An event-based platform for collaborative object-sharing

(SUBMITTED AS FULL PAPER)

Tim Kindberg

Department of Computer Science, Queen Mary & Westfield College, University of London.
London E1 4NS. timk@dcs.qmw.ac.uk.

Tel. +44 (0)171 975 5236. Fax: +44 (0)181 980 6533.

ABSTRACT

This paper presents a software framework designed to provide collaborating distributed users with highly available, persistent shared objects written in Java. Our thesis is that, since an architecture for object-sharing needs to be open, it should use events as the main object-interaction mechanism. Events allows heterogeneous collections of objects to respond to shared occurrences. In particular, we use events to propagate updates to 'alternates' of shared objects, which are instances of objects that may exist in several 'flavours', for reasons of security or appropriateness to different users' requirements. For performance and availability, each user is given a local alternate of a shared object. We provide an *active store*: a persistent repository which is able to provide alternates in an application-specific way.

We also argue that conventional group communication is not sufficient by itself for propagating events in collaborative applications. This is true in the obvious sense that group communication systems are only designed for synchronous (real-time) working, whereas collaborations typically also include disconnected and asynchronous operation. But in addition we must augment the *delivery* semantics of group communication with *agreement* semantics. Event agreement is the synthesis of shared histories for objects, based upon concurrently applied streams of events from distributed users. We show how agreement is supported by our programming model, and given an example of a shared directory.

An event-based platform for collaborative object-sharing

Tim Kindberg

Department of Computer Science, Queen Mary & Westfield College, University of London.
London E1 4NS. timk@dcs.qmw.ac.uk.

1. Introduction

This paper presents a framework designed to provide collaborating distributed users with highly available, persistent shared objects. Our thesis is that, since an architecture for object-sharing needs to be open, it should use events as the main interaction mechanism for objects.

Mushroom[1] [1-4] is a system platform for supporting network 'places' – electronic workspaces containing shared objects. These objects are written in Java, and we shall refer to them as applets[2]. Applets are coarse-grained – for example, they are documents or electronic whiteboards rather than individual components in a CAD database. They are either group-aware (for example, shared editors), or they are wrappers for legacy resources such as word-processed documents and HTML files. Places and applets are instantiated at peers – users' workstations and laptops – for efficient access, and at servers, which maintain persistent and highly available copies.

Inter-machine communication in Mushroom is largely via *events*. An event is an object describing a state of affairs, for example "Mary added the paragraph P at position p in document D", encoded as a set of attribute values. An event which one object announces to declare a new state of affairs is passed to a method at an object that handles it. We say that an event is *applied* at an object that handles it. Events differ from invocation messages in that:

- they do not designate a target object (although they typically refer to one or more objects); instead, they are delivered wherever objects have declared interest in them;
- they are first-class objects, which can be stored, queried and passed between applications.

The principal advantage of event-based systems is their support of extensible and configurable distributed applications [5] and open system software [6]. They make no assumptions about the set of objects that exist, and leave reactive behaviour to be formulated to suit the application. Given agreement about event definitions and the scope of event delivery, all objects with an interest in a particular type of event can react in any desired way when such an event is announced.

Events are a convenient mechanism to describe a change that affects the copies of a shared object distributed at servers and peers. Events are particularly useful because these "copies" may in fact be heterogeneous. For reasons of security or otherwise, each object may exist in several 'flavours', which vary in their functionality or abstract state but perform related functions. An instance of a particular flavour of an object is called an *alternate* of the object. For example, in a medical application clinician A may access a patient's record in its entirety, whereas clinician B may only use an alternate which does not contain certain sensitive information, or on which certain operations are not possible. The clinicians think of themselves as sharing a single object (although they probably realise that it appears differently to each), but in reality each has his or her own alternate. Because of our requirement for alternates, our peers and servers together implement an *active store*. This is a store which supplies object state using application code, running at servers and peers, that knows about the different forms in which that state can be supplied.

---

[1]UK EPSRC grants GR/L14602 (1996-97), and GR/L64300 (1998-2000).

[2]Our applets are not the same as instances of the Java Applet class, but they are similar in being downloadable active content.

Another advantage of events is that they provide a convenient way to schedule operations upon distributed objects. If we guarantee to apply events at all sites that require them, and if we order events according to the consistency constraints that apply in the application, then workstations and servers need only apply concurrency control locally to ensure consistent updates [7]. Since events are first-class objects, we can construct services which process them on behalf of applications. For example, services can log and replay them. Disconnected users who have made potentially conflicting changes to a document can reconcile their changes on re-connection by replaying their events to create one or more consistent versions. Tools exist for users to manipulate the *history* of an object [14] – the sequence of events applied to an object's initial state. But the requirement is often for the system to reconcile autonomous streams of events automatically. We introduce the notion of event *agreement*, which is the definition of conformance for histories for shared objects.

We go on to describe in Section 2 the requirements for our collaborative object-sharing framework, which we relate to existing work; Section 3 describes the principal architectural features of our system, the active store and event delivery and agreement functions. We end in Section 4 with a summary and discussion of outstanding issues.

