Managing Distributed Systems with Smart Subscriptions

Robert E. Filman

Research Institute for Advanced Computer Science
NASA Ames Research Center
Mail Stop 269-1
Moffett Field, California 94305
rfilman@arc.nasa.gov

Diana D. Lee

Caelum Research Corporation
NASA Ames Research Center
Mail Stop 269-1
Moffett Field, California 94305
ddlee@arc.nasa.gov

Abstract

We describe an event-based, publish-and-subscribe mechanism based on using "smart subscriptions" to recognize weakly-structured events. We present a hierarchy of subscription languages (propositional, predicate, temporal and agent) and algorithms for efficiently recognizing event matches. This mechanism has been applied to the management of distributed applications.

Introduction

This work arose in the context of developing a framework (the Object Infrastructure Framework, or OIF) to simplify creating distributed applications. That project developed technology to endow distributed systems with better reliability, security, quality of service and manageability. OIF extended the standard "procedure-call" mechanisms of distributed object technology (e.g., CORBA, Java RMI) to include discrete wrapping *injectors* on the communication paths between components. By injecting the behavior for error recovery, redundancy, security checks, intrusion recognition, priority queue management, and so forth, OIF made substantial progress in separating and simplifying the first three classes of requirements: reliability, security and quality of service [5].

However, much of manageability — fault diagnosis, intrusion detection, performance analysis and accounting [14] — proved resistant to a pure injector approach. Injectors provided a locus for recognizing managability events, but not a mechanism or architecture for reporting them. This suggested extending OIF with an event mechanism. Critical issues in the design of that mechanism were: (1) defining what makes an event interesting and (2) declaring to whom should an interesting event be reported.

We eschewed direct connections between event consumers and producers. Such an architecture is brittle, requiring too much knowledge of the structure of a system in too many places—changes in the organization of event producers need to be reflected in every event consumer. Direct connection can also impede critical activities of the producers to perform the event management, which might be as trivial as bookkeeping.

A solution to these problems is to position an intermediary between producers and consumers, an *event channel*. Producers funnel their events to the event channel. Consumers describe to the event channel which events interest them. The event channel is responsible for forwarding the appropriate events to the interested consumers. This notion of event channel also goes under the rubric "Publish and Subscribe." Event consumers *subscribe* to the (interesting) events *published* by event producers.

Here we consider how to code event channels so that exactly the right events are delivered to the appropriate consumers. We seek expressiveness without overwhelming communication cost. This is accomplished by directing only interesting events only to interested parties. Communication costs far more than processing, so it is better to expend effort checking that the communication is desired than to communicate volumes of uninteresting data. Of course, local processing isn't quite free, either. We are thus addressing two issues: (1) what *subscription languages* allow consumers to precisely express interesting events, and (2) which algorithms allow event channels to organize the subscription space so as to efficiently recognize events matching subscriptions.

Architecture

The two relevant interfaces for this discussion are event *consumers* and *event channels*. A consumer is an object to which one can *publish* an event (encoded as a string). A consumer is entitled to do whatever it wants with an event. The simplest consumer might just print the event on a debugging screen, write the

event to a log file, or update a database with some salient facet of the event. Any object can be an event *producer* by composing a string that is a good event representation and invoking publish on some consumer. The interface *event channel* extends consumer with a *subscribe* method. Subscribe takes

(1) a reference to a consumer,
(2) a description, in some *subscription language*, of the set of events interesting to that consumer,
(3) a description of what about the existing event and environment is to be reported to the consumer (that is, the structure of an event to publish to the consumer), and
(4) optional signature information (discussed below) that can be used to optimize subscription algorithms.

Subscribe returns a ticket for managing the subscription. Using that ticket, the subscriber can modify or cancel a subscription. Event channels also include a method for obtaining the closure of the set of subscriber interests—that is, a subscription that describes the union of all the channel's subscriptions.

Event channels, being consumers, also have a publish method. The implementation of publish in an event channel considers the new event in light of the existing subscriptions (and, perhaps, past events) and publishes that event (or some derivative of the event) to every consumer whose subscription matches the event.

