The Architecture of the READY Event Notification Service

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

Abstract

We describe the architecture and implementation of READY, an *event notification service* that provides efficient, decoupled, asynchronous event notifications. A significant amount of work has been underway in the *publish-subscribe* model of communication, where event consumers (subscribers) accept events from suppliers (publishers) often via a mediator. 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 domains and boundary routers that bound the scope of event distribution and control the mapping of events across domains.

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. Many commercial products are available where event consumers (subscribers) accept events from suppliers (publishers) often via a mediator. Publish-subscribe mechanisms have been retro-fitted to existing middleware products, including persistent message queue products and transaction processing monitors. The CORBA Event Service [17] and the recent CORBA Notification Service [18], a feature-enriched version of the Event Service, are efforts to standardize the middleware.

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, 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 [8, 13]. Frameworks specify a consensus 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.*). Since this consensus will 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 [9, 10]. 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 domains and boundary routers that bound the scope of event distribution and control the mapping of events across domains, among others. The overall architecture of READY, the various components used by READY servers and READY clients, a preliminary experimental evaluation of READY's matching engine, and two existing projects that use READY for event notification are also presented.

The remainder of the paper is organized as follows. In Section 2, we focus on the high-level constructs (operating over multiple events, multiple consumers, *etc.*) provided by READY. Section 3 describes the distributed architecture of READY. Section 4 offers implementation details, the current status of READY, and a preliminary performance evaluation. Section 5 offers an overview of related systems, and Section 6 concludes our presentation.

2 High-Level Event Constructs

This section briefly summarizes the high-level READY constructs; further details can be found in [10]. READY clients interact with READY using *admin, supplier,* and *consumer* sessions. Admin sessions are used for creating/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 both 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) (via a callback function). Notification behavior is controlled by session properties, including quality of service (QoS) properties which control the reliability of notification delivery, and deliver properties which control delivery order (*e.g.,* ordering by event priority order).

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. By default a disconnect implies suspension of associated specifications. If the persistent delivery session property is set, notifications are preserved by READY until delivered, allowing matching to remain enabled even while a session is disconnected.

Specifications can be grouped into a *specgroup*, enabling efficient suspend/resume of a large number of specifications. Consumer sessions can be grouped into a *congroup*, enabling sharing of specifications and specgroups by a set of sessions, as well as 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.

Although not currently part of the READY design, it seems clear that concepts from group communication systems [19, 7] are useful for session groups. For example, one could optionally specify that all consumer sessions in a congroup must receive notifications in a consistent order, group membership changes could be announced as events, *etc.*

2.1 Event Types

A READY event type definition specifies a set of required and optional fields, where each field contains a field name, a type identifier, and a value. READY types can have subtypes, and a subtype declaration simply adds additional required or optional field specifications to those of its parent type. Compound type identifiers such as `Mail.Msg.New` can be used with the convention that the structure of the type corresponds to the type hierarchy. This convention simplifies integration of a READY-based system with a publish-subscribe system: simple matching expressions over a compound type mimic the matching over compound subject names found in many publish-subscribe systems.

2.2 Specifications

A *specification* is composed of one or more 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. For example: $Ev1 : MyAlarm \mid \$Priority > 5$ 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 [18]. 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-(sub- ...) parts of top-level event fields can also be used (see [18] 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 `not`. The '`&&`' and '||' operators begin matching both LHS and RHS at the same time, where '`&&`' matches successfully once both sides match, while '||' matches successfully once either side matches. 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). The general form for a sequence matching expression is:

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

The `event_var[j..k]` indicates that the event variable will be bound to a sequence of events, where `j` and `k` indicate the minimum and maximum number of events in this 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, thus expressions using `Ev1[3..3]` or `Ev1[3..5]` or `Ev1[3..*]` all result in a sequence of length 3 if used at the top level. When contained within a larger expression, however, 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, further constraining the set of possible matches. This expression must be a boolean expression over the event variables in the matching expression (`where` cannot trigger additional matching). This form is useful for expressing inter-event constraints, especially constraints over sets of events. *E.g.,* 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 specification 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, which causes delivery of one or more events to the associated consumer session (congroup), 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 *routing domain* (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 operations over specifications, specification groups, sessions, and session groups described above are all available as local actions.

