Event Webs for Crisis Management

K. Mani Chandy, Brian Emre Aydemir, Elliott Michael Karpilovsky and Daniel M. Zimmerman
Computer Science 256-80
California Institute of Technology
Pasadena, California, USA
{*mani, emre, elliottk, dmz*}*@cs.caltech.edu*

ABSTRACT
Crises are frequent and varied, and may have profound effects on individuals and organizations. Effectively managing crises, regardless of their scope, requires the ability to quickly process potentially large amounts of information about a rapidly changing world.

This paper presents a software architecture, the *event web*, that can be used to help effectively manage crises. An event web allows for the specification, deployment, observation and management of large numbers of persistent software objects that generate, process or consume streams of events. The system is designed to be independent of any particular programming language or hardware architecture.

KEY WORDS
information systems and the Internet, information infrastructure, software agents

1 Introduction

Crises are frequent and varied, and may have profound effects on individuals and organizations. Disasters such as earthquakes and bioterror attacks threaten people's lives. Market events such as large stock price fluctuations and corporate scandals threaten the stability of corporations. A personal crisis may merely threaten an individual's peace of mind. Effectively managing crises, regardless of their scope, requires the ability to quickly process potentially large amounts of information about a rapidly changing world.

A task force that manages a crisis is a fluid coalition of individuals and institutions including government agencies, non-governmental organizations and corporations. The public, increasingly equipped with communications technology such as email and web-enabled phones, can play a valuable role in dealing with crises by participating in such a task force. Since the membership of a task force is often not known until a crisis strikes, the underlying computing and communication systems available to the task force may not be known until the task force is formed. Moreover, since membership in a task force can change over time, the availability and structure of such computing and communication systems may change during the life of a task force.

In this paper we present a software architecture, the *event web*, that can be used to help task forces effectively manage crises. An event web allows for the specification, deployment, observation and management of large numbers of persistent software objects that generate, process or consume streams of events. This architecture can provide task forces with the ability to dynamically sense and respond to conditions that arise as crises unfold.

The event web architecture is designed to be as independent of programming languages and computing platforms as possible, making extensive use of platform-neutral technologies such as XML. Using this architecture, a crisis-management infrastructure can be implemented across multiple runtime systems, such as Java and .NET, so that each component can take advantage of appropriate underlying technology while still interoperating with the rest of the components in the event web.

We first describe the event web architecture, providing an overview of its various parts and discussing the architectures of these parts individually. We then detail the execution model for software components within an event web and describe the application of event webs to crisis management. Finally, we compare event webs to some related architectures and concepts.

2 Event Web Architecture

An *event web* is an infrastructure that enables the dynamic creation, modification and management of large numbers of persistent reactive objects that process events. An *event* is a message generated by an object, describing an aspect of the system's state or history. Examples of events are the current temperature at a specific geographic location and the current price of a particular stock on the Nasdaq. Each event in an event web is an XML document, which allows implementations of the event web architecture to be written in any language for which an XML parser is readily available.

An *event stream* is a sequence of events generated by an object, describing the evolution of a particular aspect of the system's state over time. Examples of event streams are the sequence of daily rainfalls at a single geographic location and the changes to a stock's price throughout a trading day. Each event stream is specified by a function from system histories to event sequences. In practice, implementations of event streams may be approximations to these functions because determining accurate histories of distributed systems is difficult.

The event web architecture consists of four parts: the set of persistent reactive objects, called *event processors*, that process event streams; the dissemination network that distributes events among these objects; a directory that describes event streams and requests for them; and a service layer that provides various services to the objects in the system. We now describe the four parts of the architecture in detail.

2.1 Event Processors

An *event processor* is an object in the event web that may receive and process event streams and may generate new event streams. Each processor is described by an XML document in a persistent store. The description of an event processor includes information about its implementation and its current state (described in Section 3.1). Event processors are classified according to their behaviors with respect to event streams.

An *event generator* is an event processor that generates event streams. Each event generator monitors the environment in which the event web operates and generates an event stream that describes some aspect of that environment. This can be done in various ways: an event generator may have a sensor that feeds it information about the physical world, may be a passive recipient of information sent over a network (such as electronic mail) or may actively poll information sources to obtain current data about the environment. One example of an event generator is a sensor in a building security system that generates events when it detects movement in a restricted corridor; another example is an object that periodically polls a competitor's online store and generates events when price changes occur.