2. Requirements and related work

Mushroom users have several requirements for applets and the places that contain them. At the system level, the persistent state of a 'place' is a directory of applets, which users can browse without fetching undue amounts of applet state, and which the system uses to locate the applets. In what follows, 'object' may refer to an alternate of a place or an applet:

- **Object persistence between work sessions**. To support collaborations that can last many days, neither places nor applets should disappear when no users are active
- **High object availability** during work sessions, including minimal delays in saving, retrieving and updating objects
- **Disconnected operation** – this includes support for laptops and tolerance of server failure.
- **Consistency**. Some degree of consistency between alternates of a shared object at different sites, and between different objects is normally required; but the level of consistency can vary between objects, and users may accept temporary divergence in order to make progress with their work [8, 9]
- **Awareness information**. This is at a higher level than notification about a particular update to a particular applet; for example, users may want to know when a user joins or leaves a working session; or, they may need to see the entire history of a document.
- **Security**. This includes privacy, access and integrity control.

Several systems support generic object storage and event services for shared working. Bayou [9] provides an object store with data integrity guarantees, but its model of update propagation is 'gossip'-oriented: servers make pairwise exchanges of updates. This is acceptable for asynchronous working but not well suited to real-time working. Lotus Notes provides a replicated document repository with security guarantees and a limited form of conflict management, but it does not provide awareness of users' activities or support real-time working. More recently, Lotus' Notification servers [10] do provide real-time event propagation, but do not store persistent state.

Corona [11] provides server-based management of persistent object state. Servers store objects on behalf of clients, as byte streams representing the state of each object at some point, plus the updates required to render this into the current state. Servers do not run any application-specific code. Clients supply the object state and updates; and clients, rather than servers, perform all update processing. Clients can obtain object state and updates only from a server.

The Cambridge event system [5, 12] dispatches events to just those clients that have registered interest in them, normally by maintaining event filters at servers. Events may be composite. For example, an event can express that both Miss Scarlet and Professor Plum have entered a particular room.

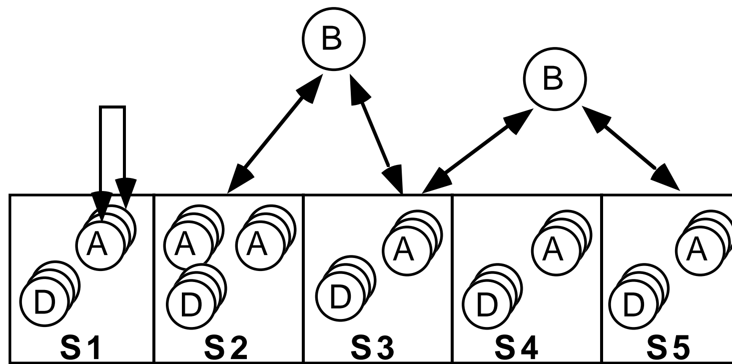

Figure 1. The active store with 5 sessions (events not shown).
A = applet alternate (replicated); B = browser, D = session directory (replicated). Arrows denote storage/retrieval.

Java 1.1 has an event model that is used primarily for user interfaces, but which could in principle be used for generic, distributed event propagation. The model suffers from the fact that objects which are interested in events must be able to identify their source. This is an unnecessary restriction. For example, a broker should be able to obtain information about the movement of gold prices, without having to know all the financial instrument feeds to subscribe to. Our event delivery system is akin to CORBA event and notification channels [13, 16], in allowing transparent many-to-many event propagation.

3. The Mushroom architecture

Like Bayou [9], we assume a weakly connected set of machines. That is, machines may fail or become disconnected. Furthermore, we aim to support both synchronous and asynchronous group working. The basis of our design is the following:

- **Objects are state machines**. All objects are assumed to be state machines [17], so that the state of an object is a deterministic function of the sequence of events applied to it.
- **Efficient event pushing**. To support synchronous working, Mushroom provides an event delivery service, which pushes new events to the objects requiring them on a best-effort basis, in order to meet the requirements of user interaction. If, for example, a user is to be able to place an object in a whiteboard and refer to it over a telephone (or other real-time audio) link, then other users need to see the new object as quickly as possible.
- **Active storage**. Mushroom provides an active store. Clients of the store are able to save and retrieve object snapshots and events, and apply events to objects as they see fit. They can in principal apply any set of events in any order, producing, for example, alternative histories of virtual worlds.
- **Event agreement**. To support object sharing (as opposed to individual construction of new objects), most applications need to impose constraints on what are allowable object histories. For this reason, our platform has to support the notion of event agreement. This defines conformance for alternates of the same shared object, in terms of their histories.

We now describe the design of the store, the event programming model and services, and agreement.

3.1 The active store

The active store contains object alternates and events (Figure 1). By using the term ‘active’, we mean to emphasise that code runs at the point of storage, including servers as well as users’ peer machines. The reasons for this are:

- This code can manipulate the state of the objects and take persistent state snapshots (checkpoints) in response to events. This enables the store to provide an up-to-date copy of object state. It also enables programmers to implement per-applet persistence policies, which do not require users to save their work.

