

An Open Architecture for Replicated Shared Objects

Tim Kindberg, September 1997

1. Introduction

This paper outlines an architecture to support object-sharing over the Internet. Our design covers a framework for supporting replicated, persistent objects written in Java. Our thesis is that, since an architecture for large-scale object sharing needs to be open, it should use events as the main interaction mechanism for objects.

Mushroom¹ [1-4] is a system platform for network ‘places’ – electronic workspaces containing shared objects. We call the shared objects mushlets (!), although we shall sometimes refer to them simply as objects. Mushlets are coarse-grained – for example, documents or electronic whiteboards, rather than individual components in a CAD database. They are either group-aware (for example, shared editors), or they act as wrappers for legacy resources such as word-processed documents and HTML files. Mushlet state is replicated at peers (users’ workstations) – for efficient access – and servers, which maintain persistent and authoritative copies. Both Mushroom and mushlets are written in Java.

Inter-machine communication in Mushroom is largely via events (although mushlets may also use RMI internally, between replicas). A simple event describes a state of affairs, for example “Sid added the paragraph ‘.....’ at position p in document d”, encoded as a set of attribute values. Events differ from invocation messages in that:

- they do not designate a target object (although they may mention one or more objects); instead, they are delivered wherever an interest has been declared;
- 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.

In Mushroom, programmers use events to describe a change that affects the replicas of an object. The implementations of the ‘replicas’ of a given mushlet such as a shared document can be heterogeneous. For example, on announcement of the “Show URL X” event, one user could be shown the page in a Netscape browser and another in Internet Explorer. Another advantage of events is that they provide a convenient way to schedule invocations upon distributed objects. If we order event-delivery 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]. Finally, 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.

We go on to describe in Section 2 the requirements for our event-based object replication framework, which we relate to existing work; Section 3 gives an outline of the architecture. We end in Section 4 with a discussion of outstanding issues.

2. Requirements and related work

Mushroom users have several requirements for mushlets:

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

- persistence between work sessions
- high availability during work sessions
- some degree of consistency between replicas at different sites, and between different mushlets; but the level of consistency can vary between mushlets
- minimal delays in retrieving and updating mushlets, but users may accept the potential for inconsistency in order to make progress with their work [8]
- ‘awareness’ information which is at a higher level than notification about a particular update to a particular mushlet; for example, users may want to know when a user joins or leaves a working session; or, they may need to see the entire update history of a document.
- privacy and access control.

Several systems support generic object storage and event distribution services for shared workspaces. Bayou [9] provides an object store with data integrity and availability guarantees, but it provides only a simple database as its notion of ‘workspace’, and is not intended for 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.

In Mushroom, we believe that there will often be a better source to obtain object state from than a central server. For example, if Sid requires an object that is not too big, then he may be able to obtain it from a colleague’s workstation down the hall without inconvenience to her. Or, Sid may be able to obtain the object from a server that is in my domain or reasonably nearby, which has cached the object after obtaining it from across the world on behalf of someone else nearby.

Furthermore, we believe that it is advantageous for servers to run mushlet code, which Java makes possible. Servers can then maintain the current state of each mushlet². This is often more economical on state transmission costs (clients do not have to supply the entire state every time there is an update); it relieves clients from having to catch up on update processing when they join a session; and the server can support application-specific policies on the parts of an object to be supplied, either for security or efficiency reasons.

The Cambridge event system [5, 12] uses servers to dispatch events to just those clients that have registered interest in them. Servers maintain event filters, in other words. This design trades off a reduction in client processing load against event propagation latency. All events pass through a server before they arrive at clients. Therefore a client must wait until the server has evaluated some number of filters and has sent the event to some set of other clients, before it receives it. The saving is that the client has not had to filter and throw away useless events.

At the other extreme, Mushroom’s only event propagation method hitherto has been to use a multicast group called a session as the transmission unit for an event. Sessions correspond to a collection of one or more objects, and events concern the objects and the users who access them. When a peer announces an event, it does so on a session. The event is delivered to all peers that belong to that session; the recipients are responsible for filtering and throwing away redundant events. This design relies on close-coupling between users. The assumption is that users’

²If a mushlet occurs in several heterogeneous forms, then it may be necessary for servers to run each type of replica.

requirements for event registration falls naturally into clusters: you and I tend to overlap a lot in our interests, if we overlap at all.

We suspect that the actual filtering requirement lies somewhere between the two extremes. Users' needs will usually be met by sessions, corresponding to their collaborations with one another. But there are circumstances in which filtering closer to the event source is more appropriate. For example, if Fred wants to know when a particular mushlet changes, but he is not involved directly in the collaborative workspace where it resides, it should not be necessary for him to receive all events transmitted on that workspace's session.

Finally, 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 she should subscribe to. Mushroom sessions are akin to CORBA event channels [13], in allowing transparent many-to-many event propagation.

3. The Mushroom Architecture

3.1 Sessions, Mushlets and Replicas

The unit of management of mushlets is the session. We can think of this as being analogous to a PerDiS cluster [14]. Each session maintains a directory of all the mushlets contained within it or referenced from it; the directory maps mushlet names to metadata. Although a mushlet may be referenced from many sessions, it belongs to only one; this 'home' session is used to propagate events pertaining to the mushlet, and to locate a current copy of its state. Session directories are essential to allow users to browse without fetching undue amounts of mushlet state. Support for multiple sessions allows the programmer to decide whether a mushlet needs its own session (because it is associated with a lot of event traffic), or whether it can share a session with other mushlets. For example, an event-intensive multi-user game may be given its own session. A collection of source code files, on the other hand, which change infrequently, could share a session.

