component of many application frameworks [3, 8]. Frameworks specify a consensus on common component types, event types and behavior patterns for a particular application domain (how do components invoke each other directly, when do components supply key events, *etc.*). Since consensus may evolve over time, it is important to use an event service that supports event type evolution. By providing high-level constructs that describe expected patterns of behavior, more of a framework's specification can be captured and shared by applications.

This paper describes our work on READY, an event notification system being developed at AT&T Labs – Research [4, 5]. READY supports 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 zones and boundary routers that bound the scope of event distribution and control the mapping of events across zones, among others. The overall architecture of READY and the various components used by READY servers and clients are also presented.

The remainder of the paper is organized as follows. Section 2 presents the READY high-level constructs. Section 3 describes the architecture of READY. Section 4 offers implementation details. Section 5 discusses related work and, finally, Section 6 concludes our presentation.

2 High-level event constructs

Here, we summarize the READY event constructs; further details can be found in [5]. READY clients interact with READY using *admin*, *supplier* and *consumer* sessions. Admin sessions are used for creating and destroying supplier and consumer sessions and session groups and for other administrative operations. Supplier sessions are used to supply events to the service. A supplier must declare the kinds of events that it will supply to a supplier session. Consumer sessions are used to register *specifications*, which describe event patterns and actions to take when matches are found for patterns – the most common action, `notify`, causes the consumer session to deliver a notification with the matched event(s).

One can *suspend* and *resume* a specification, enabling and disabling matching, respectively. One can *connect* and *disconnect* a consumer session, enabling or disabling delivery of notifications for that session. Specifications can be grouped into a *specgroup*, enabling efficient suspend and resume of a large number of specifications. Consumer sessions can be grouped into a *congroup*, enabling sharing

of specifications and specgroups and uniform control over their QoS and delivery properties. When a `notify` action occurs for a shared specification, all consumer sessions deliver the notification. A specification or specgroup is always associated with a single consumer session or congroup, and using suspend/resume on a session or congroup suspends/resumes all associated specifications.

2.1 Event structures and types

Generic matching algorithms for events of arbitrary structure can be costly and, therefore, structured events are needed to optimize event matching. READY events follow the structured event format specified in OMG's Notification Service [13]. Events consist of *header, filterable body,* and *remainder of body*. The header consists of *event type*, and *event name*, and an optional variable part that may contain QoS-specific attributes, timestamps, access rights, and so on. Event types contain a *domain name* (`D`) and a *type name* (`T`). The domain name is used to distinguish type names in administrative, application, and vertical domains. The filterable body part contains attribute-value pairs, while the remainder of the body is an opaque value.

READY event type definitions specify required and optional fields for the optional header part and the filterable body. In addition, event types (i.e., type names) can have subtypes. A subtype declaration simply adds additional required or optional field specifications to those of its parent type. Compound type names like `Mail.Msg.New` can be used with the convention that the structure of the type corresponds to some type hierarchy – this simplifies the integration with publish-subscribe systems: simple matching expressions over compound types mimic the matching over compound subject names found in these systems.

2.2 Specifications

A *specification* is composed of event *matching expressions* together with *actions* that are triggered by successful matches. Matching expressions bind event variables to events that match a given event pattern. A simple matching expression is a pattern that matches a single event; `Ev1: MyAlarm | $Priority > 5` binds `Ev1` to the first `MyAlarm` event with `Priority` greater than `5`. The general form for an expression that matches a single event (named filter) is:

$$event\_var : event\_type \mid expression$$

The `expression` is any legal expression from the filter grammar specified for the OMG Notification Service [13]. The `$` symbol stands for the event being matched against, and component expressions such as `$Priority` evaluate to values of components of this event. Component expressions that select sub-parts of top-level event fields can also be used (see [13] for details).

Simple matching expressions can be combined to form compound matching expressions that describe a combination of events using the operators '`&&`' (and), '`||`' (or), '`;`' (then), and `butnot`. The '`&&`', '`||`', and `butnot` operators begin matching both LHS and RHS at the same time, where '`&&`' matches successfully once both sides match, '`||`' matches successfully once either side matches, and `butnot` matches successfully when the LHS matches first. In contrast, the sequencing (`;`) operator does not attempt to match the RHS until the LHS has matched.