- The code can provide copies of object state in an applet-specific way, including support for multiple flavours
- The events that update the state of the store usually derive from user actions, but we can also support housekeeping applets that run without human intervention. For example, when no user is currently active in a place, a housekeeping applet could re-organise the other applets in the place.

The disadvantage of an active store compared to a conventional passive store that simply stores and retrieves arrays of uninterpreted bytes (e.g. in [10]), is the potential for a performance bottleneck. We return to this point in Section 4.

The Mushroom store is a union of the objects stored in a global collection of sessions S_i:

$$S_i = < Directory, Applets, Groups, Events, PersistenceDomain >, \text{ where:}$$

- $Directory : String \rightarrow Metadata$ is a hierarchic per-session name space mapping users' names for applets into their attributes; it is implemented by a set of alternates
- *Applets* is the set of individual alternates that implement the applets in the session, including all instances of the different applet flavours (there are mappings $Applet \rightarrow Flavour\ (class) \rightarrow Alternate\ (instance)$)
- *Groups* is a set of one or more group communication channels for distributing events in the session
- *Events* is the set of past events (selectively stored)
- *PersistenceDomain* is a set of servers which maintain persistent copies of events and persistent snapshots of alternates, as well as in-memory copies. Our 'housekeeping' applet would run at one of these servers.

The clients of the store are Mushroom browsers and the alternates themselves. Figure 1 shows browsers storing and retrieving objects, and also applet alternates supplying state to one another. Mushroom maintains the consistency of alternates of the session directory, which it presents to the user as a 'place' to browse. Applets maintain their own consistency in accord with their individual requirements. Alternates of the directory and applets exist at the peers, where users access them, as well as at the members of the persistence domain. As others have observed [10, 15], user workstations are liable to crash, and users may quit software or disconnect their laptops at unpredictable times. By contrast to peers, we can expect servers to be reasonably reliable machines, whose continuous execution of Mushroom software is ensured by administrators. Membership of the persistence domain is controlled by administrators and so is slow-changing.

The only organisation of the persistence domain which is currently implemented is primary-secondary replication. Note that this is on a per-session basis, so that different sessions can map to different primary (and secondary) servers. Peers are able to obtain state snapshots and events from any server in the corresponding persistence domain. Should the primary fail or become unreachable, users are free to continue working in places which remain accessible locally or via a secondary server, but they risk producing inconsistent and irreconcilable results.

Locating sessions and alternates

A user first encounters a place either by identifying a it with respect to a 'root' place at a server, using a URL-like syntax, or by clicking on a link to a place in another place. A link applet contains a globally unique identifier for the corresponding session, and a list of DNS names of persistence domain members for that session, which it uses to locate a copy of the session directory. As the user browses and clicks on items in this directory, the corresponding applets use the same information to fetch alternate state.

Currently, clients always obtain snapshots of sessions and alternates from one of the persistence domain members, but it is a matter of current investigation whether it is sometimes beneficial overall to allow users' workstations to supply one another.

Applet State

Applet state is stored in the form $< snapshot, flavour, timestamp >$. The snapshot is a serialisation of an alternate at some logical time. The 'flavour' is an object describing the type of snapshot that is stored. It encodes:

- the alternate class;
- any label that is needed to distinguish between alternates of the same class, such as the identity of a security principal for which it was derived. Typically, an applet programmer will produce one class that can supply alternate state of any flavour.

We record the logical time efficiently as a vector timestamp – rather than as a more specific list of identifiers of events that the applet handled. The purpose of the timestamp is to be able to detect whether one of two snapshots is later than the other, whether they are equal, or whether they are concurrent. The latter can occur, for example, if two disconnected users work on the same document.

Each event has a *context* [19], which is a generalisation of a sequence number that accommodates transmissions on multiple communication groups. The contexts of events from a given originator (peer or server) are monotonically increasing, so that the set of events from a given originator is represented by its latest event's context. The timestamp of a snapshot has an entry for each originator that has announced an event that is reflected in the snapshot. More precisely, a timestamp is a set of pairs:

$$\{< o,c >: \exists event\ \ e,\ e.originator = o \wedge e.context = c\}$$

– and the timestamp of snapshot S is one that contains precisely the most recent entry from each originator of the events that were applied to produce S. We periodically define a new timestamp epoch, designated by a unique integer from an increasing series, in order to be able to remove entries for users (originators) who no longer modify an object.

Managing state and its persistence

Consider a user clicking on an icon representing a whiteboard. The browser must instantiate an alternate locally, join the appropriate communication groups in order to receive events, and synchronise the alternate with events as they arrive. Classes (actually, Jar files) are distributed through Mushroom and cached locally to avoid re-fetching classes. So the browser usually has only to retrieve the state. By comparing the timestamp of any locally stored state snapshot with that in the session directory, the browser is able to determine whether it needs to obtain a more up to date version from another machine.

Alternate state is supplied by another alternate. First, it selects an appropriate flavour to supply, depending on the user's preferences and security profile. Second, it supplies the client with the state in one of two ways:

a) in the first case, the server's alternate is designed to store intermediate snapshots taken by clients, and to record the events needed to bring the most recent snapshot up to date. The client has to apply the events to the alternate instantiated from the snapshot.

b) in the second case, the server's alternate applies each event to itself as it arrives; it is then always in a position to supply the most up-to-date state.