2.3 Event Routing Domains

There must be a way to manage interactions between different event routing domains (different administrative domains, different application frameworks, and so on). 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. READY specifications already support such mappings. An `announce` action normally supplies an event to the local routing domain, but this can be overridden. Although one can write static adapter applications which consume events from one domain and announce events in another, our approach is more flexible. *E.g.,* dynamic control over the boundary routing is supported; one can easily add, remove, suspend, or resume (groups of) mapping specifications.

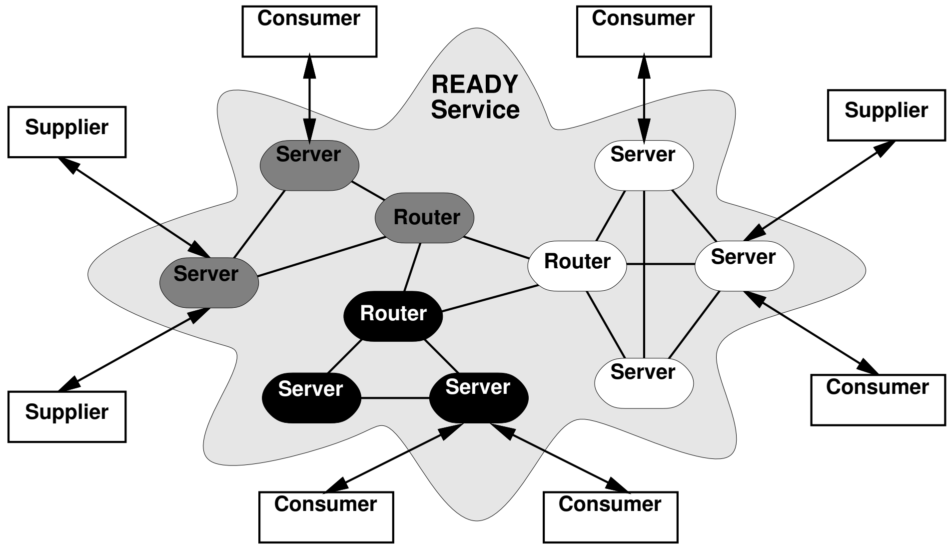

Figure 1: The architecture of the READY event notification service

A further refinement of boundary routers is the ability to restrict event throughput used by inter-domain mapping specifications. We plan to use this to carry out safe trials of new services that interacts with existing "released" services via events. Boundary mapping specifications from the trial to released event domain can perform sanity checks to ensure released services are not passed "bad" events, while bandwidth restrictions on these specifications can shield the released services from performance degradation due, *e.g.,* to a bug in a trial service that generates too many events.

3 The Architecture of READY

Figure 1 shows the overall architecture of READY. Each instantiation of READY is implemented using one or more event domains, and each event domain may contain multiple READY servers. Boundary routers link together two or more event domains, and each event domain may use several routers. Boundary routers are linked together in either a hierarchical or a peer-to-peer topology, depending on the characteristics of the applications build on top of a given READY instance.

Figure 2 shows the architecture of a READY server. Each server consists of several components