Another form of compound matching is to specify a sequence of events which all match the same "element pattern" (a matching expression used for each element of the sequence) using the following:

$$event\_var[j..k] : event\_type \mid expression$$

The `event_var[j..k]` indicates that the event variable will be bound to a sequence of events, with `j` and `k` being the minimum and maximum number of events in the sequence, respectively. Note that `j` can be zero, indicating no lower bound, and `k` can be an asterisk, indicating no upper bound. As a top-level expression, sequence matching completes when the minimum number of successful element matches occur. When contained within a larger expression, a sequence match is successful once the minimum number of element matches occur, but elements continue to be added to the sequence until either the maximum number of element matches occurs or a containing expression successfully matches.

A *where expression* can be attached to a matching expression to further constraining the set of possible matches. This expression must be a boolean expression over the event variables in the matching expression. This form is useful for expressing inter-event constraints, especially constraints over sets of events. For instance, a *span* constraint can be used to require that the maximum pairwise distance between `Priority` values for the matched events must be less than `5`.

Actions are imperative statements executed at specified points during matching. An assignment action binds an event variable to a newly-created event. This is most often used to perform *aggregation*, where a set of matching events is summarized by a single new event. Actions that operate over event variables include the `notify` action (mentioned above) and the `announce` action, which acts as a supplier of a new event. Normally notifications go to the consumer session or congroup associated with the specification, while announcements are supplied to the local *zone* (discussed below), but these defaults can be overridden. Actions that do not require contacting suppliers or consumers are referred to as *local actions*; these actions

only involve changing the internal state of the READY servers. The above described operations over (groups of) specifications and (groups of) sessions are available as local actions.

2.3 Event zones

Event consumers and suppliers can be partitioned based on logical, administrative, vertical, political, or geographical boundaries. Each such partition constitutes an *event zone*. Typically, all event traffic between suppliers and consumers in a given event zone remains within this zone. Nevertheless, there are cases where the events supplied in a zone may be useful to consumers in other zones and, therefore, routing of event between zones is needed. Inter-zone routing has to support *event translations* (zones can differ in terms of event types, conventions adopted for use of particular fields within events, *etc.*) and address any information flow constraints that may exist across zone boundaries.

READY solves inter-zone event routing with *boundary routers*, specialized READY servers that connect READY zones. Event routing and translation across a boundary is performed by *mapping specifications* that match against events in one zone and announce, possibly translated, events in another zone. Although one could write a static application that registers with two READY zones, consuming events in one and supplying events in the other, our approach is more flexible: as with other specifications, READY supports dynamic addition, removal, suspension, and resumption of (groups of) mapping specifications. Finally, a further refinement of boundary routers is the ability to restrict event throughput used by inter-zone mapping specifications.

3 The architecture of READY

Each instantiation of READY is implemented using one or more event zones, and each event zone may contain multiple READY servers. Boundary routers link together two or more event zones, and each event zone may use several routers. Boundary routers can be linked together in a hierarchical or a peer-to-peer topology, depending on the characteristics of the applications build on top of a given READY instance.

Figure 1 shows the architecture of a READY server. Each server consists of several components. These components are not server-specific; some or all them might also be used in the implementations of the various session objects created on client machines, or even as stand-alone components used by consumers for their own purposes after they receive event notifications. In the remainder of the section, we describe in detail some of these components.