Options (a) and (b) make different trade-offs between network bandwidth and server computational load. The choice depends partly upon the type of applet and local performance considerations. Note, however, that option (a) is not always possible, for two reasons. First, the server is not able by itself to generate state for an arbitrary alternate. It may be dependent on a client to generate an alternate with restricted state needed by another client. Second, it may not be possible to identify which events need to be applied. As we shall see, applets are able to change their registration of which events they are interested in.

Whichever of options (a) and (b) is chosen, we normally need also to push current and future events to the peer. The user who has clicked on a whiteboard will want to see all updates (by herself and other users) as she views it. Synchronisation is required between the arriving stream of events and the snapshot's state. The user should not miss any events; and non-idempotent events should not be applied twice. We use timestamps to identify when an event

has already been applied, and event contexts to identify when an event has been missed. Joining a communication group to receive events can take a non-negligible time. This can be amortised by applying to join the group concurrently with fetching the snapshot to which the events are to be applied. But events may still be missed, and to avoid a delay in fetching them, the supplier of an alternate acts as a 'buddy' of the client, sending events temporarily down the same pipe that it used to supply the state as it receives them (the supplier is already a member of the group).

Now consider persistence, which is achieved by committing state snapshots and events to persistent storage. In some cases, and especially with option (a), the user is responsible for saving state explicitly[3]. A persistent copy is then saved locally and, if connected, at one or more members of the persistence domain.

However, user-driven persistence is not desirable for all applets. For example, users may require state to be saved after every action. An alternative is to use events as a trigger for recording state. In this case, alternates running at peers and, in option (b), at persistence domain members, save snapshots and events to persistent storage according to a per-applet event-driven policy. Since all alternates export a *serialise*() method, it is possible to implement the persistence policy externally to the applets themselves.

3.2 Event services and agreement

Support for asynchronous and synchronous working and disconnected operation requires, first of all, two basic types of event service: delivery and storage. The delivery service distributes events as they are produced. The storage service stores events for later retrieval. In addition, we require application-specific agreement functions which define conformance for histories for shared objects. We shall examine agreement in the second half of this section. First, we describe the requirements for storage and delivery:

- **Transparency.** Objects should not be forced to learn about which objects produce the events they consume, or which objects consume the events they produce.
- **Specificity.** An object should be passed only the set of events that it currently expresses an interest in. For the sake of generality, specificity is achieved by applying arbitrary event filters – Java objects which have a *Boolean accept(Event e)* method – to determine whether the corresponding object is to consume event *e*.
- **Scope**. The event delivery system must match announced events to those objects that have expressed interest in them, and which have the right to read them. Some form of scoping is required to bound the search necessary to find all possible consumers of a given event. Once that scope is chosen, however, the system should support the selective export of events to a different scope which may not have been envisaged at first. This is known as *federation*.
- **Reflexivity.** By default, an object that announces an event will consume the same event. This allows an alternate to order updates originating from itself with respect to those originating from other alternates of the same applet.
- **Scalability**. Response times are important for the quality of interactive applications. Events must therefore be delivered efficiently so that the delay between event production and consumption is at worst $O(N)$, where N is the number of consumers. Some applications may require $O(\log N)$ latencies.
- **Support for appropriate quality of service (QoS)**. In addition to end-to-end latencies and persistence, applications often have particular requirements for delivery ordering semantics during synchronous working. For example, a simple chat applet requires that each user's contributions should appear in the order of their utterance.
- **Selective event storage and retrieval.** Users often need to go back in time selectively to review, replay and reconcile object histories.

---

[3]The applet must overcome the problem of which state to save, if users have non-identical state, and of users indiscriminately overwriting one another's saved state.

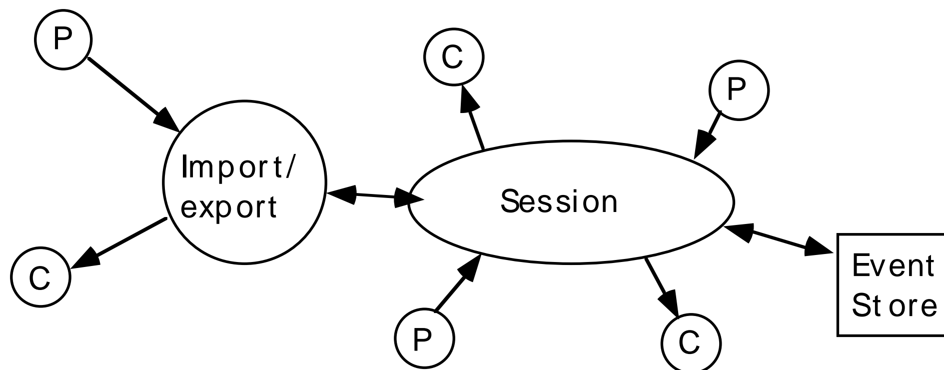

Figure 2. An event-oriented view of the architecture. P = producer, C = consumer.

