Supporting storage and retrieval of computer and human activity

Mark D. Spiteri and John Bates
University of Cambridge Computer Laboratory
Email: Mark.Spiteri@cl.cam.ac.uk, John.Bates@cl.cam.ac.uk

Abstract

This paper describes the architecture we designed and constructed to support the storage and retrieval of human and computer activity. We collect information about activities occurring in the real physical world as well as information relating to activities occurring within a computer environment. Important information regarding such activities is digitally represented through paramertised asynchronous *events*, denoting, for example, workstation operations like manipulation of files and launching of applications, or tracking of users' physical movements, usage of the telephone system, taking books from the library, etc. Our past experience has demonstrated how using events as the glue to build distributed *active* systems simplifies the construction of complex applications, and enables legacy stand-alone components to be rapidly integrated within a larger collaborative environment. Our motivation for storing events is that events can represent indexing points into activities and enable an activity to be reconstructed or replayed. This is particularly useful for capture of online collaboration, automated diary generation, memory recollection environments, and novel scenarios like visualisation and analysis of user mobility. We have designed and authored an event repository architecture that provides powerful search and retrieval facilities, enabling extraction of behaviour patterns, searching for simple and composite occurrences, and replay of stored sequences.

Keywords: distributed event programming, capture, storage, retrieval, replay

1 Introduction

This paper briefly illustrates the architecture we designed and constructed to allow the storage and retrieval of events. This architecture, sometimes also referred to in this document as an *event repository*, can capture and store events, as well as *inject* them back into applications to simulate replay of sequences of activity, or for re-building lost state. Section 1.1 describes our main motivations for carrying out this novel work. Section 1.2 describes some related work which has addressed aspects of these issues, while Section 1.3 outlines an experimental application we built as a case-study of how the architectural services developed can be used. A brief discussion of the event paradigm and building distributed applications using event-based components is found in Section 2. Section 3 discusses the design and implementation of the service architecture we designed to meet our requirements, and which we used successfully in support of the application illustrated.

The underlying framework to our system is that of distributed event-based programming. To digitally capture events relating to widely different activities, we use the event-based paradigm developed in previous work of the OPERA group [1,2]. This is designed to support the construction of large-scale active systems, that is those in which run-time notification of events can cause further actions to be initiated. Examples of such 'potentially' active systems are interactive multimedia applications, co-operative working applications, agent architectures, active databases and mobility aware applications which can respond to changes in users' physical locations. The uniform interchange of events between the possibly distributed components making up an application simplifies the

construction of a complex application and enables legacy stand-alone components to be integrated within a larger collaborative environment. Also, since we support generic manipulation of events, it is possible to cut down on application development overhead and rapidly construct support for manipulating events in programs. We developed our event system initially using a locally developed distributed object system, and later re-designed it as a layer above CORBA, using the latter as a distributed programming platform but remaining conceptually separate from it. We further discuss the event paradigm in Section 2.

1.1 Motivation

Previous research [1,2] has demonstrated that in an environment of ubiquitous computer deployment, constructing distributed applications from reusable active objects (i.e. capable of responding to externally generated events they have registered interest in) leads to efficient and dynamically re-configurable systems. Using these results as the basis of this work, we have identified a new requirement to support emerging scenarios; that the events used to glue together these applications can be stored, and then used for querying and replay of the stored activities.

For example, when used in conjunction with multimedia storage servers, stored event instances permit replay and support complex queries on historical records of multi-party computer-supported collaborative sessions. This enables the conference participants or other users to review and browse the conference recording, and replay all or segments of it. Another scenario in which storage of generic events is particularly useful is a memory recollection environment like a memory prosthesis [3]. This requires storage of various active information relating to user activities like document creation, media annotation, application usage, and location information. Temporal indexing and querying facilities are then required to be able to locate and review the activities carried out during any time period. Location and mobility information, such as that extracted from Active Badge sightings [4], can also be used on its own for various applications, such as security profiling, or visualisation of user mobility in emergency simulations. Other application areas we have identified as beneficiaries of our architecture are systems monitoring for distributed debugging, logging of phone and workstation usage for automatic diary generation, tracking of phone data for telecommunications fraud detection, and acquisition of marketing information for Internet commerce. Within some scenarios it is desirable to be able to use information from all these different sources together, hence creating a better and more fine-grained virtual picture of the human and computer activities monitored.