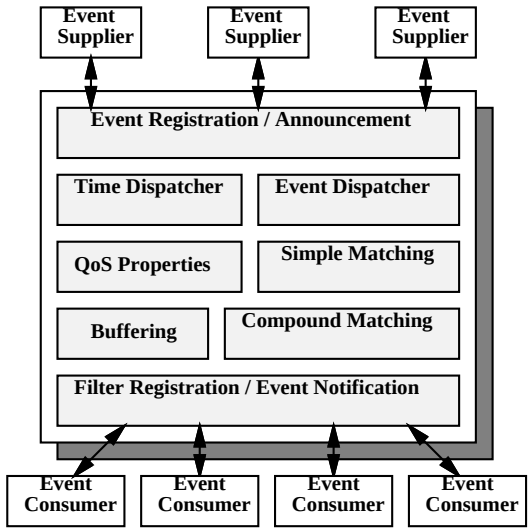

Figure 1: **Internal architecture of READY**

3.1 Time dispatcher

The *time dispatcher* (TD) triggers registered callbacks at specific times or repeatedly at specified time intervals. During callback registration, a registration handle and time-related information are passed to TD. The pair *(callback, registration handle)* constitutes a unique registration identifier and allows the same object to use the same callback for multiple registrations. Using this pair, a registration can be suspended, resumed, or removed. The time-related information can be: an absolute time; a relative time from either the registration time or some absolute time in the future; or a repeating time interval with possible start and end times.

When TD invokes a callback, the handle provided during registration is passed to the callback function, together with a pointer to an instance of a `Time::Tick` event that contains the requested time-related information when the matching expression assigns the event to an event variable (an event instance is used here in order to have the same interface for both time- and event-related callbacks). When an absolute or a relative time is specified during registration, the callback is automatically unregistered after it is successfully invoked. When a time interval is specified, the callback is automatically unregistered when an end time is specified and this time is passed.

3.2 Event dispatcher

An Event Dispatcher (ED) triggers registered callbacks when event instances are received from event suppliers. During callback registration, the caller provides a sequence of event types, a registration handle, and an optional start time at which the registration becomes active and matching occurs. The sequence of event types determines if the callback is invoked when an event instance is received, and

event types can be any of `D::T`, `D::*`, `*::T`, and `*::*`. In the first case, callbacks are invoked for events in domain `D` with type `T`. In the second case, callbacks are invoked for events in domain `D`, independently of their type. In the third case, callbacks are invoked for events with type `T`, independently of their domain. Finally, in the last case, callbacks are invoked for all event instances.

The pairs of callbacks and registration handles are used similarly to the TD case. The start time is needed for handling compound matching that uses the sequence (`;`) operator. Here, callback registrations for the RHS of the specification occur after the registered callbacks for the LHS of the specification are invoked. Therefore, to avoid losing events that occur between the time the LHS callbacks are invoked and the time the RHS callbacks are registered, events should be buffered and the RHS callbacks should be registered with the invocation time of the LHS. Buffering of events is triggered by *pre-registration* of the future RHS callbacks at the time the LHS callbacks are registered.

Similarly to TD, when a callback is invoked, the handle provided during registration is passed to the callback function together with a pointer to event instance that triggered the callback. By default, the callback remains registered after the callback is invoked. The callback can be unregistered, suspended, or resumed by invoking the corresponding ED member functions.

3.3 Simple and compound matching

Matching is implemented using *simple matching elements* (SMEs) and *compound matching elements* (CMEs), which are generated by the READY parser during the parsing of consumer specifications. These entities are organized into trees: parent nodes use children to do some of the matching work and then they combine the results in various ways or perform actions on the results (*e.g.*, notifications). Currently, every element has one parent. In the future, a matching element that covers common sub-expressions across multiple consumer specifications might have multiple parents. SMEs are the leaves of matching trees and encapsulate specifications having one of the following formats:

$$event\_var : event\_type \mid expression$$
$$event\_var[j..k] : event\_type \mid expression$$

SMEs register and unregister themselves with the event dispatcher in response to register and unregister invocations from their parents. CMEs are the parent nodes in the matching tree hierarchies, and are responsible for passing context information to their children and maintaining bindings of matched events. In addition, they are responsible for starting, stopping, suspending, and resuming matching of their children. Context information represents information that is the same across all matching elements (*e.g.,* elements that are part of the same consumer session share the same QoS). Binding information for matched events represents the mapping between event variables, which are used in consumer specifications, and event instances that resulted in successful matches. Binding information is updated by SMEs on every match and is passed to parent CMEs on demand or when matching is completed – this happens on every match when the specification specifies only one event instance, or when the maximum number of matched events is reached when ranges of event instances are used in the specification.