An *event consumer* is an event processor that receives event streams from event generators. An event consumer performs actions based on the events it receives, such as executing specific applications or sending SMS alerts to specific people when certain events arrive. One example of an event consumer is an object that sends email to a manager when it receives events that indicate significant differences between planned and actual expenditures by one of the employees she supervises; another example is an object that sounds an alarm when it receives events that indicate a dangerous level of radiation in a particular room.

Many event processors act as both event generators and event consumers, generating new event streams by integrating and processing event streams from other objects in the event web. An example of an event processor that behaves in this way is an object in a commodity trading system that receives some time-series prices of commodities and some currency exchange rates and uses that information to generate a new event stream indicating changes among buy, hold and sell recommendations.

Event processors can be added to and removed from the event web while the system is running, changing both the set of available event streams in the web and the set of recipients for each event stream. The distribution of event streams to event processors is handled by the dissemination network, discussed in the next section. The operation of event processors also depends on the service layer, discussed in Section 2.4. The execution model for event processors is described in Section 3.

2.2 Dissemination Network

The event dissemination network sends a copy of each event published by the event generators to every event consumer interested in that event. A consumer expresses its interest by *subscribing* for events with particular characteristics; the consumer specifies criteria, in the form of predicates, that must be satisfied by the events. These predicates are specified within an XML document, using expressions in XPath or some other appropriate XML query language; these expressions can then be matched against generated events in a platform-independent way. The dissemination network, which is essentially a content-based publish/subscribe system [7], uses the subscriptions to deliver events to appropriate consumers.

For each event consumer, the dissemination network has an associated set of named input event queues. Every subscription is associated with exactly one of these queues. The dissemination network appends a copy of a published event to an input event queue if and only if the event matches a subscription associated with that queue. The event consumer can assign a priority level to each of its associated event queues to determine the order in which it receives incoming events. A special priority level, *off*, indicates that the event consumer does not want to receive events from a particular queue; events continue to accumulate in a queue while it is turned off and are delivered to the consumer when (if) it turns that queue back on.

In order to make it easier to reason about the correctness of computations in an event web, the formal model for the dissemination network is kept extremely simple. The planned implementation, which will perform various filtering and routing optimizations to avoid wasting network bandwidth and router processing power, is far more complex and is not discussed here.

In the formal model, there is one event channel connecting each event generator to each input queue in the web. Each input queue has an associated set of subscriptions, which are predicates on the contents of events. The state of the network at any given time is comprised of the contents of the input queues, the set of subscriptions associated with each input queue and the states of the event channels between the event generators and the input queues.

The following four operations can cause changes to the state of the dissemination network: a new input queue can be created and registered with the network; the set of subscriptions associated with an existing input queue can be changed (by adding or removing subscriptions); an input queue can be deregistered from the network, which also causes its set of subscriptions to be removed from the network; and an event generator can send an event, which atomically adds one copy of the event to each event channel associated with that event generator.

An event placed in an event channel takes arbitrary finite time to arrive at its destination input queue. Once it arrives at the destination queue, it is either inserted at the tail of the queue or discarded, depending on whether or not it satisfies any of the subscriptions associated with the queue.

The state of an event channel is a queue containing the events in the channel, with the least recently sent event at the head and the most recently sent event at the tail. All changes to the state of an event channel are atomic; for instance, the enqueuing of an event on an event consumer's input queue and its removal from the channel occur in one atomic operation.

2.3 Event Directory

The directory in an event web enables individuals and event processors to advertise characteristics of event streams that they generate or need. Individuals browsing the directory may find descriptions of event streams that enable them to provide additional information, by creating new event processors to perform some computation using those existing streams. Similarly, they may find requests for event streams with specific characteristics in the directory and respond to such requests by creating new sensors or event processors to satisfy them. Thus, the set of event streams in the event web becomes better suited to the needs of users over time.

2.4 Service Layer

The service layer in an event web provides various services to event processors. In the current architecture, these services are limited to the creation of new event processors and the handling of timeouts. The state of the service layer is given by a set of pending service requests. Whenever the set of pending service requests is non-empty, the layer removes one request from the set of pending requests and handles it; this is an atomic operation.

The service layer is modular, which allows new services to be created and added to the architecture as they become necessary. The use of services is discussed in more detail in the next section, as part of the description of the execution model for event processors.