We propose that a generic service architecture that can store and retrieve “event instances” is a novel and desirable approach to handling these and other situations with similar requirements. Supporting the capture and retrieval of events is analogous to recording checkpoints (for rollback purposes) in a database system. However, unlike that example, the event store is general purpose and can be used for a wide range of applications. The main points of interest in an activity can be reconstructed at a later date using these temporal indices.

1.2 Related work

To capture these widely different activities, we used the event-based paradigm we have developed in previous work [1], [2], and illustrated in brief in Section 2. This is more powerful than other event environments like CORBA’s [5] in that it also provides support for higher level services like trading, brokerage, filtering, and composition of new events through specification of event sequences. However, we do use CORBA as our underlying distributed communication architecture. This provides us with both platform and language independence. Storage of event instances or ‘messages’, ‘notifications’, ‘actions’ or ‘traces’, has been carried out in a number of application domains, but it has not been the focus of the research. Mainly, it has been treated as a secondary aspect and has mainly been sidelined as secondary to issues like system optimisation and efficiency. Consequently, no satisfactory generic model has been arrived at, and no generic multiple-purpose event detection and storage architecture has been proposed. Of relevance to this investigation were projects undertaken in specific environments where some of the issues have been investigated, primarily:

- *Memory assisting tools like memory prosthesis* [3]. Video records of a user’s activities are maintained and accessed by indexing through a listing of activities the user undertook (like typing, using the phone, opening documents, changing office). The Coral suite of tools [6] support real-time capture and subsequent access to informal collaborative activities. While providing input to our work, we took the approach of using more

generic support in the form of our event store. Tools like these can be built more easily by employing our architecture and would benefit from our powerful retrieval mechanism.

- *Usability studies*, as in [7]. Such systems tend to focus on capturing local user interface events and do not consider distributed environments.
- *Temporal database query languages*. These tend to be more concerned with how the data changes over time. The focus of this research has been mainly on the semantics and optimisation of temporal query algebra. The theory behind these languages and algebra is quite advanced, but only a few experimental implementations have been carried out [8]. While some aspects of this research are relevant to the proposed investigation, such as expressing of temporal intervals for querying, the emphasis on database states and their variance in time is not as relevant.
- *Fault-tolerant computing and distributed debugging*. Ford [9] and other authors [10] have examined event detection and capture for storage purposes in the context of fault-tolerant computing and distributed debugging. Logs are maintained of event 'traces', where these traces represent low-level system activities like processor communication. This work has largely assumed integration with a debugging environment and is tailored for specific classes of events. [11] has attempted extending query support to be more generic.

The above listing is but a fraction of the large volume of work that has been valuable input to our work. These and other projects have investigated aspects of this proposed research in differing and specific contexts. Our research distinguishes itself by focusing on providing a generic platform for event modelling, storing and retrieval. This avoids bespoke services, allowing any event classes in an open environment to be stored.

1.3 Experimental case-study

In this section we describe an application environment which employs the event repository's functionality; monitoring and review of daily user activities, and in particular during collaboration sessions.