3.4 QoS management

Several QoS properties may be used to specify session properties and indicate the delivery characteristics of notifications. Session-related QoS properties determine the reliability guarantees of consumer sessions, supplier sessions, and READY servers. The supported session-related QoS properties, which are compatible with those of the OMG Notification Service [13], are:

- *None*: events may be lost due to network, node, and process failures;
- *Retry*: transient message failures are masked by re-transmission of lost messages;
- *Reconnect:* transient message and network failures are handled by re-transmission of messages and establishment of lost connections;
- *Persistent Session:* all registration information, such as specifications, QoS settings, and established sessions, is recorded to persistent storage;
- *Persistent Event:* events are recorded to persistent storage.

Notification-related QoS properties determine the ordering of consumer notifications. Configuration files can be used to set notification QoS properties for all consumers. In addition, consumers can use the *admin* session handle to set these properties for all consumer sessions, or they can set them on a per session or session group basis. The supported options are:

- *Priority*: the priorities assigned to events, if any, determine the order of delivery;
- *Validity*: the time-to-live assigned to events, if any, determines the order of delivery. The time-to-live may be expressed as an absolute time or as a relative time. Events whose time-to-live expires are discarded;
- *Ordered*: notifications are buffered when consumers explicitly ask for notifications, receive notifications in batches, or schedule delivery of notifications after an absolute or relative future time. The size of the buffer can be set to a number of notifications or a time duration, and the delivery policy can be: *None, FIFO, Priority, Validity, Supplier-based, Type-based,* and *Timestamp*. Among them,

the last one is the most challenging since it may require global ordering of event timestamps to be implemented. The implementation of global ordering of event timestamps is done by placing matched events in a queue that is ordered by the lowest event timestamp. Notifications occur when the event at the front of the queue has a timestamp that is less than the current time minus `occur_delay`, where `occur_delay` is the sum of the worst-case clock skew and communication delay between an event supplier and a READY server.

3.5 Event matching for consumer specifications

Event matching can be performed at consumers, suppliers, and READY servers. While matching at consumers is fully de-centralized, it requires routing of all events to consumers and, therefore, it is mostly targeted towards high speed connections, complex specifications, and consumers that are interested in the majority of the generated events. When event matching is done at suppliers, network traffic is minimized since events not needed by consumers are not injected into the network. However, this approach is not applicable to environments where the matching overhead may affect the performance of the supplier.

When event matching is performed at READY servers, network traffic towards consumers may be reduced. However, servers may experience scalability problems, depending on the number of connected consumers, the volume of generated events, and the complexity of the registered specifications. Therefore, this approach is mostly applicable to environments where the matching overhead is small, the connections between READY and consumers are slow, and consumers subscribe to small subsets of the events that get announced.

To overcome the above shortcomings and to be able to support several application environments in a highly scalable fashion, READY employs a hybrid matching scheme where matching starts at consumers, as consumers register specifications. Then, parts of or entire specifications are moved upstream towards event suppliers to reduce event traffic. This movement is based on heuristics, where the ideal specification for movement upstream has two properties: it is long lived; and it discards a large fraction of the events it filters. Consumers can enable/disable dynamic movement on a per specification basis.

4 Implementation details

The current implementation of READY supports socket-based TCP/IP and reliable broadcast communication mechanisms. Under TCP/IP, socket connections are created between suppliers and READY servers and between consumers and READY servers. Each supplier application uses only one socket connection per READY server, and supplier sessions are multiplexed over this connection. Similarly, each consumer application uses only one connection with a READY server, and consumer sessions are multiplexed over the same connection.

The reliable broadcast communication infrastructure is provided by IONA's OrbixTalk [16] publish-subscribe product. A wrapper around OrbixTalk was built to leverage the subject-based matching capabilities provided by OrbixTalk. The same wrapper can be used to support other publish-subscribe products (*e.g.,* Tibco's Information Bus, TIB [17]). Event and specification registrations are implemented by using the `Ready.Register.Event` and `Ready.Register.Spec` subjects, respectively. Specifications that use only event types are implemented using the subject-based matching capabilities of OrbixTalk; the event type is used as the subject.

READY servers are implemented as listener applications. All READY servers could potentially receive consumer specification registration messages. However, only one of them will handle the initial registration based on the given network configuration and the event types referenced in the specification. Moving event matching to suppliers is implemented by having suppliers register to receive announcements with `Ready.Dynamic.Match` as their subject. Such announcements are generated by READY servers when they determine that moving matching to suppliers is beneficial.