- **Security**. Events raise interesting security issues, covering secrecy, access control and integrity checking. Since events are stored and replayable, users become concerned with protection of the past, and not just the present and future. This is a non-trivial issue because users join and leave collaborations. Since users have local copies of objects, they can contrive to perform whatever operations they like upon them. Security requires ensuring that only legitimate state snapshots are acquired, that only legitimate events are applied, and that both are disclosed only in accordance with users' policies.

Figure 2 shows the overall organisation of the event architecture designed to meet these requirements. Event producers and consumers transfer events over a session, which manages event scoping as well as storage. Events pertaining to a cluster of objects are propagated with a default scope that is equal to the collection of sites where those objects reside. The figure also shows an import-export service, which is used for federating sessions, and an event store, which is used for temporary and permanent storage of events.

By default, events are not persistent, and they originate from, and are delivered to, only the current members of the session[4]. FIFO (source) ordering is the default. However, this arrangement can be extended. One or more event stores may join the session to log events – either all events, or ones satisfying application-specific criteria. A session member may query a store for any events that were transmitted over a period when that member was entitled to join. The event import/export service also joins the session. Its purpose is to:

- allow session members selectively to consume events produced by non-members
- allow non-members selectively to consume events produced by members – or events synthesised from these.

Other event services may be supplied to the session members. For example, a membership monitoring service could announce when a user who can take on a particular role has joined the session.

The main programming interface for events is shown in Table 1. Event consumers use *setHandler*() to supply a filter object, which has a method to accept or reject the events announced on the session. The method also declares the object which is to handle the event – often the invoker. The priority is used to order event handling. The *announce*() method causes a copy of the given event to be distributed to wherever a filter accepts it. Optionally a communication group is specified, which causes the event to be delivered over a narrower scope than the session members as a whole.

To meet the requirement to federate event delivery, clients use *setExporter*() to declare an object that will export events selectively. The system serialises the object and recreates it at the server that is supplying the export service, for the sake of efficiency. This server passes all events that are classified as exportable to it. It can then choose to pass on particular events, or to synthesise compound events from them. For example, it could examine "user enters" events,

[4]A member of the session is a member of one of its communication groups.

Session method	Function
setHandler(EventFilter f, EventHandler h, int priority);	Declare filter *f* to determine which events are to be handled by *h*
announce(<Group g, > Event e);	Announce *e*, optionally declaring a (sub-)group *g* to use for delivery
setExporter(EventHandler h);	Declare a handler to export events selectively from the session
markForExport(EventFilter f);	Declare a filter to determine which events may be exported
markForImport(EventFilter f);	Declare a filter to determine which events may be imported

Table 1. Event programming interface.

and announce when both Colonel Mustard and Mrs White have joined the session. The handler typically announces events on another session, although it may of course pass them on to an individual site.

By default, only those with sufficient privileges to join a session may establish an exporter object when not a member. The privileged method *markForExport*() designates certain events as ‘For Export’ – either to specific principals, or to anyone – in which case the export service will pass them to remote event handlers, where permissions allow. In a similar vein, we allow those with sufficient privilege to mark events for import, using *markForImport*(). The instance of EventFilter is used to determine which principals can import events of which type into the session. For example, users who are not currently able to join a session could announce their request to do so by announcing a “User Requires Entry” event.

The server that hosts EventHandler objects is secure from attack by them – at least, up to the limits of Java’s security provisions [18]. This is because they are passed no local objects, except events that have been validated as being for export. The server software contains no public static variables or methods.

Scope and scalability

The performance implications of design decisions in event delivery systems are not obvious, and we now discuss the background to some of the above decisions. Consider the three possibilities for filter placement in an event-delivery system:

- **Source filtering**. If all announced events were filtered at source, then in some cases we could avoid some events needlessly travelling over the network. The scheme has the disadvantage of burdening an event source with applying potentially many filters required by other nodes. We also burden the system with the task of establishing the location of all possible sources of events that might pass the filter.
- **Destination filtering**. All events in the collaborative session are delivered to all sites; the destinations apply filters and throw away redundant events. This scheme imposes some unnecessary computational load on the destinations, but it can take advantage of multicast protocols to minimise bandwidth utilisation and transmission latencies.
- **Filtering at an intermediate node**. Systems such as SGAP [20] (the successor to NSTP) place filters at intermediate servers. Events travel first to a server, which evaluates the filters supplied by every potential destination node, and sends the events only to where the event was accepted. This design trades off a reduction in destination processing load against the latency incurred by evaluating every destination’s filters at the server. This type of filtering also misses the potential benefits of using multicast protocols.

Destination filtering and intermediate-node filtering are the most viable options, but the choice is not the same in all situations. If the product of the event generation rate and *event miss rate* –

the proportion of unwanted events that travel to sites – is close enough to zero, then destination filtering supported by multicast communication is a reasonable choice. This is the approach we have adopted for supporting collaborative applications, where users' interests tend to fall into common categories. For example, anyone collaborating in a place normally needs to receive events concerning updates to the set of applets in the place. Anyone currently using the whiteboard needs the set of events announcing updates to it.