3 Event Processor Execution Model

An event processor is an event driven object that executes a sequence of *transitions* over its lifetime. We first discuss the internal structure of an event processor, then detail exactly what is meant by execution of a transition and briefly describe some implementation considerations.

3.1 Internal Structure

The state of an event processor is specified by the values of its variables, which include the priorities assigned to its input queues and a timeout value. Each event processor has a state transition function, an event generation function and a service request function. These map an incoming event and the current state of the event processor to the next state of the processor, a (possibly empty) sequence of events generated in response to the incoming event and a (possibly empty) set of service requests, respectively.

Event processors interact with each other only by publishing events and receiving events for which they have subscribed. At any given time, the next event to be processed by an event processor comes from the input queue with the highest (non-off) priority in the processor's current state. The event processor may always receive events from the service layer, regardless of the priorities the processor has assigned to its input queues.

An event processor's timeout value is used by the service layer. The event processor can request that the service layer send it a timeout event when no other events have arrived for it within the timeout period.

3.2 Transition Execution

An event processor begins a transition by receiving an event. It applies its state transition function, event generation function and service request function to this event and its current state to determine its new state, the sequence of events that are published as a result of the transition and the set of service requests it makes. It then updates its state and sends the resulting events and service requests.

Each transition occurs as an atomic operation. The executions of the state transition, event generation and service request functions are expected to terminate. If any of them do not terminate within a reasonable period (determined by the event web), the processor may be removed from the system.

3.3 Implementation Considerations

In practice, the event web may contain a multitude of event processors distributed across multiple computing systems. The event web runtime on each system may choose to load an event processor into memory from its description in persistent storage only when the processor can execute a transition, then write the processor's description back to persistent storage upon the transition's completion. To accomplish this, manager processes on each system monitor the input queues and service layer events associated with event processors and determine when it is necessary to load them from and save them to persistent storage. There is a performance penalty incurred by writing to persistent storage for every state transition, but the advantage is that a single runtime can support a much larger number of event processors than it can load into memory at one time.

A key aspect of the event web is that users provide specifications, but not implementations, of event processors. A program in Java or C# is an implementation, while a description of a state-transition machine is a specification. Event processors are specified using XML and their transitions may be implemented by any means supported by the event web runtime; currently, XSLT transitions and Java transitions are supported.

An XSLT transition specifies a state transformation from a combined XML representation of an incoming event and an event processor's current state to a combined XML representation of the event processor's next state, generated events and service requests. XSLT transitions can be executed on any platform with an XSLT engine.

A Java transition specifies a state transformation as a method of a Java class named in the XML specification of the event processor, which takes as parameters the incoming event and the current state and returns the next state, generated events and service requests. Java transitions can be executed on any platform with a sufficiently modern Java virtual machine.

XSLT transitions are substantially more difficult to implement than Java transitions, because it is hard to specify generalized state transformations in terms of textual modifications. They are also generally slower to execute than equivalent Java transitions. However, they have the advantages of being more platform-independent and of relying only on textual transformations to manipulate the state of an event processor and interact with the event web.

4 Application to Crisis Management

Crises occur unexpectedly, and task forces that deal with them must mobilize quickly. Task force members who are not information technology specialists must be able to create systems that allow them to quickly process incoming data about the crisis situation.

The event web allows task force members to specify conditions on the history of the system and actions to be taken when those conditions hold. The pair consisting of a condition and its corresponding action is called a *when-then rule*. For the purposes of this work, the actions are limited to the publishing of appropriate events. Two example when-then rules are, "when the history of temperature readings for the reactor core indicates that the temperature has increased more than 20°C in the last 15 minutes, then send an alert event," and, "when the number of units of type O blood in the hospital drops below 50, then request more type O blood from the county blood bank." More formally, the condition of a when-then rule is a predicate on the history of the system and the action is a description of the event that will be sent when the predicate holds.

A when-then rule is implemented by one or more event processors. When a rule is submitted to the event web, a rule construction object decomposes the rule's predicate into its component clauses. For each clause, a specification for an object that can evaluate the clause is located and used to construct an event processor, which is then added to the event web. Input queues with appropriate subscriptions are created for each new event processor. For instance, the "when" condition of the second example when-then rule could be expressed as "number of units of type O blood < 50." This would be implemented by a single event processor that compares a number received in its input events to 50. It would have a single input queue, subscribed to events reporting the number of units of type O blood in the hospital, and would send an appropriate output event when it received an input event with a number of units less than 50.