Distributed activity monitoring

We have built a collaborative platform that includes video/audio conferencing, text channels and shared drawing. This application is composed of a set of individual event-aware (or active) components, which are glued together by management components, as described in Section 2. The application is thus composed dynamically, and it can be made to use any activity as a trigger to perform some actions. For example, if two users are seen together in the same real-world location, then play them a video message on the nearest workstation briefing them on the progress of a project. We monitor workstation activities like logging-on and file operations, web browsing, as well as tracking of the physical location of users through Active Badges [4], monitoring usage of phones, printers and borrowing of library books. Events from these external sources, together with events relating to synchronisation of the collaborative environment components, are automatically indexed by the event repository, consequently allowing later replay and/or querying of a session. We have used this to generate high level minutes of meetings and for automatic generation of diaries. Media streams associated with a collaborative session are also stored (of at least the person who is speaking). These stored sessions can be replayed at normal, slower or faster speeds, and the replaying positions can be specified in temporal terms relative to the beginning of the session or with respect to *any other event or sequence of events* within the session. We believe in the hypothesis that the human mind often forgets specific details of an activity, but will remember its temporal relationships to other events, which occurred before or after it. We can perform queries corresponding to expressions like "show me the URL of the slide that John showed after Giles joined the meeting" and "show me the name of the book I took out of the library last month, just before I met Pawel in the corridor".

We have also used the event repository to counter for the situation when the network fails temporarily, resulting in a conference being partitioned into two or more segments. To show a user what happened whilst he/she was disconnected, exact copies of all local collaboration objects are instantiated on their desktops. Event repositories located within each participant's collaborative environment are then invoked and requested to feed back replays of what event originated from that participant. This allows the participant to view through the collaboration that went on 'on the other side of the partition', and then decide whether or not to add these updates to his/her session. All

users can discuss what happened in the conference partitions, and decide which of the event sequences they wish to be added to the session.

2 Building application with events

This section introduces distributed systems development using events as background to storage and retrieval of events. By using events as a uniform activity metric, systems/applications can be built around a generic event-response paradigm. Single or ordered combinations of events can be used to trigger further actions within the system/application. Using events as the uniform interchange of activity between the possibly distributed components simplifies the construction of a complex system/application. In this paper, we use the term *active system* to describe a large system or application composed of components inter-operating via event federation.

2.1 Components and glue

Event-based programming relies on two main concepts:

- Existing event-based components are used as the building blocks of applications. These may be distributed around the local network or Internet. They can inform interested users of pertinent events, such as a change in the location of a mobile user. The power of event-based system building is that no component has to be designed to work specifically with any other component. Event components thus need to implement two interfaces, an event registration interface if they are event sources, and an event notification interface if they are 'interested' in receiving events from other components. An event injection interface can also be provided, in which case management components can dynamically inject events into components according to pre-defined rules, as below,
- To compose many event-based components into an active system, a management component must be written. This is the bespoke part of the system, composed of event-based rules that describe actions to take in response to the receipt of certain events from the components within the system. Rules act as distributed systems integration glue, using any class of event from any components to drive any other components by injecting any actions.

In a multi-user application, for example a multimedia conference, each user owns a set of components, but the overall system is composed of the interacting set of all users' components. In such a situation, it is likely that each user will have a separate management component to take care of his/her own components.

Event sources can generate events of one or more event classes and likewise an event component can register interest in widely different event types. We thus generically represent events as classes, analogous to object classes. Each event has typed attributes, instances of which uniquely identify a captured activity. For example, an event source that provides information about the locations of users can offer monitoring facilities for the following class of event: `LocationEvent(Domain,Name,Type,Location)`. Class `LocationEvent` identifies that an entity has changed location in a specific domain. The type of the entity can be, for example, a person or a piece of equipment.

2.2 Filtering and federation

Our event system allows interested parties to *register interest* in events with certain characteristics. Only events that match the registered patterns are dispatched to the interested parties, thus reducing network traffic and filtering requirements.

To register interest with an event source, clients provide an *event template*. This is an event instance, with fields for exact match filled in and those for wildcard match expressed as variables. An example for class `LocationEvent` would be: "`LocationEvent("cl.cam.ac.uk", "John.Bates", "Person", L)`", i.e. report wherever John is seen in the Computer Lab.

When event instances are notified to clients, values for all class attributes are given, e.g. the instance `LocationEvent("cl.cam.ac.uk", "John.Bates", "Person", "Meeting Room")` means John has been seen in the meeting room.