To help minimise the event miss rate, we allow for sessions to have multiple groups. Each session has a main group, which is used to propagate events concerning the directory of applets in the session. Applets that transmit events at low volume can use this same group. But applets that announce events at high rates need to be assigned their own communication group. Currently this assignment of groups to applets is explicit.

Event delivery and agreement

It is not desirable to meet our delivery requirements using a group communication system with strong ordering or atomic delivery guarantees [21]. The aim of such systems is to make delivery a single, atomic action, with uniform guarantees for all recipients; the application can apply an event as soon as it is delivered. We wish to allow disconnected operation without paying a performance penalty, and to allow applications to impose only the ordering constraints that they actually require. We consider event propagation from source to application to consist of two concurrent activities, delivery and agreement. The delivery service distributes events, but they may or may not be applicable at the time they are received; agreement is the application-specific process of constraining the events, and their order, which are allowed to belong to a shared object's history. Depending on the semantics of the application, an alternate of a shared object may decide to apply a non-agreed event. It could, for example, be used to represent a tentative state of affairs, such as the introduction of an applet into a place with a tentative name. Once an event has been agreed, it may be applied to an alternate with confidence that others applying the same agreed events with the same ordering constraints will be consistent enough for sharing.

There are two basic delivery services:

- unreliable, FIFO-ordered multicast – this is what is required for events that represent changes to soft state: it is not significant if an event is missed, because another event that will make it redundant will be transmitted soon
- reliable, FIFO-ordered (multicast and point-to-point).

FIFO-ordering is needed for most non-trivial applications. It requires careful definition, because a user may be active in several places concurrently. Is it necessary to order events so that ordering is respected for any other site active in the same combination or a sub-combination of places? In our design we chose to restrict the ordering to individual sessions. That is, X may see two updates to two places from Y out of order; but X will never see two out-of-order updates from Y to the same place. Even within a session, reliable FIFO multicast ordering requires care, because an individual site can be a member of several groups. Simple sequence numbers will not suffice to allow a consuming site to detect a missed event [19]. Events are labelled with a set of sequence numbers, one per group, generated at the announcing site: $\{< g, seq >\!: \; seq = count\ of\ events\ announced\ over\ group\ g\}$; such a set is what we referred to above as a context.

To illustrate the need for agreement, consider two users, each concurrently introducing an applet into the same place, announced by events of the form *new(applet, name)*. It is possible that they will choose the same name, "X". An integrity property of places (session directories) is that no two objects may have the same name. A convention can be temporarily applied locally, of using the originating user's name to make the object's name unique: when the first event arrives the object will be called "X (by Tim)", and the second "X (by Fred)". This serves to give the objects unique names; but it is not satisfactory as a permanent state of affairs. First, it is not what either user wanted. Second, the events that were transmitted were of the form *new(applet, "X ")* – which is *false* with respect to the state of the alternates.

```
handleEvent( Event e )
{
        e.wait();                          // Wait until have processed events that this depends on
        if ( e.isNotAgreed() )
        {
            if ( IAmAgreementSiteFor( e ) )
            {
                e' = checkApplicability( e );    // Return same/derived event to apply, or null
                if ( e'!= null )
                        session.announce( e' );
                else
                        session.announce( e.repudiated() );
            }
            else
                applyTentatively( e );            // Apply event in recoverable way
        }
        else
            apply( e );  // May involve roll-back to previous state, & re-application of events
}
```

Figure 3. Algorithm for the alternates of an applet, one of which is at the 'agreement site'.

The solution is to nominate a site which will enable sharing by constraining allowable histories for the affected object. It does this by labelling events as 'agreed' and re-announcing them. Agreement may entail:

- **transformation** of the event – for example, the second event in our example to arrive at the agreement site could be transformed into the event *new(applet, "X (by Fred)")*; the first could stay the same, so that the names would be "X" and "X (by Fred)"
- **ordering** – if all sites apply events in the order received from the agreement site, the agreement site imposes a total order on the events; we shall see below that weaker ordering is possible
- **stability** – the agreement site may choose not to agree the event until it has been stored at a minimum number of other (server) sites, in order to guarantee the level of its availability.

The generic agreement algorithm for the alternates of a given applet is shown in Figure 3. It assumes that any alternate that applies events that have not yet been agreed retains the capacity to roll-back if necessary when an agreed event is announced. It applies it after the last agreed event, and then re-applies events that have tentatively been applied since. Any site may be nominated as the agreement site. It is normally a server in the persistence domain, but it could equally be a mobile computer of a privileged user, who has special rights to determine the applet's history. In any case, it is an alternate of the applet that announces agreed events, since agreement is application-specific. Note that an event can be repudiated as well as agreed. Repudiation means that an event cannot be applied at all – for example, for security reasons – is to be deleted from the object's history.

