CORBA Notification Service: Design Challenges and Scalable Solutions

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

E. Panagos
Voicemate Inc.
thimios@voicemate.com

Abstract

In this paper, we present READY, a multi-threaded implementation of the CORBA Notification Service. The main contribution of our work is the design and development of scalable solutions for the implementation of the CORBA Notification Service. In particular, we present the overall architecture of READY, discuss the key design challenges and choices we made with respect to filter evaluation and event dispatching, and present the current implementation status. Finally, we present preliminary experimental results from our current implementation.

1. Introduction

Today, there is growing interest in the use of general-purpose event notification services as "middleware" for monitoring the behavior of large-scale distributed systems, enabling reactive processes, and integrating independently-developed applications. In an event-based communication model, information producers announce information on one or more communication channels, while information consumers subscribe to channels of interest. Several commercial products are available where event consumers (subscribers) accept events from suppliers (publishers), often via a mediator.

The CORBA Notification Service [13] offers asynchronous and decoupled event-based communication between distributed and heterogeneous applications. The most important feature of this service is *filtering*. Filtering refers to the ability to specify constraints that should be satisfied by the information propagated to consumers. As the volume of information announced by suppliers (*i.e.,* incoming events) grows, it becomes more important to enable consumers to express their constraints accurately in order to increase precision, *i.e.,* the ratio of relevant to non-relevant events that are dispatched to them. However, increasing the accuracy of filters implies increase in filter complexity, which in turn impacts the performance of the notification service.

In this paper, we present READY, a multi-threaded C++ implementation of the CORBA Notification Service on top of the AT&T omniORB, a free high performance CORBA ORB [14]. The main contribution of our work is the design and development of scalable solutions for the implementation of the CORBA Notification Service. We address scalability in two ways: performance and engineering. We are concerned with providing an architecture that supports growth in the number of consumers and the filters they register so that similar performance can be achieved by increasing the available computing resources, without re-engineering the service. READY addresses this scalability challenge by exploiting parallelism during filter evaluation and event dispatching.

The remainder of the paper is organized as follows. In Section 2, we offer an overview of the CORBA Notification Service. In Section 3, we investigate filtering and dispatching alternatives that can be used in an implementation of this service. In Section 4, we present the architecture of READY and describe in detail its filter evaluation and event dispatching components. In Section 5, we present a preliminary quantitative evaluation of the current implementation. In Section 6, we compare our work against existing work and, finally, we conclude our presentation in Section 7.

2. The OMG CORBA Notification Service

The Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) standard describes a software bus, called Object Request Broker (ORB), that allows applications to communicate with both local and remote objects by invoking methods statically/dynamically, independent of the specific programming language and techniques employed to implement these objects [11].

Under the standard, two-way CORBA communication model, a client invokes operations on objects located at a server and then waits until the server sends back a response. This model, however, is quite restrictive for applications that do not want to be aware of the server that will respond to their requests, for applications that want asynchronous

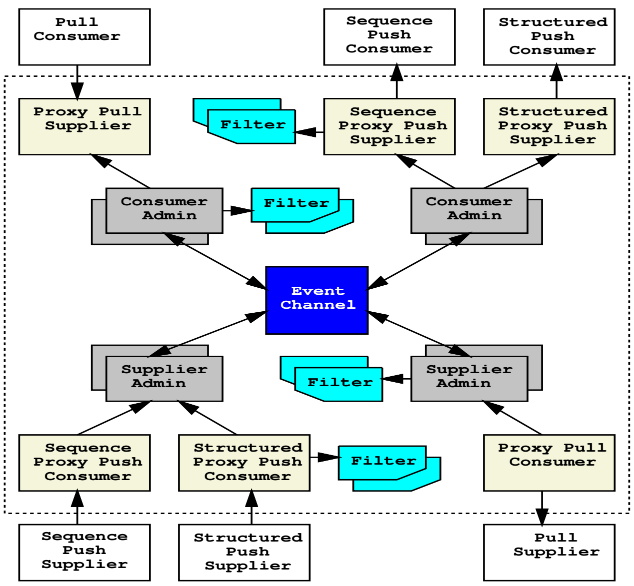

Figure 1. CORBA Notification Service

calls, and for servers that want to broadcast data to numerous clients without knowing their identities.