Event-based components also have an interface for action injection. This allows a client of a component to invoke methods that inject actions into the component. The action injection interface methods for a component are often the same as the classes of event that the object can notify.

3 Storage and retrieval of activity information

In this section we describe the event repository architecture we have designed and constructed to allow the storage and retrieval of sessions of events. In the example of a multimedia conference, the event repository allows replay of the conference by replaying the events which occurred in the session at the same temporal separation as they originally occurred, and injecting them back into instances of the collaboration components. Event streams, composed of various different types, can be replayed back from any point in the recording, as well as with respect to any event or sequence of events occurring before or after it. The replay points can also be specified in terms of temporal intervals. Instead of replaying, one can ask for lookup and retrieval of specific patterns of event occurrences, and then use these collectively, thus specifying a *composite event*, recursively in other queries or replays.

3.1 Representation of events

For the purposes of generic storage, event instances are generically represented as objects of a particular event class, and the classes all inherit from the basic type '`event`'. This base class encapsulates the basic properties of an event as an instance of an action, or phase of an activity, occurring at a point in time, having a unique timestamp, scope, originating source, version (if applicable) and ownership tag if desired.

This classification enables event types to be defined quickly, and since they can be categorised through inheritance relationships, it becomes possible to execute generic query operations on abstract classes, which then automatically include the sibling class instances. The class hierarchy can be appended-to dynamically, and it is possible to have multiple class hierarchies defined simultaneously by tagging their member classes with different interface versions. This is similar in concept to Java's serialisation interface [12]. Examples are a '`Location`' abstract class with specialised inherited classes tailoring for '`ActiveBadge`' and '`GPSLocator`', or a '`Telephone`' hierarchy with sub-classes relating to picking up the phone handset, calling a number, and putting the phone on-hook again.

3.2 Storing events in sessions

In order to provide a context to associate related event instances, for example all the event classes that occurred with a specific conference, or all the events related to a user's activities for a particular week, sessions are provided. A session is particularly useful for limiting scope of retrieval and replay. A session is set up by an application; the application then sends events that it wants to be stored in the session to the event store. Any class of event can be stored in a session and an event instance can appear in multiple sessions.

3.3 Repository architecture

We designed and built our own database sub-system tailored for capture of high volumes of multiple streams of relatively small data objects (events) which need not be modified as they are to constitute a historical record. Every instance of the event repository consists of various components working concurrently and co-operatively, primarily a number of repository clients, and a repository server. While we have striven to keep the server component of the architecture as generic as possible to be of use in different scenarios with diverging requirements, the repository clients can be tailored to be highly specific to the application. Our aim in designing this architecture was to provide a core set of generic services coupled with highly and easily tailored components, so that the whole can provide the

required facilities without compromising on speed and efficiency. The architecture of the event repository is illustrated in Figure 1.