Application-specific ordering is sometimes more appropriate than the default of FIFO event ordering semantics, or the total ordering which an agreement site may impose. The announcer of an event can declare a dependency of one event upon another:

```
event.dependsOn(Event e);
```

– correspondingly, a recipient calls:

```
event.wait();
```

– to block the caller until those events on which it depends have been handled. We give a more formal account of the use of the *dependsOn* relation in specifying conformance for histories in

the appendix. As an example, consider the following events that pertain to applets in a place which has a non-hierarchical directory:

Event	**Description**	**Depends on**
new(a, X)	Applet a (a= unique system id) is imported with name X	–
delete(a)	Applet a is deleted	new(a, X)
rename(a, X, Y)	Applet a is renamed from X to Y	rename(a, Z, X) or new(a, X)

To maintain the integrity of the directory the following steps are necessary:

- the agreement site agrees all events of the form *new(a, X)*, unless an object exists with name *X*, in which case it derives a new name X' (using the user's name as described above) and agrees the transformed event *new(a, X')*
- all sites announcing *delete* and *rename* events make them depend on the last event (*rename* or *new*) that named the affected object. The agreement site respects the *dependsOn* order while it processes *rename* and *delete* events (as does any site). It transforms an event *rename(a, X, Y)* if an object named *Y* currently exists
- the agreement site repudiates all events that do not designate an extant object

These steps will determine a set of agreed histories, but the resultant histories may not be satisfactory to all users. For example, a user may rename an object only to discover on re-connection that someone else has deleted it.

Finally, note that a single event may be agreed separately in several objects' histories. For example, the event "Colonel Mustard and Mrs White were both in place 'Conservatory' at time T" could be given a definite ordering in the history of each of these characters.

4. Summary and discussion

We have described a programming framework and system architecture designed to support distributed users who share highly available, persistent objects. In Mushroom, the shared objects are places (which are essentially directories) and the applets they contain. Both places and applets are instantiated as one or more alternates distributed at servers and the user machines (peers) where they are used. Alternates may be heterogeneous, particularly for reasons of security. We have argued that events are particularly suited for object-sharing frameworks. They support heterogeneity. They enable us to meet users' requirement for high-level awareness information during collaborations. They also fulfil users' requirement to inspect the histories of objects, and not just their current state.

We provide a store for snapshots of object state and events. The store is active: application code runs to provide alternates that match the preferences and security profile of clients. If, for example, the requirement is for some users to see a place without certain confidential objects, and for others to see all objects in the place, then we must have a way of supplying clients with appropriate alternate state in an application-specific way.

The store is divided up into sessions, which correspond to the user's notion of place. Sessions also manage the scope of delivery of events. The delivery service pushes events to synchronously collaborating users, and to the servers that make up the session's persistence domain. On re-connection, users can supply the events they generated, and retrieve events generated by others during disconnection. The system assumes no responsibility for the schedule of event application: this is left to applications.

We have shown how events are 'agreed' as belonging to a shared object's history. Ordering constraints ensure that alternates apply agreed events to produce states that are consistent within the demands of the particular application.

Maintaining alternates at servers which maintain up-to-date, complete state for applets enables us to produce the latest applet state for any client. But this has performance

implications. We are investigating quantitatively the performance implications of different shared object management schemes, by simulation as well as by instrumentation of Mushroom.

Engineering questions also continue to surround the design of the event delivery system. Neither of the two main schemes we described, multicast-based-destination-filtered and point-to-point-intermediate-filtered, is satisfactory for all circumstances. Such factors as the event announcement rate, the event miss rate and the physical distribution of the users affect the system's scalability (measured by event latencies). The same quantitative investigation just referred to also aims to throw light on the efficacy of hybrids of these two schemes, and of systems that make a dynamic choice between the two, based on self-instrumentation.

To our knowledge, no other work exists on security for event-based systems. We pointed out above that events do, however, raise new security questions. As an example, consider the possibility that, while Fred is not present in a place, Sid writes something rude about him, shows it to his colleagues, then erases it. In conventionally edited documents, Fred would be none the wiser. Unfortunately, Sid forgot that Fred has the right to read all events applied to the document. While we have prototypical solutions to event security, there are two challenges that remain. First, the access control model has to be easy to understand, so that novice users such as the clinicians we are currently working with can consistently and accurately apply it. We are working on an approach that utilises secure containers. Second, the need to propagate confidential events has an impact on the engineering of our event delivery service. Event scoping is a function of secrecy as well as interest, and our architecture does not yet reflect this.

Finally, we turn to our notion of agreement. The decision to separate event delivery from event agreement enables application writers to support disconnected working and to give tentative feedback to users, who demand quick responses for effective interaction. Our scheme is the event-oriented equivalent of Bayou's use of dependency checks and merge procedures [9]. There are two main differences. First, the Bayou designers transform their write operations on the fly as necessary, in order to maintain integrity. Our agreed events play a similar role, but it is important to record what the events were transformed into, since users want to know the history of objects. Furthermore, the announcement of agreed events saves their recomputation at other sites. Second, the Bayou design is concerned to apply either total ordering or happened-before ordering to all updates. We believe that this is unnecessary in many cases. Our example of updates to a shared directory show that only some events need to be ordered with respect to one another in order to maintain the directory's integrity. However, our programmed solutions to the agreement problem are *ad hoc*, and we are working on a programming framework for application writers, to support them in expressing constraints [22] as well as dependencies.

References

[1] T. Kindberg (1996). Mushroom: a framework for collaboration and interaction across the Internet. Proc. CSCW & the Web, 5th ERCIM workshop, Busbach, U., Kerr, D., and Sikkel, K. (eds), GMD, pp. 43-53.

