Using Events to Build Large Scale Distributed Applications

Richard Hayton, Jean Bacon, John Bates and Ken Moody
University of Cambridge Computer Laboratory
Pembroke Street, Cambridge CB2 3QG, United Kingdom

Abstract

We have extended an Interface Definition Language to handle event registration and notification. Clients register interest in specified classes of events, and servers then notify them of any occurrence asynchronously. Event occurrences are identified by parameters which conform to IDL typing constraints and can therefore be used in synchronous method invocations. Methods to handle registration and notification are generic and can be inherited by objects of any class: as a by-product of IDL processing the stubs to handle event creation and decoding are generated automatically. We have implemented a prototype composite event recogniser based on nested finite state machines and have defined an event algebra and language to specify composite events.

The approach is inherently scalable in that only events in which an interest has been registered are notified. Alternative approaches lead to polling, mining for event data or being flooded with superfluous events.

1 Introduction

The term "event" is used in many spheres of computing to capture the notion of an autonomous asynchronous occurrence. Here, we are concerned with events within an object-orientated distributed programming (OODP) environment and have designed a unified framework for programming with objects and events.

Current OODP technology is based on objects with typed interfaces together with support for providers to advertise them, and for users to locate and invoke them [1, 2]. This object-orientated paradigm is well established but does not capture fully the dynamic, event-driven behaviour of many systems. Some "active" objects monitor continuously for specific occurrences and may then notify interested clients asynchronously. We believe that applications will, increasingly, need to request and receive notification of asynchronous occurrences (events) as well as carrying out synchronous service invocation.

1.1 Emerging Classes of Service

The list below introduces some types of service which are emerging in distributed environments along with the applications which access them:

- *Multimedia Services:* Applications such as Hypermedia can require access to components from different services, e.g. those supporting video, audio, text or pictures. User interaction with one distributed medium component can cause an application to initiate actions on another. For example, clicking on a marked area of a video presentation may cause it to be halted and an audio commentary output.

- *Cooperative Systems:* Cooperative applications support collaboration between a number of users. Events in one application instance must be reflected in others.
- *Services involving Mobility:* e.g. services which utilise an 'Active Badge' system. Each user is provided with a mobile badge device which periodically transmits a unique infra-red signal. Distributed detectors report people's locations to monitoring services. Applications must be able to request and receive information such as when people's locations change.
- *Agent Architectures:* Agents are services which apply "intelligent heuristics" when reporting on their monitoring activities. An example is reporting that mail has arrived only if it is from someone interesting. We envisage agents which apply heuristics to notifications from a range of services.
- *Active Database Systems:* i.e. a standard DBMS augmented with facilities to monitor for certain sequences of database activity. Uses of such monitoring include security checks and constraint enforcement. Standard object invocations by some clients trigger events of interest to others.
- *Telecommunications Monitoring and Management:* Network management systems require monitoring for fault and overload conditions. Condition notifications must be controlled.

All of these service types are 'active' and 'continuous'. They not only carry out user requests but also monitor concurrently for certain events. Several exist for the specific purpose of monitoring. Applications which use these systems must be informed of event occurrences which they can then use to initiate further actions.

A single application will often be interested in events from a number of sources. An architecture is required in which basic services carry out standard event detection and notification allowing additional functionality, such as event composition, to be provided by higher level services.

1.2 Our Event Approach

We believe that many applications are developed 'from scratch' because appropriate monitoring facilities are not provided by existing services. To redress this we propose a standard extension to services to incorporate **event specification** (by services), **registration** (by clients at services) and **notification** (by services to clients).

Some applications will require notification of events from multiple services and will need to detect specific patterns of composite event occurrence from these different sources. If services are built to conform to our proposed standards, their clients can compose events into expressions. Since composite event detection is needed in many applications we have built a **composite event service** to meet this requirement. We have also designed a language to specify composite events.