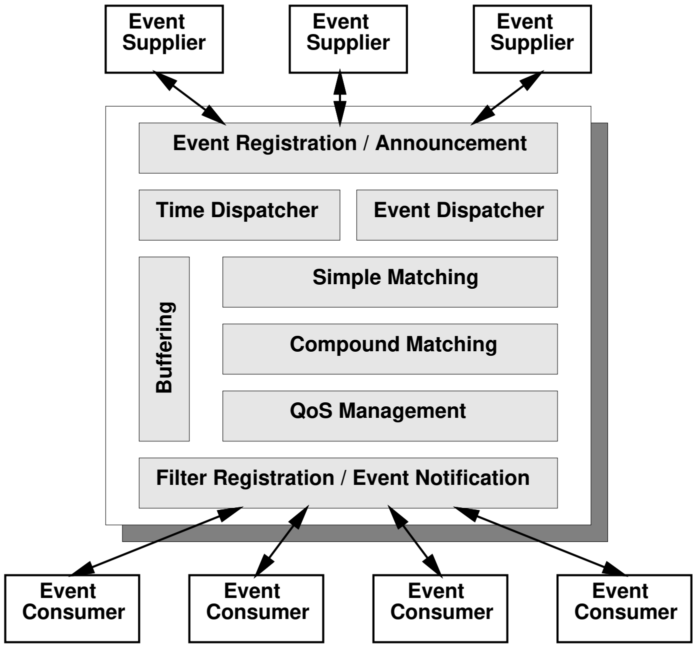

Figure 2: The internal architecture of a READY server

that are linked together to offer high-performance event notification to clients. 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 these components and their interactions.

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)* is a unique registration identifier. This 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. If an absolute time is used with value less than the current time, or a repeating time interval is used with an absolute end time that is less that the current time, the registration fails.

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. We should note that a pointer to an event instance is passed to the callback because we wanted to have the same interface for both time- and event-related callbacks.

By default, when an absolute or a relative time is specified during callback registration, the callback is automatically unregistered after it is successfully invoked. However, when a time interval is specified, the registration status of the callback function can be determined by the returned status of the invocation. The alternatives are: `continue`, `unregister`, `suspend`, and `fail`. In the first case, the callback function remains registered unless an end time was specified during registration and the end time is passed. In the second and last cases, the callback is unregistered. In the third case, the callback is suspended, and if an end time was specified during registration and the callback is not resumed prior to this time, the callback is automatically unregistered.

3.2 Event Dispatcher

An Event Dispatcher (ED) triggers registered callbacks when event instances are received from event suppliers. During the registration of a callback, the following information is required: an event type, a registration handle, and an optional start time at which the registration becomes active and matching occurs. The event type is used by the ED to check if the callback should be invoked when an event instance is received. The event type can have any of the following formats: `D::T`, `D::*`, `*::T`, and `*::*`. In the first case, callbacks are invoked for events in the domain `D` with type `T`. In the second case, callbacks are invoked for events in the 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 pair of callback and registration handle is used similarly to the TD case. The start time is required for handling compound matching that uses the sequence (`;`) and negation (`not`) operators. In these cases, callback registrations for the right hand side (RHS) of a compound specification occur after the invocation of the registered callbacks for the left hand side (LHS) of the specification. Therefore, to avoid losing events that occur between the time the LHS callbacks are invoked and the time the RHS callbacks are registered, the RHS callbacks should be registered with the `match_time` of the LHS, which we assume will be no more than `match_delay` in the past.

The right choice of `match_delay` depends on the configuration of READY, and must take into account: (1) worst-case CPU processing time for determining that a LHS has matched; (2) worst-case clock skew between READY server clocks; and (3) worst-case delay due to communication between READY servers (part of LHS matching time can include moving events, partial match info, or filters across servers). Rather than buffering all events for a time period of `match_delay`, we use *pre-registration* of future callbacks so that ED knows which event types must be buffered.

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

3.3 Simple and Compound Matching

Simple matching elements (SMEs) and *compound matching elements* (CMEs) are used to implement matching. 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. Matching trees are generated by the READY parser during the parsing of consumer specifications.

SMEs are the leaves of matching trees. They encapsulate consumer specifications having the following structure:

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

or

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

SMEs register and unregister themselves with the event dispatcher in response to register and unregister invocations from their parents.