[2] T. Kindberg (1996). A stake in Cyberspace. Proc. 7th. ACM SIGOPS European Workshop, Connemara, Eire, Sept., pp. 83-88.

[3] T. Kindberg, G. Coulouris, J. Dollimore and Jyrki Heikkinen (1996). Sharing objects over the Internet: the Mushroom approach. Proc. IEEE Global Internet 1996, London, Nov., pp. 67-71.

[4] Mushroom project home page: http://www.dcs.qmw.ac.uk/research/distrib/Mushroom.

[5] J. Bacon, J. Bates, R. Hayton, and K. Moody (1996), Using Events to Build Distributed Applications. Proc. 7th. ACM SIGOPS European Workshop, Connemara, Eire, Sept., pp. 9-16.

[6] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers and S. Eggers (1995), Extensibility, safety and performance in the SPIN operating system. Proc. SIGOPS '95, ACM, pp. 267-284.

[7] A. Schiper and M. Raynal (1996), From group communication to transactions in distributed systems. Comm. ACM., vol. 39, no. 4, pp. 84-87.

[8] T. Kindberg (1996), Notes on concurrency control in groupware. ftp://ftp.dcs.qmw.ac.uk/pub/distrib/publications/Mushroom/ccInGroupware.ps.gz

[9] K. Peterson, M. Spreitzer, D. Terry, M. Theimer, and A. Demers (1997). Flexible update propagation for wweakly consistent replication, *Proc. 16th ACM symposium on operating systems principles*, pp. 288-301.

[10] J. Patterson, M. Day, and J. Kucan (1996), Notification servers for synchronous groupware. Proc. CSCW96, pp. 122-139.

[11] H. Shim, R. Hall, A. Prakash, and F. Jahanian (1997), Providing flexible services for managing shared state in collaborative systems. Proc. 5th European conference on CSCW, pp. 237-252.

[12] J. Bates (1996), A Framework to Support Large-Scale Active Applications. Proc. 7th. ACM SIGOPS European Workshop, Connemara, Eire, Sept., pp. 1-8.

[13] Object Management Group (1995). The Common Object Request Broker: Architecture and Specification, Revision 2.0.

[14] W. Edwards and E. Mynatt (1997), Timewarp: techniques for autonomous collaboration. Proc. CHI 97, Mar., pp. 218-225.

[15] O. Babaoglu and A. Schiper (1994), On group communication in large-scale distributed systems. Proc. 6th ACM SIGOPS European Workshop, pp. 17-22.

[16] Object Management Group (1996), The Notification Service Request for Proposals.

[17] F. Schneider (1990), Implementing fault-tolerant services using the state machine approach: a tutorial, *ACM Computing Surveys*, vol. 22, no. 4, pp. 299-319.

[18] D. Dean, E. Felten, and D. Wallach (1996), Java security: from HotJava to Netscape and beyond. Proc. IEEE Symp. on Security and Privacy, May.

[19] T. Kindberg (1995), A sequencing service for group communication. Abstract in Proc. Principles of Distributed Computing, Ottawa, Aug., p. 260. Available as tech. report no. 698, Dept. of Computer Science, Queen Mary & Westfield College, U. of London.

[20] M. Day (1997), Simple general awareness protocol. Tech report, Lotus Corp.

[21] R. Friedman ans R. van Renesse (1996), Strong and weak virtual synchrony in Horus. Proc. 1996 symposium on reliable distributed systems, IEEE Press.

[22] G. Starovic, V. Cahill and B. Tangney (1995), The ECO model: events+constraints+objects. Tech report, dept. of Computer Science, Trinity College Dublin.

Appendix. Specifying conformance for histories.

Consider a computation *C* in which each of several sites P_i $(i \in \mathrm{I})$ operates upon an initial state S_0 and produces a partial history $H_i = < e_1^i,\ e_2^i,\ ..,\ e_{N_i}^i >$ of events.

Let the agreed events of *C* be $A_C = \{e_i^a:\ i = 1,\ ..N\} = \{agree(e):\ e \in \coprod_{i \in \mathrm{I}} H_i\}$. Let $\overset{i}{\rightarrow}$ be the serial order of P_i ($a \overset{i}{\rightarrow} b$ *iff* P_i announced event *a* before event *b*). Let $\rightarrow$ be the transitive closure of *dependsOn* and $\overset{i}{\rightarrow}$ $(i \in \mathrm{I})$. Then A_C is such that, if $< e_1,\ e_2,\ ..,\ e_N >$ is any permutation of A_C that respects $\rightarrow$, then if S_i is the object state after the application of e_i, the application's integrity invariant $I(S_i)$ always evaluates to true ($i = 1,\ ..N$).

Some events are repudiated: if *repudiated*(*agree*(*e*)), then *agree*(*e*) is not applied.

At one extreme, each e_{i+1}^a $(i = 0,\ ..N-1)$ could be made to *dependOn* e_i^a, leading to total ordering at all alternates. But a judicious use of this relation can lead to greater concurrency between operation at different sites.