The CORBA Notification Service [13], which extends the CORBA Event Service [12], addresses these limitations by offering a decoupled, publish-subscribe model of communication. This is accomplished by using the concept of *events* that originate at *suppliers* and are delivered to any number of *consumers*. Suppliers are not required to know the number and identities of the consumers and, similarly, consumers do not have to know the event suppliers.

Figure 1 shows the general architecture of the CORBA Notification Service. The service consists of three object types: *event channels, administrative objects,* and *proxies*. Conceptually, event channels receive events from event suppliers and dispatch them to event consumers. Administrative objects are factories for proxy objects and provide "group-level" operations that apply to all of their proxies. Finally, proxy objects are the CORBA communication endpoints for clients of the service.

The CORBA Notification Service supports two fundamental styles of event communication: *push* and *pull*. Under push, an event supplier sends events to a channel by invoking a push operation on the proxy object it is connected with, and the channel delivers events to event consumers by invoking a push operation on them. Under pull, the channel requests events from suppliers by invoking a pull operation on them, and consumers receive events by invoking a pull operation on the proxy object they are connected with. Both operations are implemented using the standard two-way CORBA communication model, blocking or non-blocking, and all combinations of push/pull suppliers and consumers can be used.

Three kinds of events are supported: *untyped, typed,* and *structured*, as well as sequences of structured events. Untyped events package an event message, which is stored in a data structure defined by the clients of the Notification Service, into a CORBA `any`. Typed events correspond to interfaces used for event communication. These interfaces must be agreed upon by the clients of the service. Finally, structured events provide a well-defined data structure, comprised of name-value pairs, into which a wide variety of event types can be mapped.

Event suppliers and consumers can associate *filters* with both proxies and administrative objects. Each filter contains a list of *constraints*. Each constraint is a boolean expression, which is specified in a constraint language, over the attributes of an event. At least one of the constraints in a filter must be satisfied in order for an event to be either sent to the channel or forwarded to a consumer. In the case where multiple filters are associated with a proxy or administrative object, at least one filter must be satisfied in order for an event to be either sent to the channel or forwarded to a consumer.

Filters attached to an administrative object are shared by all proxies created by this object, and they are combined using AND or OR semantics with proxy filters. In the AND case, an event is sent to the channel or forwarded to a consumer when one of the filters attached to the proxy and one of the filters attached to the administrative object are satisfied. In the OR case, an event is sent to the channel or forwarded to a consumer when either a filter attached to the proxy or a filter attached to the administrative object is satisfied. The case where no filter is specified is equivalent to a filter that is always satisfied.

Another important feature of the CORBA Notification Service is the ability for clients of the service to define quality of service requirements that determine the reliability, validity, and delivery characteristics of event messages. The quality of service requirements can be specified at several granularity levels (event message, proxy, administrative object, and channel). Finally, the CORBA Notification Service defines interfaces that enable the creation and management of networks of channels.

3. Filtering and Dispatching Alternatives

The scalability of a CORBA Notification Service implementation depends on several factors, including the hardware platform and operating system used, the capabilities of the underlying CORBA ORB, the performance of the matching engine, and the efficiency of the event filtering and dispatching modules. In the remainder of this section, we concentrate on the various architectural alternatives that can be used for event filtering and dispatching assuming, without loss of generality, push-based communication.

Since filter evaluation is CPU intensive and event dispatching is I/O intensive, overlapping them would increase parallelism. This observation suggests that separate threads should be used for filter evaluation and event dispatching. However, in order for this approach to scale, the overhead of the extra threads and synchronization that may be required has to be less than the overhead of having the same set of threads be responsible for both filter evaluation and event dispatching.

3.1. Filter Evaluation

There are two main dimensions for efficient filter evaluation: parallel processing of the events sent to the channel, and parallel processing of consumer filters. We explore each dimension separately, assuming sequential execution of the other dimension.