5 Related work

Yeast [9] is a general purpose event notification system that provides a global space of events that is visible and shared in a wide area network. A event action specification can have a pattern of events which, when matched, triggers a specified action. High-level applications are built as collections of specifications. READY (*REliable, Available, Distributed Yeast*) extends the Yeast specification language by adding structured events, QoS directives, sessions and session groups, and several other language constructs.

The CORBA Notification Service [13] extends the CORBA Event Service [12] by adding structured events, event type discovery, event filtering, sharing of filters among several consumers, QoS properties, and an optional event type repository. Although these additions are closely related to our work, READY offers better grouping mechanisms, much more powerful filtering (compound matching, aggregations, *etc.*), and handles event routing between servers transparently. In contrast, networks of CORBA Notification Channels have to be explicitly managed by the clients of the Notification Service.

NEONet [11] is a persistent messaging middleware offering a publish-subscribe mechanism with format parsing and conversion capabilities. Content-based subscrip-

tion rules are supported, *i.e.,* multi-field matching for single events. Similar to READY, subscribers in NEONet can specify message formats so that they receive the same format from multiple suppliers; this is done by selecting some fields from the original message and adding additional data. Unlike READY, NEONet does not offer local action support, grouping constructs, and compound event matching.

Some of the widely used publish-subscribe middleware products are the Velocity server by Vitria [14], EventWorks by Level8 Systems [15], and Tibco's Information Bus (TIB) [17]. Velocity and EventWorks are based on the CORBA Event Service and use multiple event channels for event notification. While these products support a long list of features, including QoS guarantees, event caching and event replication, they do not support READY's grouping constructs, compound event matching, local actions and event aggregation.

The OPERA group at the University of Cambridge has ongoing work on event-based distributed computation [1, 7, 10]. This work includes a compound matching service that has constructs similar to READY's compound matching operators. In terms of functionality, one of the main differences between this project and READY is our study of higher-level constructs such as the session and grouping constructs. Recently, this group began exploring federation of event services [2].

TAO [6] is an object-oriented, real-time event service that extends the CORBA Event Service to suit real-time systems and support periodic, rate-based event processing, and efficient filtering and correlation. Similar to READY, TAO suppliers specify the types of events they generate, and consumers register filters and QoS demands, with real-time constraints being the most important. In contrast to the rich filtering capabilities provided by READY, TAO only offers source-based, type-based, source- and type-based filtering, and simple event correlation based on conjunctions and disjunctions of events.

6 Conclusions

In this paper, we presented the architecture and implementation of the READY event notification service. In particular, we presented the high-level constructs provided by READY, we discussed several architectural components employed by READY, and we offered some implementation details. Compared to current event products, READY offers a richer set of 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. Preliminary experimental results showed that a scalable event service platform with the judicious set of features and appropriate types is the right approach for providing the glue for developing distributed applications independently.

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] J. Bates, Bacon J, K. Moody, and M. Spiteri. Using events for the scalable federation of heterogeneous components. In *8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, Sintra, Portugal, September 1998.

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

[4] 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.

[5] R. E. Gruber, B. Krishnamurthy, and E. Panagos. High-level constructs in the READY event notification system. In *8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, Sintra, Portugal, September 1998.

[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] C. Ma and J. Bacon. COBEA: A CORBA-based event architecture. In *USENIX COOTS'98*, Santa Fe, New Mexico, April 27–30 1998.

[11] New Era of Networks Inc. NEONet 3.1. http://www.neonsoft.com/prods/index.html.

[12] Object Management Group (OMG). Event Service specification. ftp://www.omg.org/pub/docs/formal/97-12-11.pdf.

[13] Object Management Group (OMG). Notification Service specification. ftp://ftp.omg.org/pub/docs/telecom/98-06-15.ps.

[14] D. Skeen. The enterprise-capable publish-subscribe server. http://www.vitria.com/.

[15] Level8 Systems. EventWorks. http://www.level8.com/eventworks-wp.htm.

[16] IONA Technologies. OrbixTalk. http://www.iona.com/products/messaging/index.html.

[17] Tibco. TIB/Rendezvous. http://www.rv.tibco.com/rvwhitepaper.html.