In OIF, event producers can use not only the local arguments of their calls but also information from the thread's *environment* in deciding if a particular event is worthy of publication. OIF arranges to have the salient elements of this environment copied as part of the annotations of ordinary calls [5]. For example, a process could tag a particular call with some special symbol and recognize processes created as consequences of that call as retaining that symbol in their environment.

In OIF, every virtual address space has one globally (within that address space) well-known event channel. Any application or injector that has an event to report can publish to that event channel. Any consumer that wishes to receive events can subscribe to its local event channel. Event channels on different virtual machines can subscribe to each other. In this way, the publish and subscribe mechanism becomes distributed, while the most appropriate local decisions are made about whether to distribute an event.

How do the various virtual machines become aware of each other? OIF offered two different mechanisms. The first was to make the application was responsible for setting up the event channel network and arranging appropriate subscriptions among the nodes of that network. We are currently exploring the alternative, where information about the event channel structure is distributed by the framework as part of communications. This has an interesting "event horizon" event—knowledge of channels interested in the effects of an action would travel as fast as that action and its consequences.

Events

Event systems usually support one of two different kind of events. Most event systems define strongly-typed record-like event classes, where the class structure is globally known. In such a model, the manipulator of an event knows exactly which fields it has. Often the subscription language consists of merely describing interest in all records of a given class or subclass. This has the advantage of giving the programmer a reliable set of information on which to build—if I have an event of type t, I know it has fields x, y, and z. It has disadvantage of requiring too much commonly shared information, both in space and time. We do not want to demand that every event channel have knowledge of all kinds of events or even to posit the existence of an event definition repository. We expect the event structure to change, both as temporary event types are created to answer the questions of debugging and as new event types are created as part of the system evolution.

OIF takes the opposite tack. In OIF, events are property sets (name–value pairs), without system restrictions (or promises) as to the existence of any particular name–value pair in any particular event. OIF provides a marshaling mechanism for converting event structures to strings for transmission, and an unmarshaling mechanism for reinflating them back to the property–value pairs. Thus, the string event representation:

"userid: Fred; time: 12:40:18; type: error; message: read unhappy maknam"

would translate into an event object with four properties (userid, time, type and message) with the corresponding (string) values. The interpreter is responsible for doing data conversion for numeric operators. There are specific notations for strings that represent remote object references and values that are themselves events.

For debugging and system evolution, the property approach allows us to introduce new event fields into a running system. In terms of subscription languages, reference to the fields of events is straightforwardly uniform. This has the further virtue that no common understanding of event structure definitions is required across the distributed system. It has the corresponding disadvantage that we lack compile-time checks that structures will have properties not explicitly demanded in subscriptions.

Subscription languages

A goal of this work was to minimize uninteresting communications. Broadly, this suggests a richly expressive subscription language, where a subscriber can precisely describe which events are of interest. However, the richer the subscription language the more effort is involved both at coding time in creating the subscription interpreter and at run time in deciding if a particular subscription is satisfied by a given set of events. In OIF, we created a series of subscription languages of increasing expressiveness.[1] In OIF, we have four subscription languages: *propositional*, *predicate*, *temporal* and *agent*.

The **propositional language** deals solely in the existence of properties of events. A subscriber can express interest in A, B, and C, and any event that mentioned (as properties at the top-level) A, B and/or C matches.

The **predicate language** provides a way to refer to the values fields of events (and subfields of contained events), constants, and values from the environment; and to combine these values with relations (e.g., "less than") and propositional connectives (e.g., "or," and "not") to form a logical well-formed formula. Using a Cambridge-prefix syntax, a subscription matching error or warning events for user Joe would be:

```
(and (or  (= type 'error)
          (= type 'warning))
     (= user 'Joe))
```

The **temporal language** loosens the prior restriction to single events. The propositional and predicate languages reference a single event at a time and, as a default, forward that event to the consumer. The temporal language allows for expression of relations among several events. Thus, one can talk about the existence of events E1, E2, and E3, such that E1 has occurred before E2, which occurred before E3, and which share a common user. We use JESS [6], a RETE-based forward chaining, rule-based expert-system shell for our temporal matchin engine. In the temporal language, preceding subscription is

```
(event (time ?t1) (userid ?u1))
(event (time ?t2) (userid ?u2))
(event (time ?t3) (userid ?u3))
(test (< ?t1 ?t2 ?t3))
(test (eq ?u1 ?u2 ?u3))
```

To deal with the finiteness of memory, we guarantee only that the most recent *n* events will be available for matching and that new subscriptions might recognize old events.