Under parallel processing of events, a single thread evaluates all consumer filters being attached to administrative objects and proxies. An obvious benefit of this approach is that no synchronization is required at the event level. However, the same filter may be evaluated concurrently by two or more threads for different events and, consequently, synchronization may be required for protecting the consistency of internal filter state. This synchronization overhead may be substantial when a large number of filters is used and events arrive at a high rate.

Depending on the complexity and selectivity of filters, the evaluation of all relevant filters for a given event may be completed faster than the evaluation of all relevant filters for a second event by a different thread. Therefore, the order in which events are delivered to consumers may differ from the order in which these events were supplied to the event channel. While this is not an issue for consumers that are not sensitive to event ordering, it introduces additional overhead when events must be dispatched in the order in which they arrive (most monitoring applications require events originating from the same supplier to be dispatched in the order in which they were sent to the channel).

Under parallel filter processing, disjoint groups of filters are evaluated in parallel for the same event by several threads. Here, different criteria can be used for grouping "related" filters together, *e.g.,* all filters attached to an administrative object or a proxy are assigned to the same group. Since each filter is accessed by one thread, synchronization at the filter level is not required. Furthermore, there is no mismatch between event arrival and filter evaluation order because each event is processed after filter evaluation has taken place for all events that arrived before it. Finally, the number of threads and filter groups can reflect the complexity and number of registered filters, leading to better scalability.

However, because an event may be accessed by many threads at the same time, updates to event-specific state would have to be serialized, or each thread would have to maintain its own copy of event state. Event state is needed when filters specify constraints that need to extract specific value parts of event attributes. Furthermore, the allocation of filters to filter groups may lead to substantial differences in the processing overhead for each such group or unnecessary evaluation of filters for the same event. The latter occurs when filters attached to an administrative object belong to a different group than the filters attached to the proxies of this object and the inter-filter operator has OR semantics.

3.2. Event Dispatching

Events are dispatched to consumers after the successful evaluation of their filters. Since the event channel is responsible for sending matched events to consumers when push communication is used, the performance of the channel depends on the dispatching strategy employed. Assuming a large number of connected consumers, a multi-threaded event dispatching architecture is the preferred choice. The two main threading alternatives here are *thread per consumer* and a *thread pool*. Each alternative uses the standard CORBA two-way communication protocol of unicast messages over TCP/IP connections.

Under the first choice, a thread is created for each consumer to handle the forwarding of events. Although this approach is simple to implement, it can consume a large number of operating system resources when the number of consumers is high. In addition, it is very inefficient for consumers that connect to the channel for short periods of time [17]. Nevertheless, when a small number of consumers are connected to an event channel for long periods of time, this approach offers good performance.

In the thread pool approach, a fixed number of threads is used for dispatching events to all connected consumers. This approach avoids the shortcoming of the thread per consumer approach since these threads are *not* associated with a single consumer. However, it may require synchronization between threads when they share the same queue(s) that contain the events that need to be dispatched.

4. The Architecture of READY

Figure 2 shows the internal architecture of a READY event channel. The event channel manages an event queue that stores the *only copy* of each announced event. The size of the queue can be restricted to a maximum size or set to infinite. In the former case, the current implementation discards new events when the queue becomes full.