2 An Event Architecture

2.1 Event classification

In our architecture, events are specified using an extended RPC interface definition language (IDL). Like RPCs, events are of distinct types, and may have a number of parameters of simple or complex types. This combination of RPC and Event specification simplifies naming issues, and allows existing Trading mechanisms to be used for the location of event servers of a particular type. In addition, it is common for services to combine RPC and event interfaces, which simplifies the passing of parameters between the two domains. For example the interface to a print server is:

```
Printer: INTERFACE =
BEGIN
 Print : OPERATION [ file : STRING ]
                   RETURNS
                   [ jobno : CARDINAL ];

 Finished: EVENT   [ jobno : CARDINAL ];
END.
```

The IDL file is preprocessed to produce client and server *stubs* for the operations, and to create *constructors* for the events. The event constructor creates a *generic event object* representing an instance of the event of the given type. This may then be signalled or manipulated using a set of standard utilities that need not be aware of the concrete type of the event. Destructors are also created in the form of access methods to un-marshall the parameters of an instance of an event of a given type. The constructor and destructor for the 'Finished' event above are as follows

```
Event *Printer_Finished(int jobno)
```

and

```
int Printer_Finished::jobno()
```

2.2 Registration and Notification

In order for a client to be notified of the occurrence of an event, it must have first *registered interest* in the event with the issuing service. The alternative scheme is for all events to be broadcast to all possibly interested parties. Although suitable for a centralised system, such an approach clearly does not scale to a distributed environment. Registration has other benefits. Monitoring need not take place continuously, but only when a client is interested. For many events registration can take place as a side effect of some other operation.

Registration takes place in two stages. Firstly, a client registers with the service establishing a session, and supplying a delivery address for event notification. During this process, the client may provide various credentials, and *admission control* based on these will take place.

Once a session has been established, the client may register interest in event occurrences of different types. In order to reduce the number of *uninteresting* events that the client is notified about, the client supplies an *acceptance expression* that will be satisfied by interesting events, but not by uninteresting ones. The format of this expression is significant. A complex acceptance expression might reduce the number of uninteresting expressions that are notified, but it is less likely that the event server will be able to derive useful information that will allow it to restrict the amount of unnecessary monitoring. Complex acceptance expressions can also place an additional load on the event generating service. For general expressions, the matching cost is linear in the number of registrations. In large scale applications, this may be unacceptable. The form of acceptance expression chosen is an *event template*. This is an instance of an event with a number of wild card parameters. (cf. query by example). For the printer example, registration using the template

$$\mathbf{Finished}(27)$$

would lead to notification when job 27 was complete, whereas registration using

$$\mathbf{Finished}(*)$$

would lead to notification whenever any job completed.

Templates are a simple but powerful mechanism. In particular, they are amenable to automatic generation, which is required for composite event detection, and matching against templates is an $\mathbf{O}(log(no.\ of\ registrations))$ cost.

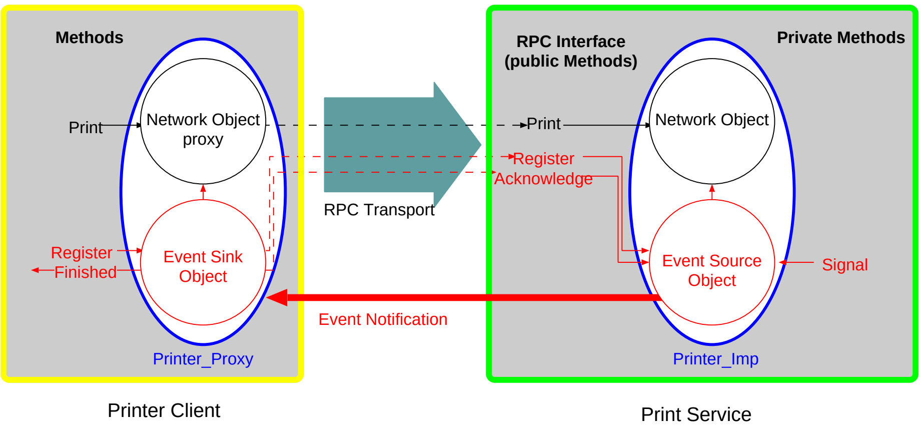

Figure 1: Client and Server Objects in a Print Server

The event service naturally extends the semantics of remote object invocation used in modern OODP environments. The details of the implementation of event notification are hidden by the EventSrc and EventSnk objects giving a useful abstraction similar to that provided by RPC libraries.