The **agent language** carries the implication of the Cambridge-prefix form to its logical extension. Subscriptions are themselves programs, invoked by event occurrences and able to examine the local event repository. This is thus a mechanism for distributing agents throughout a system. Since we have not yet implemented an agent language, we have little to say about them except to note their existence at the top of the language hierarchy and their straightforward implementation with any of the standard Lisp-like interpreters.

In operational terms, the subscription of a subscribe method expects a string. The event channel parses this string with respect to the particular language. In our implementations, we used Cambridge prefix form as the grammatical substrate of the various subscription languages, as it is the simplest-to-parse recursive language.

---

[1] In this we are reminded of the hierarchies of automata, formal language grammars, and logics, where successively elements extend the expressibility of simpler mechanisms, often at the cost of greater complex computability. In practice, in both formal language theory and OIF, these structures are not always strictly hierarchical.

Event channel algorithms

We developed several algorithms to improve the efficiency of recognizing matching subscriptions: *sig*, *memo*, *lattice*, *compile* and *Rete*. (We have implemented all but the fourth.)

Sig, memo and lattice rely on recognizing the *signature* of subscriptions. The signature of a subscription, $\mathcal{S}(s)$ is set of event properties demanded by the subscription s. For example $\mathcal{S}$("(and (or (= type 'error) (= type 'warning)) (= user 'Joe))") is {type, user}. We call the properties mentioned in an event, e, the fields of the event, $\mathcal{F}(e)$. Both $\mathcal{S}$ and $\mathcal{F}$ can be represented with bit-vectors for fast subset comparisons.

Sig. Given event e, for each subscription, s, Sig checks that $\mathcal{S}(s) \subset \mathcal{F}(e)$ before evaluating s. Sig is a quick way of excluding certainly uninteresting events. Sig is appropriate for applications that generate a variety of different events and use computationally complex subscriptions.

Memo. For each unique set $\mathcal{F}$, Memo keeps a cache of those subscriptions for which $\mathcal{S}(s) \subset \mathcal{F}$. When another event with fields $\mathcal{F}$ arrives, Memo only needs to examine the subscriptions in the cache. On subscription updates, Memo can either examine the power set of the signature of the changed subscription, updating the corresponding memo values, or (in practice) clear the memo table. Memo is useful for situations where the subscription set changes slowly and events with the same fields occur repeatedly.

Lattice extends Memo with a notion of subsumption. That is, if $\mathcal{S}(x) \subset \mathcal{S}(y)$ and $\mathcal{S}(x) \not\subset \mathcal{F}(e)$, then $\mathcal{S}(y) \not\subset \mathcal{F}(e)$, and y could not match e. In general, the signatures of subscriptions form a lattice with respect to subset. The lattice algorithm constructs the (sparse) lattice as a data structure. The lattice algorithm works by "flattening" the lattice to a single path. Lattice handles subscription change more easily than Memo, and is most appropriate is when there is a lot of subsumption in the subscription structures.

Compile. Each subscription can be viewed programmatically: if the subscription condition is met, then perform the forwarding action. Compile treats the entire subscription set as a program by (1) sequencing the subscriptions, and (2) performing arbitrary compiler optimizations on the resulting program. In particular, elements such as common sub-expressions can be moved forward so as to be computed only once, tests such as (> x 3) can be placed so as to shadow (> x 7), and subsumptions can be realized by moving subsumed rules into the then-parts of more general subscriptions. Compile is most appropriate for a relatively static subscription set that contains a large number of common sub-expressions.