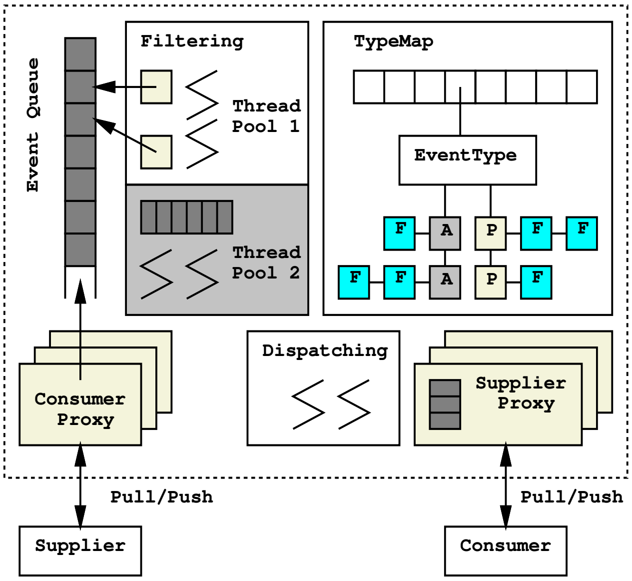

Figure 2. Internal Architecture of READY

4.1. Efficient Filter Selection

For each event sent to an event channel, there is a set of filters that must be evaluated. Since the set may only contain a very small portion of all consumer filters, it is important to be able to efficiently compute it. Typically, the constraints specified in a filter include a list of domain and type names that must be present in an event to make the event relevant to the filter [13] – '*' is used when the domain or type name is not important or not specified. We use a hash table ($TypeMap$) whose key is the event type, *i.e.,* domain name and type name, and value a reference to the filter, to address this.

For each filter, we maintain the list of event types referenced in the constraints of the filter. We use '*' for domain and type names when no names are specified. From this list, we create the list of the most general event types that dominate all existing types[1], and we insert an entry into $TypeMap$ for each entry in the list. When the filter constraints are updated, $TypeMap$ is updated when the list of dominating event types changes.

Upon filter evaluation for an event with domain name D and type name T, the filters to be evaluated are the $TypeMap$ entries under the keys $(*, *)$, $(D, *)$, $(*, T)$, and (D, T). However, care is required when evaluating these filters to avoid redundant work and dispatching the same event multiple times to the same consumer. This is because the set contains both administrative and proxy filters. We address this by organizing the entries for each key present in $TypeMap$ into two lists. The first list contains filters attached to administrative objects, grouped by administrative object, and the second list contains the filters attached to proxy objects, grouped by proxy, as shown in Figure 2. Filters in each group are evaluated sequentially and once one of them is satisfied, the rest of the filters in the same group are skipped.

4.2. Multi-threaded Filter Evaluation

Our filter evaluation approach is based on the parallel filter evaluation alternative presented in Section 3. In particular, we group administrative objects into administrative groups using hashing. The number of administrative groups is a configuration parameter. Several threads are created to carry out filter evaluation for these groups, and each group is handled by only one thread. These threads can belong to one or two thread pools, depending on the particular configuration of the event channel.

In the former case, each thread evaluates the filters of the administrative objects in the group(s) it manages, together with the filters of the proxy objects created by these administrative objects. In the latter case, threads in the first pool evaluate the filters of the administrative objects present in the groups they manage. Depending on the evaluation outcome, an entry may be inserted into a *work queue* that is shared by the two thread pools. Each work queue entry consists of a reference to an event E, a reference to an administrative object A, and the outcome of the evaluation of A's filters with respect to E. Each such entry is dequeued from the queue by a thread in the second queue that, in turn, evaluates the filters of all proxies created by the administrative object referenced in the entry.

By splitting the filter processing between two thread pools, filter processing becomes sensitive to both the complexity and distribution of filters between administrative objects and proxies. Consequently, higher scalability can be achieved because of the increased level of parallelism. However, the total number of threads to be used, as well as their assignment to the two pools is a very important issue.

If we assume that the complexity and number of filters attached to administrative objects and proxy objects are similar, then the ratio of administrative objects to proxies can be used to guide thread distribution. Finally, the total number of threads that should be active at a time is machine dependent, and is determined by the number of available processors, multiplied by a factor that accounts for intra-processor concurrency.

[1]Event type ET_i dominates ET_j when the domain name of ET_i is the same as the domain name of ET_j or has the value of '*' and the type name of ET_i is the same as the type name of ET_j or has the value of '*'.

4.3. Scalability of Constraint Expression Matching

If there are relatively few constraints per event type, or if all of the constraint expressions are independent, then it makes sense to evaluate each filter's constraints independently. Here, efficient matching depends on an efficient constraint evaluation mechanism. If there is a large number of constraint expressions that share common subexpressions, it may be more efficient to treat all of them as part of a single constraint matching problem, where shared subexpressions could be evaluated just once. While we have designed such a mechanism, we note that the placement of filters at the administrative objects already allows for sharing of constraints across consumers, and it is not clear how important it will be to optimize common subexpressions across filters placed at proxy objects.