2.3 Library Support

As events are marshalled into *generic event objects*, a library of utilities can be created to manipulate them, and in particular event services, such as composite event servers and event multiplexors, need not understand the concrete type of the event instances they manipulate. Libraries have been created to deal with both client and server action on event notification, template matching, event trading and failure tolerance. In addition the client and server libraries interact using a secure protocol based on roles[1]. As events are integrated into the RPC system, a server *implementation object* will inherit a `Signal` method and a client *proxy object* will inherit `Register` and `Callback` methods. Figure 1 illustrates the objects involved in a session between a Print service and a client of that service, and figure 2 gives example code for the client and server.

3 Composite Events

Clients of event services typically perform monitoring functions: for example, monitoring the location of a person, or counting the number of people in a meeting. Monitoring applications are generally interested in particular sequences of events, rather than the occurrence of a single event.

Although each application could be designed in an ad-hoc manner, it is advantageous to design a general purpose composite event recogniser. There are several existing architectures for composite event recognition [5, 6], but these were designed for centralised event systems and are unsuitable for distributed event recognition.

We outline the design of a composite event language in section 4.

[1]The security architecture is described in another paper at this workshop [3], and a description of the event extensions can be found in [4, chapter 7].

```
void print_document(Printer *p,
                    char *filename)
{
  int jobno;
  Event *e;

  jobno = p->Print(filename);
  e = new Printer_Finished("",jobno);
  p->Register(e,(void*) filename);
}

void Printer_Proxy::Finished(Printer_Finished *e,
                            void *state)
{
  int jobno;
  char *filename = (char*) state;

  jobno = event->jobno();
  printf("File %s (job %d) has been printed\n",
          filename,jobno);
}
```

(a) A client of the Print Service

```
int Printer_Imp::Print(char *filename)
{
  int jobno = add_to_queue(filename);
  return jobno;
}

void mainloop(Printer_Imp *ptr)
{
  for(;;)
  {
    JobInfo *j;
    Event *e;

    j = get_from_queue();
    do_print(j->file);

    e = Printer_Finished("",j->jobno);
    ptr->Signal(e);
  }
}
```

(b) The Print Service itself

Figure 2: A complete Example

In `print_document` a printer client executes the `Print` method on the remote server object and then registers interest in the completion of the job. The printer service maintains a queue of jobs, and after completing each, signals the completion as an event. When this client's job is complete, the event is signalled and (ultimately) the client's callback method `Finished` is invoked. The client interrogates the event to discover the job number and prints a message.

Note that the client may provide an additional piece of state at registration time that is returned along with any matching events. In this example this is used to maintain a pointer to the filename associated with the print job.

3.1 Distributed Time

In a distributed environment time is the key consideration. There is, in general, no *global clock* and care must therefore be taken when comparing time stamps on events from different sources. Distribution also leads to *variable delay*. If we are interested in determining which of two events `A` and `B` occurred first, we cannot rely on the order in which we are notified, as notification of either event may be delayed. We must therefore wait to see if notification has been delayed, or determine this by other means, such as the receipt of a periodic checkpoint or *heartbeat*[2]. Together with the increased scale and complexity of a distributed environment, these features make it difficult to take a *global view* of the system. In general we can only ensure that we have received all events up to time t after we have received events from the most delayed event source; i.e. at time $t + \Delta_{\text{worst}}$.

Evaluation models that require a global view will therefore have an inherent delay of Δ_{worst}. Unfortunately the majority of existing composite event detection schemes rely on such a simplifying assumption. An important motivation when designing a composite event detector for distributed events was to reduce the requirement.