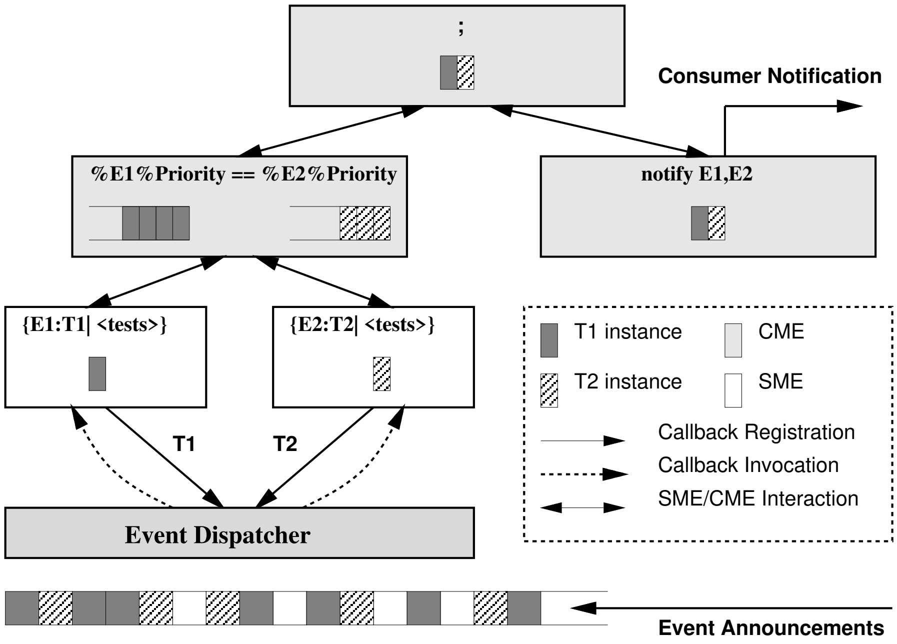

Figure 3: The matching tree for a consumer specification

CMEs are the parent nodes in the matching tree hierarchies, and are responsible for passing context information to their children as well as 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 that share the same "context" (*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.

The interactions between SMEs and CMEs is explained with the help of an example. Figure 3 shows the matching tree generated by READY for the following specification:

$\{E1 : T1 \; | < tests > \}$ && $\{E2 : T2 \; | < tests > \}$ where

$$(\%E1\%Priority == \%E2\%Priority);\ notify\ E1, E2$$

Once the matching tree is created, both SMEs register callbacks with the Event Dispatcher (ED), which are invoked when instances of events with types $T1$ and $T2$ are announced, respectively. Upon a callback invocation, ED passes to SME a pointer to the event instance that triggered the callback. Next, the SME checks whether the $< test >$ part of the specification is satisfied. If it is, an event binding for the matched event is given to the parent CME, which handles the *where* part of the specification. This CME maintains the bindings it receives from the SMEs in queues.

When a binding is received from a SME, the CME searches the queue of the bindings reported by the other SME to see is there is a pair that matches the *where* clause constraint. When such a pair is found, the binding for the pair is passed to the root CME. The root CME then passes consumer-related context information and the binding to the CME responsible for the *notify* clause (context is not shown in Figure 3), which is responsible for sending a notification containing the event instances specified in the binding to the consumer.

3.4 QoS Management

Several QoS properties may be expressed 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. These properties turned out to be compatible with the ones specified in the OMG Notification Service [18]. The supported session-related QoS properties are:

- *None*: Network, node, and process failures can cause events to be lost prior to reaching a READY server, within a READY server, and prior to reaching a consumer;
- *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 re-establishment of lost connections;
- *Persistent Session:* Network failures are handled as above and, in addition, supplier and consumer registration information, such as specifications, QoS settings, and established sessions, are recorded to stable storage;

- *Persistent Event:* Protection against network failures and non-catastrophic failures (*e.g.,* disk crashes) is provided by recording events to a stable log.

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.

- *Priority*: The priorities assigned to events by suppliers, if any, are used to order the delivery of notifications;

- *Validity*: In multiple application environments events may have a time-to-live (validity) assigned to them. The time-to-live may be expressed as an absolute time (*e.g.,* October 10, 1998) or as a relative time (*e.g.,* 1 hour from now). When the valid period of an event is passed, either due to delays within the READY network topology or due to buffering, the event is discarded;

- *Buffered*: Buffering is needed when consumers explicitly ask for notifications, receive notification is batches, or schedule delivery of notifications after a specific future time (absolute or relative). The size of the buffer is set to either a number of notifications or a time duration, and a policy is used to specify the notification delivery order;

- *Ordered*: Notifications can be ordered using one of the following policies: *none, FIFO, priority, deadline, supplier-based, type-based,* and *timestamp.* Among these alternatives, the last one is the most challenging since it may require global ordering of event timestamps to be implemented. Global ordering is not required when the consumer is interested in receiving notifications for events generated by a single supplier. 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 several places: consumers, suppliers, and READY servers. The simplest implementation is to perform all matching at the consumer. While this approach 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, the 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 afore-mentioned shortcomings and, in addition, 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.