Regardless, single-constraint matching must be efficient for the channel performance to scale with a large number of registered filters. Our implementation translates constraint expressions into a bytecode sequence for a simple stack-based virtual machine. The constraint grammar is fairly straightforward; translation to bytecodes is done in a single pass by a parser generated by `bison`. Translation occurs just once, when a constraint is added to a filter; there is no per-evaluation cost and evaluation by our bytecode interpreter is fast.

4.4. Event Queue Management

The event queue is accessed by three thread types: supplier, filter evaluation, and maintainance. Supplier threads insert new events into the queue. These threads are either created by the underlying ORB when push communication is used, or managed by READY when pull communication is used, respectively. Threads responsible for filter evaluation of administrative group members need to access all events in the queue. Furthermore, these threads must access exactly the same events. Finally, maintainance threads perform garbage collection of events that are not needed any longer.

To simplify event queue management, the current implementation does not allow the thread pool that handles filter evaluation for administrative groups to change its size during run-time. The size of this pool is registered with the event queue manager, and each thread in the pool is assigned a handle that is used for accessing events. Each handle maintains the current state of a thread with respect to the event that should be accessed next. As long as there are more events in the queue to be accessed by a thread, access proceeds *without* synchronization. Once all events are processed by a thread, the thread is blocked until new events are inserted in the queue.

READY associates a reference counter with each event present in the event queue. Events whose reference counter is 0 are garbage collected. When an event is inserted into the queue, its reference counter is set to the number of handles that have been allocated for the above threads. When a thread asks for the next event, the reference counter of the last event processed by this thread is decremented by one. This process prevents garbage collection from discarding events that have not been processed by all threads.

4.5. Event Dispatching

Each proxy connected to a consumer has a queue associated with it. The queue contains references to events that need to be dispatched to the consumer. Events are inserted into proxy queues by threads performing filter evaluation and get dispatched to push consumers using one of the two supported alternatives: *thread pool* or *thread per consumer*. When an event is inserted into a proxy queue, its reference counter is incremented by one. When an event is dispatched to a consumer, its reference counter is decremented by one.

We should note that event dispatching is a crucial component of an event channel because it can affect the performance of the channel. The reason for this is the synchronous callback required for sending events to push consumers. Here, the thread that pushes an event to a consumer remains blocked until the consumer code returns from the callback function that handles the event. If this function is not designed to process events quickly, the dispatch rate of the event channel may drop considerably. Even worse, the channel may stop functioning correctly when a thread pool is used for event dispatching and each thread in the pool is blocked waiting for a consumer to process an event.

READY addresses this problem by monitoring the threads that push events to consumers. In particular, for each such thread, we maintain a timestamp and a flag. The timestamp corresponds to the time of the most recent attempt to push an event to a consumer, and the flag is set for the duration of the push operation. Using the above information, we can easily detect whether a given thread is being blocked for too long during the push of an event. Currently, we mark the proxy connected to such consumer so that there are no attempts to dispatch more events to the consumer, and we create a new dispatch thread.

5. Performance Evaluation

Assuming that suppliers and consumers connect to an event channel for relatively long time periods, the number of announcements plus the number of notifications per second, *A+N per Second*, is a simple means of measuring event-based *message traffic* to and from the channel. In the remainder of the section, we examine the performance of

Parameter	Setting
Event queue size	0 (infinite)
Garbage collection period	300 seconds
Number of administrative groups	2
Number of threads in thread pool 1	2
Number of threads in thread pool 2	4
Number of event dispatching threads	8

Table 1. READY Configuration Parameters

the current READY implementation using the above metric.

5.1. Experimental Settings

In all experiments, push communication is used and only one supplier is present. The rate at which events are supplied is determined by $goal$, which is the number of events per second that should be supplied to the channel. Therefore, if N consumers are connected to the channel, and each one receives all events, then the ideal throughput for the channel is *(1+N)*goal* per second.