For example, in the badge system we may wish to detect sightings of two users Giles and Roger, in order to detect if Giles ever enters a room in which Roger is situated. This could be represented by the expression

[2]Described in the companion paper [3]

```
Whenever Seen(Roger,x)
   Repeat
     If Seen(Giles,x) then Signal Together(x)
   Until Seen(Roger,y)
```

Consider the situation in which Roger and Giles are first seen in room T14, and then seen in room T15. If the events from the badge sensor in room T14 are delayed, we do not want this to disrupt the detection of the second meeting. An optimal detector would process events in the order in which they were received, and would detect the second meeting independently of the first: a 'global view' detector, which must process events in the order in which they occurred, would block and (eventually) detect the first meeting first. Note that both evaluations ultimately return the same results. A secondary issue is that by enforcing a global view, we require a total order on all event instances. The operators in our language require only pairwise comparison of (some) event time stamps, and this lesser requirement is easier to justify in a wide-area environment where clocks are not completely synchronised.

4 A Composite Event Language

A composite event specification language based on modifications to regular expressions is given in detail in [4]. It is of equal expressive power to regular expressions, but has the advantages of clarity and use of parameters. In addition it has an efficient evaluation machine and may easily be extended to deal with operational criteria, such as network delay, that might lead to the misordering of events.

A composite expression is constructed from *base event templates* using a set of operators {*without, sequence, inclusive or, whenever*} described below. The parameters to these templates may be literals or variables. An evaluation Φ of a composite expression C is defined in terms of the start time of the evaluation s, and an initial set of variable bindings given in an environment E. An evaluation returns a set of tuples of (*occurrence time*, *environment*). *occurrence time* is the time at which the composite event triggers, and *environment* is a set of (*variable name*, *value*) pairs. A composite event may trigger many times, with the same or different occurrence times.

The operators were constructed so that evaluation of different parts of an expression can take place *independently*. That is to say, delays affecting evaluation of one sub-expression need not affect the evaluation of different parts of the expression. This reduces the need for a global view, and helps reduce the impact of delay or network failure.

Without (A-B) This is a limited form of negation. $\Phi(A, B)$ returns the set of occurrences where an **A** is triggered without an earlier **B**.

Sequence ($A; B$) This is a straightforward sequence. However, unlike regular expressions, this expression matches any sequence of **A** followed by **B**, regardless of intervening events. This is more natural semantics in a distributed environment where there are many unrelated events.

Or ($A|B$) This returns the set of occurrence times when either of **A** or **B** have occurred.

Whenever ($\$A$) This is the most powerful operator in the language and is used to represent multiple independent evaluations. Informally $\$\mathbf{A}$ returns the set of occurrence times when **A** is satisfied. Each evaluation has a separate environment so, for example, $\$\mathbf{Born}(x)$ will trigger whenever someone is born, with x instantiated to their name, and $\$\mathbf{Born}(x); \mathbf{Die}(x)$ will trigger whenever someone is born and then that person dies.

5 Implementation

The composite event language was designed to have an efficient implementation using a 'push down' finite state machine. An expression is represented by a sequence of states, where each state evaluates a sub-expression using a sub-machine. Each of the operators in the language can be represented directly by a state, and there is therefore a straightforward correspondence between an expression and the machine used to evaluate it.

An evaluation consists of a 'bead' traversing the machine. Each bead carries with it an environment, and beads may 'split' when appropriate to give branching evaluation. Beads are evaluated independently and this removes the need for a global view of event instances. However the relationship between beads is maintained, so that they may be destroyed when no longer required. When a bead enters a state it is 'pushed down' to the sub-machine and leaves a record behind detailing its current environment. During evaluation of the sub-expression the bead may be destroyed, or may split any number of times. When any of these beads return, the type of the statement determines whether the bead should proceed or if some other action should be taken. For example when a bead enters the state for a **A** *without* **B**, it will split and one bead will be sent to each sub-machine. When a bead from **A** returns, it will only proceed if no bead from **B** has arrived (or may arrive) with an earlier time stamp.