With respect to dynamic matching, one can question if it is a good idea to move matching processing costs from clients to servers. The actual benefit of discarding an event will depend on such factors as event size, whether event batching can be used, number of downstream event routing steps that are spared the need to process the event, *etc.* If READY servers are saturating due to excessive event traffic, dynamic matching could move the system out of saturation, effectively increasing the available "event bandwidth." In this case latency due to saturation is replaced with latency due to matching tests, and the result could actually be improved latency. Even when the READY servers are not saturating, if the network is highly utilized then wasting network bandwidth will result in poor performance.

4 Implementation Details and Current Status

Generic matching algorithms for events of arbitrary structure can be costly. Therefore, structured events are needed for optimized event matching. READY events follow the structured event format specified in OMG's Notification Service [18]. Events consist of a *header, filterable body,* and *remainder of body.* The header contains a fixed portion, which consists of the *event type,* and *event name*, and a variable part, which may contain QoS-specific attributes, timestamps, access rights, and so on. The filterable body part consists of attribute-value pairs. The remainder of the body is an opaque value.

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 [22], a publish-subscribe product that offers subject-based matching. 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 [23]). 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.

4.1 Experimental Results

To examine server overheads in our current server implementation, we ran some simple micro-benchmarks with one supplier, one server, and one consumer, all running on the same machine (a standard Sun Ultra 2 with dual 300 MHz ultrasparc processors and 256 MB of RAM). The load on the machine was light, and `gettimeofday()` was used to collect the performance numbers.

Our goal for these benchmarks was to examine performance and scalability of our matching code. Thus, we measured only the cost of processing an event once it has been received by the server. Our future plans are to examine overheads for different networking protocols, where micro-benchmarks measuring send and receive costs for remote suppliers and consumers will be used.