Two machines were used for the experiments. Supplier and consumer processes were executed on a 2-processor (266 Mhz) Sun UltraSparc2 with 512 MB of main memory and running Solaris 2.6. READY was executed on a 4-processor (300 Mhz) Sun Enterprise 4500 with 2 GB of main memory and running Solaris 2.6. The two machines are connected by a 100 Mbps switched Ethernet. Event suppliers, consumers, and READY are written in $C++$, using the latest release of omniORB [14], and were compiled using SUN's $SC4.2$ with the $-O2$ compilation flag.

Each event has 10 filterable fields, named *Field_0, Field_1, ... Field_9*. *Field_0* contains a string whose size is between 28 and 40 characters. *Field_1* contains a counter value. The supplier increments the counter by one every time it pushes a new event to the channel. For the rest of the fields, a random CORBA `long` is assigned to odd field numbers and a random CORBA `double` is assigned to even field numbers. Finally, the type of each event instance is randomly selected from a predefined list of types.

To obtain our numbers, we run a set of clients for a long time interval and report measurement (or *deltas*) every $2,000$ announcements or notification occurrences. Finally, the configuration settings for each event channel are summarized in Table 1. Here, the two thread pools correspond to the two thread pools used for filter evaluation.

5.2. Event Channel Scalability

Here, we present two experiment sets. In each set, the number of consumers ranges from 1 to 50, and each consumer uses one filter that is attached to the proxy it is connected with. In the first set, the number of notifications is set to the number of event announcements (*i.e.,* for N consumers, each receives $1/N$ notifications). This was achieved by assigning a serial number to each consumer, ranging from 0 to N-1, and creating a filter constraint that is satisfied only when the value of *Field_1* modulo N is equal to the consumer serial number. By keeping the number of notifications equal to the number of announcements, the messaging overhead of the channel remains the same regardless of the number of consumers. However, the administrative and filtering overhead of the channel increases as the number of consumers increases.

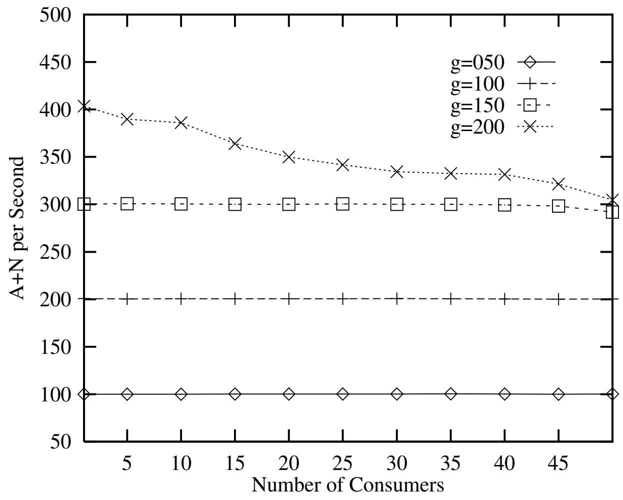

Figure 3. Notifications == Announcements

Figure 3 shows the results of this experiment set for event announcement rates g, ranging from 50 to 200. In the ideal scaling case, *A+N per Second* would be two times the rate of event announcement for any number of consumers. As we can see from Figure 3, READY achieves ideal scaling when g is 50 or 100. When g is 150, the ideal scaling of 300 is preserved as long as the number of consumers is less than 45. When g is 200, the event channel cannot sustain the ideal rate of 400. In particular, when the number of consumers is 25, the performance is about 15% lower than the ideal, while with 50 consumers, the performance is 25% lower than the ideal. Since each consumer has only one filter, the degradation in performance is due to administrative overhead.