Rete. The first four algorithms deal with matching a single event to a single subscription. The temporal language matches multiple events to a subscription. In our implementation, we used the Jess implementation of Rete [7] for pattern matching.

Which is best? The optimal subscription channel algorithm is a function of the expected distribution of events and subscriptions. Some algorithms take advantage of an expected variety in the published events, while others do better on related or repeated event types. Similarly, the amount of effort expended when a new subscription is received can be worthwhile only given a particular frequency of subscription changes.

Applications

We have implemented the event channel mechanism described here in the OIF distributed computing framework, and applied it in a demonstration application [8]. That application implements a simulation of a distributed, competitive network management application. It uses injectors to achieve quality of service (i.e., real-time performance), manageability and security. It used the event mechanism to dynamically drive "inspector" user interfaces. The event mechanism also proved critical in debugging the application, particularly as the injector mechanism could be set to generate events on every remote invocation. Events could then be selectively scanned to get a trace of interprocess calls, and this trace could be transparently directed to both visible graphic user interfaces and textual logs.

In general, in OIF one can arbitrarily and dynamically modify the injectors of proxies or set the default behavior of a set of proxies to include a particular injector. By making an injector that generates trace events and applying that injector appropriately, the event mechanism can be made to track the patterns of interprocess calls in the system.

Related work

Event models. In the taxonomy of the Framework for Event-Based Software [2], OIF's event mechanism uses point-to-point, application-to-application communication. Modules have no explicit specification of

their interfaces. It supports dynamic system modification and allows fully abstract naming. Our publishers are Barrett's informers; our consumers, listeners; and our event channels, routers. The subscription mechanism effectively serves to do message transformation. We posit no delivery constraints beyond the underlying distributed object framework. The local event channel on each virtual machine serves as a group. Rosenblum and Wolf [11] describe a seven-model framework for event observation and notification. Within that framework, our publishers are the invoker objects and subscribers are the objects of interest. Events are the explicitly generated by invoking the send event action, naming is implicit in the naming of event fields (the property-based model), observation is by explicit subscription, information is by the action of a subscription, pattern abstraction and filtering is by the pattern part of the subscription language, and the partitioning arises naturally from the set of subscriptions made. We have no explicit time model, notification is by distributed object technology calls, and the resources for sorting through subscriptions are provided by the sender and the intermediary event channels.

Event implementations. Bates [3] argues for using a rule-based publish and subscribe system to debug heterogeneous, distributed systems. Primitive events are defined and source code is annotated so that the executing program generates event instances. Bates also uses a rule-based engine for complex event detection, fairly similar to Rete, though independently discovered.

The Elvin project is a publish-subscribe service that delivers notifications on the basis on the event's content [12]. It has an event subscription language that allows subscribers to place some constraints over the notifications, flexible definition of events that allow developers to define events as required, dynamic definition of event types, and allows the creation of new events based on old events. Elvin also introduces the idea of *quenching* that "allows event producers to receive information about what consumers are expecting of them so that they need only generate events that are in demand." In contrast to Elvin, which has a single centralized event channel, OIF's event channels are distributed.

The Ariadne Debugger in TAU stores an execution history graph of events and allows the subscriber to specify patterns using a simple subscription language that is capable expressing temporal relations among several events but unable to express other simple prepositional or complex relations among events [13]. To "compensate" for where the language lacks, Ariadne "provides a scalable, spread-sheet like interface for exploring match trees."

The CEDMOS project architecture is composed of event-producers and event-consumers that are connected through event-transformers. "The event transformers convert streams of incoming events into different streams of events, which are ...of interest to the event-consumers" [1]. To facilitate the event transformers, a graphical tool facilitates the definition of complex event from simple events.

Brant and Kristensen apply events to web-based notification. Their architecture includes the notions of annotated lists, a well-worked-out datatype mechanism and a good implementation of filtering [4]. Intermetrics [10] describes a design for applying events to doing debugging of distributed, components. Luckham and Frasca apply event patterns, causal histories, filtering and aggregation to provide higher levels of abstractions for managing distributed systems [9].