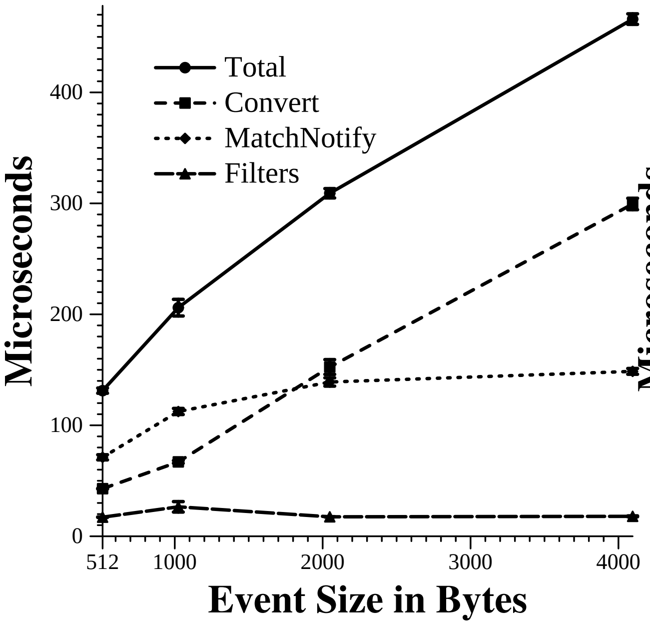

Figure 4: Results for 100 event types

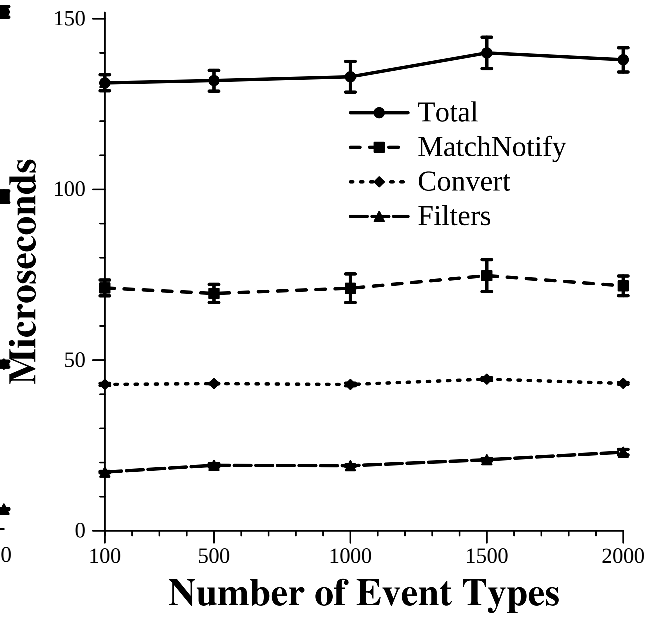

Figure 5: Results for event size 512 bytes

Figures 4 and 5 present results for two sets of experiments. We measure cost of processing a single event by averaging over a batch of size 20,000; the graphs show error bars for 95% confidence intervals (10 batches per data point). Four costs are presented: time to convert the linear representation of an event as sent in a buffer to a C++ struct (`Convert`); time to determine which simple matching elements must be matched against this event (`GetFilters`); time to perform matching and also to write resulting notifications to a socket (`MatchNotify`); and the total of the these three costs (`Total`).

Figure 4 shows how event processing overhead varies with event size. We expect socket write overhead and also conversion cost to grow slowly with event size, while filter lookup should remain roughly constant. This expectation is met, except the conversion cost currently rises too quickly, due to unnecessary copying (this will be fixed shortly).

Figure 5 shows how event processing overhead varies with the number of event types used in event specifications. Since size remains constant (512 bytes), we expect all measures except for filter lookup to remain roughly constant, which is what occurs. Essentially, this experiment tests the scalability of our choice of hash table. In absolute terms the growth due to added types is very small (roughly 2 microseconds per 500 additional events). In relative terms the percent growth is higher than we would like. Thus, some hash table tuning is suggested, but has low priority.

While our experiments suggest two places to tune the matching code, the measured overheads are already very low, and demonstrate that READY can scale gracefully with increasing message size and increasing number of event types.

4.2 Projects Using READY

Currently, READY is being used by two projects inside AT&T Labs, Hermod and Marvel. Hermod [4] provides a high performance, extensible framework for developing applications and services for unified messaging. Hermod views the messaging system as a distributed active database with READY being the component that exposes modifications that occur in the core of the system. Services interested in receiving notification about specific events and event combinations register with READY as event consumers. As a result, services on top of the messaging system can be implemented very rapidly, without requiring any knowledge of the internal structure of the system. An example of such service is SaveMe [3], an archiving system for electronic documents.

Marvel [25] is a web-based network management environment that aggregates information from several agents and presents it as web pages containing Java applets. Marvel is currently being used to manage a broadband data network with cable modems acting as active network elements. The event architecture of Marvel consists of network element management agents (even sources), an event aggregation agent, aggregation management objects (AMOs), and an event visualization layer. READY plays the role of the event aggregation agent and is responsible for collecting events from the various network elements, aggregating, filtering and correlating them before sending them to event consumers (AMOs). Consumers indicate the frequency with which they want event notification.

5 Related Work

Yeast [14] 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, quality of service directives, sessions and session groups, and several other language constructs.

The CORBA Notification Service [18] offers a decoupled, publish-subscribe model of communication via *notification channels*. This service significantly enhances the CORBA Event Service [17] by adding event filtering, durable connections, and delivery-guarantee semantics. Suppliers supply events to notification channels while consumers register simple filters with channels. A Notification channel passes to each registered consumer all supplied events that pass its filters. Suppliers and consumers do not interact directly with a channel, but rather with channel proxies. By attaching a set of consumers to a single proxy object, one can arrange to have these consumers share a set of filters, thus providing a limited form of READY's consumer session groups. In addition to better grouping mechanisms, READY specifications are much more powerful than simple filters, providing compound matching, aggregation, local actions, and so on.