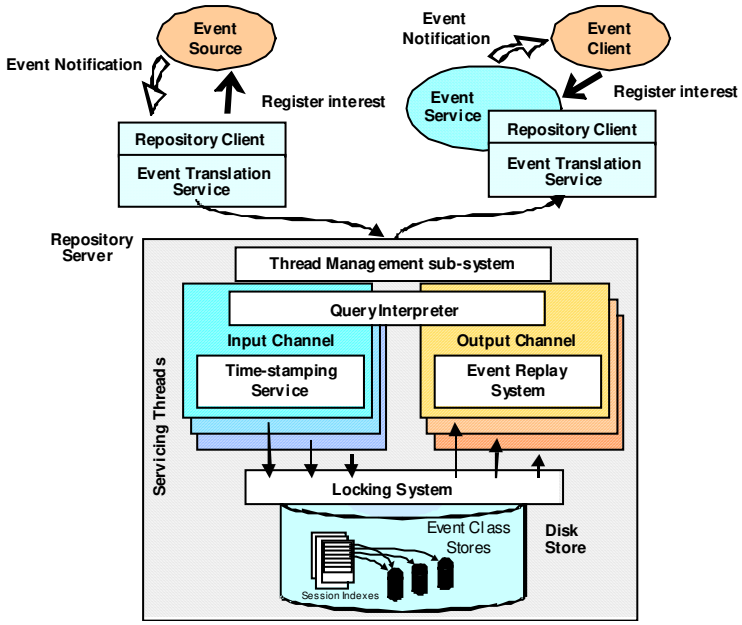

Figure 1 - Event store architecture

A specialised repository client registers interest in the particular event type to be stored, either with the local event broker if one is running, or directly with the event source. The client needs to be granted a sufficiently high security level so as to be able to receive notifications for the events it registers interest in. After translating the events it is notified to the repository's internal format (if required), and then passes them on to a thread within the repository server. This timestamps the events and stores them sequentially in separate disk buffers while generating higher level session indices as required. The client also acts as a command query parser for instructions sent to the repository by a user, a user application, or intelligent agent. While these commands are mostly to do with subsequent retrieval of the information stored, some operations are related to storage as well. For example, one can specify that a particular stream of events from some event source is to form part of a session recording involving various event sources. The server consists of a number of layers working together in tight configuration to minimise multiplexing. Multiple input and output streams can be handled concurrently. For each repository server it is customary to have multiple repository clients each catering for different event type streams.

Depending on the application domain where the repository is being used, there may be:

- *one centralised instance of the repository server capturing the activity information;* An example for this first case is a centralised repository instance which captures all movement sightings from all Active Badges in a domain. Application components can then interface with it to request a replay or analysis of these movements.
- *there may be multiple instances, for example, one per user;* such a configuration can be used to monitor all activities pertinent to a user or his agents; an example being for automated diary generation. The repository could monitor all workstation activities in the background, receive notifications from system-wide services like the user's movement sightings and use of library facilities, and monitor his engagement in online collaborative activity.
- *or a repository for each application or component;* A debugging application can thus use the repository to monitor system execution, allow analysis and review of the resulting trace and then destroy it once it terminates execution. In collaborative scenarios, it might be desired not to keep a permanent record of the interaction, but nevertheless use the event repository to support disconnected operation. The conference management objects can instantiate their own event repositories, use them to provide replay for participants after network partitioning, but then destroy them and all their records once the collaboration terminates.

While these latter options increase the storage requirements, the bandwidth required does not increase in a multicast event-brokering environment.

3.4 Timestamping and ordering

While any previous timestamp the event instances would have been tagged with from the source or some intermediate broker is retained, the repository considers the time of storage as the main timestamp. It consequently structures and orders data according to this. While there are considerations for and against this, we decided to make this distinction because in the case of collaboration the repository is acting as a distinct view of the application. This is analogous to the same way as the user it is associated with in a collaborative session has a specific view of the collaboration. In loosely coupled asynchronous distributed environments where large distances and consequently large latencies are involved, the user may not receive messages like, for example, video or shared drawing board actions, in correct order. While technically this view is incorrect, as it does not depict the way actions really happened in global time, nevertheless it does reflect the way the user saw the interaction. If the storage tool is to be of assistance in review and recollection of ideas, then it is important also to retain this viewpoint. In such conditions we also install separate instances of the event store for every user.

3.5 Event query and replay

We provide a language to manipulate stored event classes and instances, as well as event sessions. Queries can be composed through a query interface provided by the event store, either interactively, through scripts, or dynamically by application components. We provide a set of keywords and constructs to express contextual-temporal relationships and notions of temporal intervals in which one or more sequences of events might have occurred. While the syntax and concepts behind this query language are too elaborate to be discussed in the space available here, we will illustrate its use through the following examples;

- "*Replay from the group's diary from when Mark entered room T12 and John was there.*" One way of expressing this is to find the period when John was in T12 and then see if Mark was seen within that period in the same room. If this is successful, then replay from that instance onwards.

```
replay fromsession OperaDiary
fromwhere (badge.Name = 'Mark.Spiteri' and badge.location = 'T12')
containedin (: (badge.Name = 'John.Bates' and badge.Location = 'T12')
until (badge.Name = 'John.Bates' and badge.Location != 'T12') :)
```