In the second set of experiments, each consumer filter matches all events. Here, in the ideal scaling case, the number of announcements plus notifications would have been $(N + 1) * g$, where N is the number of consumers and g the event announcement rate. Figure 4 shows the results for this experiment set. When g is 50, READY achieves ideal scalability as long as the number of consumers is less

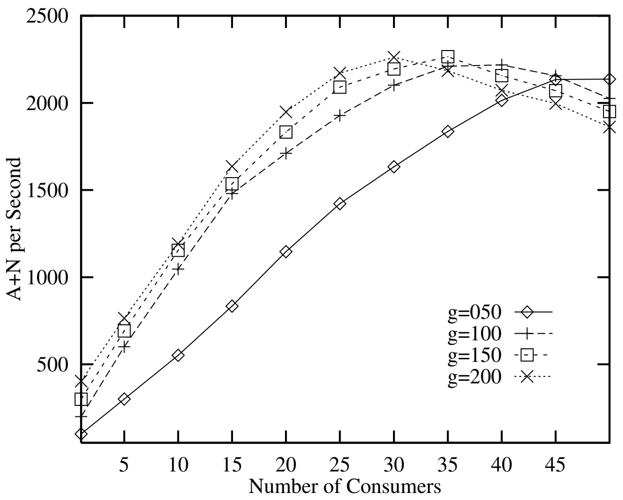

Figure 4. Consumers Receive all Events

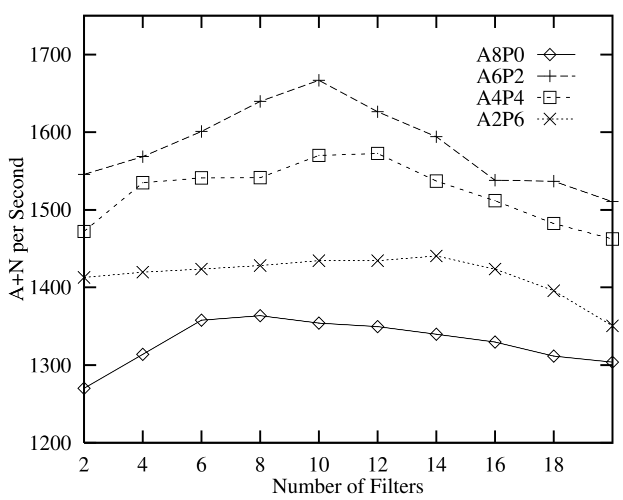

Figure 5. Filter Evaluation Alternatives

than 40. When the event announcement rate is increased, READY exhibits very good scalability when the number of consumers is kept below 35 for g 200, 40 for g 150, and 45 for g 100. After these points, performance starts to degrade, and the drop in performance increases faster for higher event announcement rates.

The administrative overhead of the channel increases considerably when the number of events that need to be dispatched to consumers increases. This is because READY has to acquire an exclusive latch on each event that is dispatched to multiple consumers by different threads[2]. Since the latch is held for the duration of the dispatch, a thread may be blocked while trying to send an event to a consumer when the same event is being sent to a different consumer by another thread. As the number of events that are sent to all consumers increases, the blocking overhead starts to affect the scalability of an event channel.

5.3. Threading Alternatives

Here, the number of consumers is fixed at 15, and each consumer uses several filters, ranging from 2 to 20. These filters are distributed evenly between administrative and proxy objects (each consumer has its own administrative object). The inter-filter operator is set to AND, and filter constraints were designed to force the evaluation of all filters for each event. Finally, the number of administrative groups and filtering threads is set to 8, and the announcement rate of events is set to 100.

Figure 5 shows the results for 4 different thread distribution settings, denoted by `AnPm`. `An` is the number of threads in the first pool (these threads evaluate administrative-level filters only), and `Pm` is the number of threads in the second pool (these threads evaluate proxy-level filters only). For the `A8P0` setting, there is only one thread pool used for filter evaluation (see Section 4 for details).

As we can see from the above figure, using two thread pools for filter evaluation scales better than using just one thread pool. The same figure also shows that the number of threads to be used in each pool is an important issue. In addition, the figure shows that choosing the number of threads is each queue by using the intuition we provided in Section 4, does not always produce the best results. In this case, `A4P4` was expected to be the best. However, `A6P2` proved to be the winner. Currently, we are in the process of performing more experiments to come up with guidelines for choosing the number of threads.

6. Related Work

Event-based communication systems can be classified into three categories: *channel-based, subject-based,* and *content-based*. In channel-based systems, suppliers send their events to some channel while consumers connect to a channel and receive all events sent to the channel. Since no filtering is supported by the channel, scalability is achieved by either having one thread for each consumer, when the number of consumers is small, or using a thread pool, when the number of consumers exceeds a threshold that depends on the underlying hardware and software configurations.