Implementing when-then rules for conditions in a distributed system is difficult, because determining the state of a distributed system is like taking an instantaneous global snapshot [5] of a machine with many moving parts. A system's history cannot be determined precisely because, without perfect synchronization of their clocks, all components cannot agree upon a universally accepted time. Therefore, the event web only implements an approximation to when-then rules.

5 Related Work

Content-based publish/subscribe systems use intelligent routers to direct messages to destinations based on their content. Examples of such systems are IBM's Gryphon [1, 2], the University of Colorado's SIENA [3] and Eurecom's XNet [4]. In some content-based systems, messages are routed based on associated headers containing properties and values; in others, they are routed based on direct examination of their content. The latter method of routing is similar to the dissemination system of an event web, which matches events to subscriptions throughout the system. Of the publish/subscribe systems mentioned, the dissemination system of an event web is closest to XNet, because both are designed to handle XML-based messages with subscriptions specified as XPath expressions. We intend to leverage the work done on XNet to optimize the implementation of the event web architecture.

Rules engines [6, 8, 9] enable efficient executions of chains of rules to determine whether complex predicates hold on a system. An event web could be considered a distributed rule engine, where event processors evaluate predicates on incoming events and generate new events based on those evaluations. However, event webs do not target the same application space as rules engines; while rules engines typically deal with static rules evaluated on possibly dynamic data sets, event webs are intended to evaluate rapidly changing sets of subscriptions in a distributed system where the set of event generators and the types of events generated may also exhibit frequent changes.

Web services, which also use XML as their core technology, typically involve synchronous request-respond interactions, using protocols such as SOAP over transports such as HTTP, between a client requesting a service and a server responding to the request. An example Web service is a pricing database offered by an electronic bookseller that responds to a request for pricing information about a particular book by providing the current price. Interactions within an event web, by comparison, are based on asynchronous event streams. A bookseller connected to an event web would publish streams of events describing changes to pricing information, and these events would be received by all interested subscribers. The usage of directories in Web services and event webs is also substantially different: directories in Web services tell clients how to request particular services and what types of responses to expect, while directories in an event web describe the characteristics of available event streams.

Commercial sense and respond systems, such as iSpheres Halo™ [10], are designed to detect changes in the state of distributed data and alert users and applications to these changes. An event web is essentially a sense and respond system, but with a more dynamic architecture than existing commercial applications; while commercial sense and respond systems are designed to operate as part of relatively static enterprise computing systems, event webs are designed to support a large and rapidly changing set of users and systems.

6 Conclusion

Crises are inherently unplanned, which implies that information systems that assist in crisis management must be created quickly when the need arises and modified as requirements change. The event web architecture is appropriate for such systems: event processors can be added to and removed from an event web as necessary to accommodate the size and scope of the task force and the crisis being managed; the dissemination network ensures that event consumers receive only the events they need and that event generators need not explicitly send their events to particular consumers; and the event directory allows for quick discovery of the types of information available in the system. The structuring mechanism of when-then rules allows task force members, who are not generally information technology specialists, to easily specify conditions of interest and actions to take when those conditions occur.

The event web architecture is designed to be as independent of specific hardware and programming environments as possible, through the utilization of cross-platform standards such as XML and XSLT. We expect the architecture to be amenable to the application of standard techniques for ensuring reliability and fault tolerance in distributed systems.

Conventional techniques are superior to event webs for unchanging applications deployed within a single organization, such as an insurance company whose procedures for administering claims remain unchanged over many years. However, event webs are superior to conventional techniques in dynamic situations, such as crises, where a system must be created rapidly, reconfigured continuously and distributed across multiple organizations.

References

[1] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 53–61. ACM Press, May 1999.

[2] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 262–272. IEEE Press, June 1999.

[3] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[4] R. Chand and P. A. Felber. A scalable protocol for content-based routing in overlay networks. In *Second IEEE International Symposium on Network Computing and Applications*, pages 123–130. IEEE Press, April 2003.

[5] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining the global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.

[6] Fair Isaac Corporation. Fair Isaac decision tools. `http://www.blazesoft.com/`.

[7] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[8] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[9] ILOG. `http://www.ilog.com/`.

[10] iSpheres Corporation. Halo™. `http://www.ispheres.com/`.