Summary and discussion

We have discussed the publish and subscribe mechanism in the Object Infrastructure Framework. This mechanism has proved to be a powerful tool in debugging and managing distributed systems, supporting functions such as fault diagnosis, intrusion detection, performance analysis, and accounting. Key elements of this work are the ability to inject event generators into existing components existence within a framework that provides a continuing environmental context, the use of unstructured events, rich subscription languages, and selectable and efficient algorithms for subscription resolution. Topics for further work include (1) subscription-forwarding mechanisms that do not require tree-like branching, (2) security mechanisms for subscriptions and event channeling (including the ability of an event generator to limit who could notice his events), (3) quantifying the actual performance of different event-channel algorithms in realistic cases, (4) implementing agent subscription languages, and (5) implementing subscription compilation.

The ideas expressed in this paper have emerged from the work of the MCC Object Infrastructure Project, where Dr. Filman was on assignment from Lockheed Martin Corporation and have been further developed at NASA Ames. We thank Stu Barrett, David Filman and Ted Linden for discussions on this subject and Cecilia Aragon, David Korsmeyer, and Tarang Patel for their input on drafts of this paper.

References

1. Baker, D., Cassandra, A., Rashid, M. CEDMOS: Complex Event Detection and Monitoring System. Microelectronics and Computer Technology Corporation, 1998.
2. Barrett, D. J., Clarke, L. A., Tarr, P. L., and Wise, A. L. An Event-Based Software Integration Framework. *ACM Transactions on Software Engineering and Methodology 5*, 4 (October 1996) 378–421.
3. Bates, P. C. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *ACM Transactions on Computer Systems 13*, 1 (February 1995), 1–31.
4. Brandt, S. and Kristensen, A. Web Push as an Internet Notification Service, W3C Workshop on Push Technology, (Boston, Massachusetts, September 1997), http://keryxsoft.hpl.hp.com/doc/ins.html.
5. Filman, R. E., Barrett, S., Lee, D. D., and Linden, T. Inserting Ilities by Controlling Communications. To appear in *CACM*. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/oif-cacm-final.pdf
6. Friedman-Hill, E. J. Jess, The Java Expert System Shell. DANS98-8206 Distributed Computing Systems Sandia National Labs.,. Livermore, CA, (September 1998), http://herzberg.ca.sandia.gov/jess
7. Forgy, C. L. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence 19*, 1 (1982) 17–37.
8. Lee, D. and Filman, R. Verification of Compositional Software Architectures. Workshop on Compositional Software Architectures, Monterey, California, January 1998. http://www.objs.com/workshops/ws9801/papers/paper096.doc.
9. Luckham, D. C. and Frasca, B. Complex Event Processing in Distributed Systems. Stanford University Technical Report CSL-TR-98-754 (March 1998), ftp://pavg.stanford.edu/pub/cep/fabline.ps.Z
10. Ress, J. Intermetrics' Owatch Debugging Technology for Distributed, Component-Based Systems. OMG-DARPA-MCC Workshop on Compositional Software Architectures (Monterey, California, January 1997). http://www.objs.com/workshops/ws9801/papers/paper058.html.
11. Rosenblum, D. S., and Wolf, A. L. A Design Framework for Internet-Scale Event Observation and Notification. *Proceedings of the Sixth European Software Engineering Conference/ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering* (September 1997), 344–360.
12. Segall, B. and Arnold, D. Elvin has left the building: A publish/subscribe notification service with quenching. *Proceedings of AUUG97* (Brisbane, Queensland, Australia, September 1997).
13. Shende, S., Cuny, J., Hansen, L., Kundu, J., McLaughry, S., and Wolf, O. Event and State-Based Debugging in TAU: A Prototype. *Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools* (Philadelphia, May 1996), 21–30.
14. Stallings, W. *SNMP, SNMP-2, and CMIP: The Practical Guide to Network-Management Standards*. Reading MA: Addison-Wesley 1993