Subject-based systems extend channel-based systems by adding a level of filtering between suppliers and consumers. Event suppliers sent their events to a channel after attaching an "address" or "subject" to them. The subject encapsulates

[2]The current release of omniORB throws marshaling exceptions when several dispatch threads attempt to send the same event to different consumers. We are currently working with the designers of omniORB to implement a more scalable solution.

the particular type of the event and allows consumers of a channel to only receive events that have a specific subject or subject pattern. Several commercial products offer subject-based communication [20, 19, 9] and some research prototypes have been developed around this model [5].

When most of the announced events match the subjects/subject patterns registered by consumers, the threading approaches used in the channel-based model can be used here since matching overhead is minimal when compared to I/O overhead. However, when most of the announced events are of no interest to the majority of the consumers, or when consumers have real time delivery constraints, a separate event filtering module is highly desirable. To the best of our knowledge, only [5] explores this issue to some degree. However, the emphasis of this work is on event dispatching architectures when consumers have real-time requirements.

Content-based systems extend subject-based filtering by allowing consumers to specify filters that use the entire structure of the events [10, 18, 3, 6, 2, 8], and existing implementations of the CORBA Notification Service specification [7, 15, 4]. While some performance studies exist for some of the above systems, these studies concentrate on specific modules of the systems, *e.g.,* event matching algorithm [1]. In contrast, the work presented in this paper focuses on the threading policies and architectures that can benefit the scalability of a content-based event system.

Although the available information with respect to the internal architecture of commercial implementations of the CORBA Notification Service is limited due to the proprietary nature of them, it seems that a multi-threaded architecture on top of a multi-threaded ORB is the preferred choice. Regarding performance numbers, only [16] shows performance results. These performance experiments, however, only study the threading policies for sending notifications to consumers, and they do not show how the various threading alternatives for filtering affect the performance of the system.

7. Conclusions

The CORBA Notification Service offers a powerful mechanism for asynchronous, decoupled, event-based communication between distributed and heterogeneous applications. However, since the service is just an interface specification, the design choices made during the implementation of the service have a direct impact on the performance that can be achieved. In this paper, we presented READY, a multi-threaded implementation of the CORBA Notification Service specification. In particular, we presented the overall architecture of READY, the design choices we made with regards to the filter evaluation and event dispatching components, and the results of a preliminary performance evaluation.

References

[1] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on the Principles of Distributed Computing*, Atlanta, GA, USA, May 1999.

[2] S. Brandt and A. Kristensen. Push as an internet notification service. http://www-uk.hpl.hp.com/people/ak/doc/ins.html.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design of a scalable event notification service: Interface and architecture. Technical report cu-cs-863-98, Department of Computer Science, University of Colorado, August 1998.

[4] DSTC. CORBA Event Notification Service Project. http://www.dstc.edu.au/Products/CORBA/ Notification_Service/.

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

[6] IBM Research. The Gryphon Project: Advanced Message Brokering. http://www.research.ibm.com/gryphon.

[7] IONA Technologies. OrbixNotification. http://www.iona.com/products/messaging/notification.

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

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

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

[11] OMG. Common Object Request Broker Architecture. http://www.omg.org/corba/whatiscorba.html.

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

[13] OMG. Notification Service specification. ftp://ftp.omg.org/pub/docs/telecom/99-07-01.pdf.

[14] omniORB2. A high performance CORBA 2 ORB. http://www.uk.research.att.com/omniORB.

[15] PrismTech. OpenFusion. http://www.prismtechnologies.com/products/ openfusion/main.htm.

[16] PrismTech. OpenFusion notification service performance evaluation. http://www.prismtechnologies.com/products/ openfusion/Whitepapers/performance.pdf.

[17] D. Schmidt. Evaluating architectures for multithreaded object request brokers. *CACM*, 41(10):54–60, Oct. 1998.

[18] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. http://www.dstc.edu.au/Elvin/doc/papers/auug97/ AUUG97.html.

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

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