6 Examples

The following examples are taken from the *active badge* application. We have developed an event service that generates a **Seen** event whenever a badge is detected in a room. We may combine these events with those from a fire alarm service and those from an active database in order to recognise composite events.

Enters(B, R)
Leaves(B, R)

In order to detect someone entering a room, they must first be seen in one room and then be seen in a *different* room.

$$\Big(\$\mathbf{Seen}(B, R')\Big)\;;\;\Big(\mathbf{Seen}(B, R) - \mathbf{Seen}(B, R')\Big)$$

In the above expression, we start a new evaluation each time a badge is seen. When the badge is next seen, another evaluation starts, and the previous one either accepts or fails depending on the room the new sighting is in. 'Whenever' is the most closely binding operator, and 'Sequence' is the least closely binding, so the parentheses given above are not required. The expression for **Leaves** is identical, except that the old location is signalled, rather than the new one.

$$\mathbf{Leaves}(B, R') = \$\mathbf{Seen}(B, R'); \mathbf{Seen}(B, R) - \mathbf{Seen}(B, R')$$

Trapped(P)

By combining events from the badge system with those from a fire alarm service, we can signal all people who are seen after an alarm is raised. This can be done by waiting for an alarm, and then signalling each badge sighting before the all clear. We make use of the fact that database lookup can be modelled as an event, in order to determine the name associated with each badge.

$$\$\mathbf{Alarm}(); (\$\mathbf{Seen}(B) - \mathbf{AllClear}()); \mathbf{OwnsBadge}(B, P)$$

Trapped2(P)

A refinement of the above example is to trigger only *once* for each person still in the building one minute after the alarm. In this example, a side expression is used to set a variable to the time after which badge sightings should be signalled. @ is a special variable indicating the occurrence time of the last event.

$$\$\textbf{Alarm}()\{t \Leftarrow @ + 60\};\\ \Big(\textbf{AbsTime}(t); \$\textbf{OwnsBadge}(B, P); \textbf{Seen}(B)\Big) - \textbf{AllClear}()$$

The power of treating database lookups as events is made clear in this example. $\$\textbf{OwnsBadge}(B, P)$ will create one evaluation for each tuple in the database matching this template; i.e. one per person. Interest will be registered in each person's badge, and the composite event will trigger the first time a badge is seen.

7 Further Work and Conclusions

These ideas have been tested through a prototype implementation and evaluation using the active badge and multimedia facilities in our Laboratory. We have built a number of services incorporating the event interface and have found application development considerably eased by this standard treatment. We have also built and used a composite event detection service based on nested finite state machines and have defined an event algebra and composite event specification language. We have recently begun a collaborative project with NORTEL to apply these ideas to telecommunications management.

In summary, we believe that the approach is intuitive and makes it considerably easier to develop large scale applications. The prototype has been implemented in C++ by a simple extension to widely used software technology.

Acknowledgements

We acknowledge the UK EPSRC for supporting this work under grant GR/K77068. We are grateful to ICL for general support of our research group.

References

[1] Object Management Group. The Common Object Request Broker: Architecture and specification. Technical Report 91.9.1, Object Management Group, December 1991.

[2] Architecture Projects Management Limited *Advanced Networked Systems Architecture Testbench Implementation Manual*, 1993.

[3] R. Hayton and K. Moody, An Open Architecture for Secure Interworking Services, ACM SIGOPS European Workshop, September 96.

[4] R. Hayton, OASIS, An Open Architecture for Secure Interworking Services, University of Cambridge PhD thesis March 1996, Technical Report TR399.

[5] S. Gatziu and K.R. Dittrich. Detecting composite events in active database systems using Petri nets. In *Proceedings of the 4th International Conference on Research Issues in Data Engineering: Active Database Systems, Houston, Texas*, February 1994.

[6] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the 18th VLDB Conference, Vancouver, British Columbia, Canada*, 1992.