- "*Retrieve when Kerry walked from the tea-room to her office (T15) and started editing a document sometime last week.*" Instead of replaying this will return all event sequences which match the specification provided.

```
retrieve fromsession KerryDiary
elementsof((badge.Name = 'Kerry.Rodden' and badge.Location = 'TeaRoom')
followedby (badge.Name = 'Kerry.Rodden' and badge.Location = 'T15')
followedby (workstation.documentedit.user = 'Kerry.Rodden'))
containedin (: 10/02/1998 to 17/02/1998 :)
```

To summarise, it is possible to find events relative to other events that have occurred before or after them, as well as with respect to specified bounded intervals. Intervals can be operated on to determine overlapping periods, and if desired, output can be filtered so that only specific event types, or events with specific parameters, are returned.

4 Conclusion

In this paper we have briefly illustrated our design for an architecture providing storage and retrieval of human and computer-based activities. We have achieved this by modelling activities uniformly as events and providing a model for representing event classes and instances, an architecture for distributed event propagation, and a model for querying and retrieving event-based information. The event store provides generic support for a range of applications until now catered for by bespoke services.

One application we have built to investigate applying event storage consists of storing online collaborative conference sessions in conjunction with real-world monitoring for review or as a memory aid. The replay facilities support recreating all of or sections of, a conference as it happened, using events as temporal indices to other media. Segments of the collaborative activity can be identified and then replayed according to other elements of activity within the collaboration. Finally, replay is a useful tool for re-building application state following disconnection as it enables instant review of the missed interaction.

Building on this initial architectural basis, we intend to elaborate on the retrieval facilities and provide high level support for simpler intuitive querying. We are also investigating applying data mining technologies to stored event instances.

References

[1] Bacon J, Bates J, Hayton R, and Moody K, "Using events to build distributed applications", in *2nd International Workshop on Services for Distributed and Networked Environments*, IEEE, 1995

[2] Bates J, "A Framework to support large-scale active applications", in *SIGOPS European Workshop*, ACM 1996

[3] Lamming M, Brown P, Carter K, Eldridge M, Flynn M, Louie G, Robinson P, and Sellen A, "The design of a human memory prosthesis", *The Computer Journal*, pages 153–163, 1994

[4] Harter A and Hopper A, "A distributed location system for the active office", *IEEE Network*, 8(1), 1994

[5] Object Management Group (OMG), *Event Service Specification*, ftp://www.omg.org

[6] Minneman S, Harrison S, Janssen B, Kurtenback G, Moran T, Smith B, and van Melle B, "A confederation of tools for capturing and accessing collaborative activity", in *Proceedings of the 3rd ACM International Conference on Multimedia (Multimedia '95)*, pages 523–534, San Francisco, CA USA, November 1995

[7] Badre AN, Hudson SE, and Santos PJ, "An environment to support user interface evaluation using synchronized video and event trace recording", Technical Report, Graphics, Visualization and Usability Center, Georgia Institute of Technology, USA, 1993

[8] Goralwalla IA, Tansel AU, and Tamer Ozsu M, "Experimenting with temporal relational databases", Technical Report, Department of Computer Science, University of Alberta, 1995

[9] Ford R, "Non-intrusive real time event capture", in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 225–227, Santa Cruz, CA, USA, May 1991.

[10] Lee I, Iyer RK, and Tang D, "Error/failure analysis using event logs from fault tolerant systems", in *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 10–17, IEEE Computer Society Press, Montreal, Que., Canada, 1991

[11] Mansouri-Samani M and Sloman M, "GEM: A generalised event monitoring language for distributed systems", Technical Report, Imperial College Department of Computing, July 1995

[12] Arnold K and Gosling J, *The Java Programming Language*, Addison Wesley, 1996