READY also masks distribution issues: READY is implemented as a set of cooperating servers that together implement all necessary event routing, where suppliers and consumers can simply connect to the nearest server – there is currently an RFP for managing a distributed topology of CORBA Notification Channels, but at present such a topology must be explicitly constructed by clients of the Notification Service.

Microsoft's Distributed Component Object Model (DCOM) extends the Component Object Model (COM) [5] by adding network-based component interaction. COM provides event notification capabilities via specialized objects, referred to as "connectable objects," which establish connections between event suppliers and event consumers. However, COM does not support a decoupled mode of communication nor does it provide any of the higher-level constructs provided by READY.

NEONet [16] is a persistent messaging middleware offering a publish-subscribe mechanism with format parsing and conversion capabilities. Content-based subscription 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 functionality.

Vitria's Velocity server [20] and Tibco's Information Bus (TIB) [23] are two widely used publish-subscribe middleware products. Velocity, which is based on the CORBA event service, uses multiple event channels for event notification, offers access control lists for security, caches and replicates events for increased performance and provides quality of service guarantees. TIB, which is based on a subject-based addressing scheme, supports similar functionality. In contrast to READY, these products do not offer grouping constructs, multi-event matching, local actions and event aggregation.

Level8 Systems EventWorks [21] is a reliable and scalable publish-subscribe middleware product which provides rule-based event processing. EventWorks, based on the CORBA event service, uses the concept of themes, which is similar to event channels. Themes use a hierarchical name space that is stored in a directory. Attributes may be associated with themes, and they can be used for filtering. The only similarities between READY and EventWorks is the use of self-describing events and the support for quality of service levels that provide total ordering of events, duplicate free notifications and replication for redundancy.

The OPERA group at the University of Cambridge has ongoing work on event-based distributed computation [1, 12, 15]. 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 [11] is an object-oriented, real-time event service. One of the main contributions of TAO is the extension of the CORBA event service to suit real-time systems and support periodic, rate-based event processing and efficient event filtering and correlation. TAO provides source-based and type-based event filtering, event correlations, and real-time event dispatching. Similar to READY, TAO suppliers specify the types of events they generate, and consumers register filters and quality of service 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.

Active databases offer an *event-condition-action* (ECA) to support database functionality such as view materialization, index maintenance and integrity constraint enforcement [24]. The ECA mechanism consists of an event expression, one or more conditions and an action that is triggered when the conditions are satisfied upon event occurrences. Active databases support composite events for event correlation. However, virtually all active databases are based on a centralized architecture and, therefore, are not suitable for distributed systems. In addition, with the exception of HiPAC [6], the actions specified as part of the ECA rules are internal to the database and do not interact with external applications.

6 Conclusions

In this paper, we presented our work on READY, a high performance event notification service. In particular, we presented the high-level constructs provided by READY, the overall architecture of a READY instance, the various components used by READY servers and READY clients, a preliminary experimental evaluation of READY's matching engine, and two existing projects that use READY for event notification.

Compared to current event service products, READY offers a richer set of entities (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.

We should note that providing high-level constructs does not imply that READY is inefficient. In fact, the opposite is true: high-level declarations enable many optimizations that are not otherwise available. For example, high-level supplier descriptions, combined with high-level consumer specifications, enable low-level event-routing optimizations. In addition, 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 towards suppliers.

Our work demonstrates 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. Currently, we are debugging the implementation of the compound matching component. In addition, we are implementing filter movement, boundary routers, and authentication and security constructs.

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] S. Berchtold, A. Biliris, and E. Panagos. SaveMe: A system for archiving electronic documents using messaging groupware. In *International Joint Conference on Work Activities Coordination and Collaboration*, San Francisco, California, February 1999.

[4] 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.

[5] Microsoft Corporation. Component object model (COM). http://www.microsoft.com/com/.

[6] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ladin, D. R. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. *ACM-SIGMOD Record*, 17(1):51–70, March 1988.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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

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

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

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

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

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

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

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

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

[24] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[25] S. Yucel and N. Anerousis. Event aggregation and distribution in web-based network management systems. Technical report, AT&T Labs – Research, August 1998.