In general, we replicate a mushlet's state at the peers (user workstations) where it is accessed, and at a collection of servers designated as the persistence domain for the mushlet's session, which hold persistent and authoritative copies of the mushlet's state. As Babaoglu and Schiper [15] observed when motivating their group communication architecture, user workstations are liable to crash, and users may quit software at unpredictable times. Therefore Mushroom keeps only an approximate measure of the membership of a particular peer in a given session, similarly to Cosquer and Verissimo's index of the reachability of a peer [16]. 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 slow-changing.

Since mushlets react to events, it is natural to use events to describe the persistence requirements for an mushlet. That is:

- we can use events as a trigger for recording a snapshot of the mushlet's state on persistent storage;
- conversely, we can classify each snapshot using the set of events that are reflected in it.

Mushlets running at persistence domain members save snapshots to disk according to a per-mushlet policy. For example, the state could be logged only after certain types of events are processed; and the state could be logged either by taking a complete snapshot, or by taking a snapshot periodically and recording the set of events that the mushlet has handled since then. Since the system hands events to mushlet replicas, and since all mushlets export a *serialise()* method, there is the potential for the persistence policy to be decided externally to the mushlets themselves.

To access a mushlet, a user must locate a copy of the mushlet state, join the mushlet's session in order to receive events, and bring the copy up to date with more recent events. Peers and servers that are members of the session keep hints about the locations of mushlet replicas. Persistence domain members always possess a copy, but there may be a cheaper copy to be had nearby. Per-domain Mushroom servers (similar to WWW proxy servers) also keep hints about copies that other local session members have requested.

Mushlet state is stored in the form $\langle \textit{flavour}, \textit{snapshot}, \textit{timestamp} \rangle$. The 'flavour' is a label describing the type of state that is stored, where a mushlet allows heterogeneous representations. The snapshot is a serialisation of the object at some logical time. We record the logical time efficiently, by default, as a vector timestamp – rather than as a more specific list of identifiers of events that the mushlet handled. A vector timestamp contains serial numbers of events, indexed by the identifiers of the sites that announced them. In order to minimise the size of timestamps, the persistence domain periodically defines a new epoch, by declaring the set of events deemed to belong to the last epoch. Timestamps are defined relative to the most recent epoch.

By default, all peers and the members of the persistence domain run the same mushlet code. However, for security reasons, it may be necessary to hand peers different state, depending on the rights of the corresponding users. We require mutual protection of session members against users who try to acquire state to which they have no rights; and of peers against those trying to hand them invalid state. We are developing communication protocols to achieve these ends, but we also are addressing the protection issues that arise when executing several Java applications on one host. Our approach is similar to that suggested for PerDiS [17].

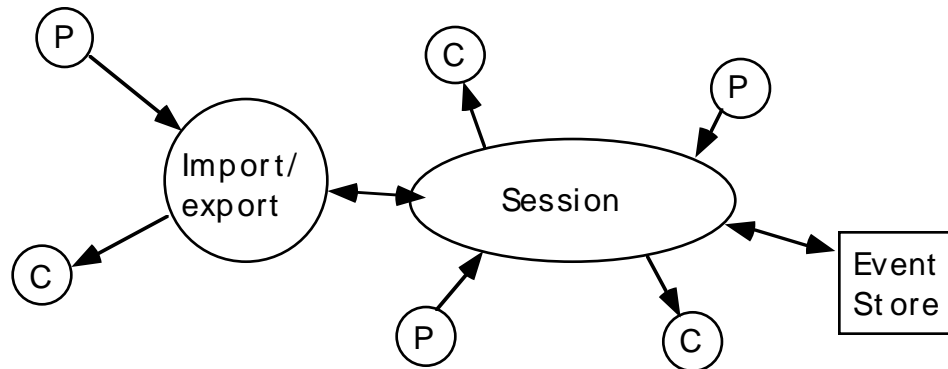


Figure 1. An event-oriented view of the architecture. P = producer, C = consumer.

3.2 Events

Figure 1 shows the overall organisation of the event architecture. It is designed to propagate and store events flexibly and efficiently. Event producers and consumers transfer events over a session³. Events are defined by their Java class (although definition by CORBA IDL is also possible). By default, events are not persistent, and they originate from, and are delivered to, only members of the session. FIFO ordering is the default. However, this arrangement can be extended. First, one or more event stores may join the session to log events – either all events, or ones satisfying certain 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

³ An individual session member may be both a producer and consumer. Although session members are processes, it is individual objects that produce and consume events.

- 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. This includes event ordering (see below) and event synthesis. For example, a membership monitoring service could announce when a user who can take on a particular role has joined the session.

The programming interface for events distributed within a session is:

```
session.setHandler(MeventFilter f, MeventHandler h, int priority);  
session.announce(Mevent e);
```

In the first method, an event consumer supplies a filter object, which has a method to accept or reject the events announced on the session. It declares the object which is to handle the event – typically itself. The priority is used to order event handling (daisy-chain style). The second method causes a copy of the given event to be distributed to wherever a filter accepts it.

Consumers that wish to receive events without joining a session use a different call:

```
session.setExporter(MeventHandler e);
```

The system serialises the object of class `MeventHandler` 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, 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 using RMI.

The system sends the user’s credentials implicitly when handling these calls. By default, only those with sufficient privileges to join a session may establish an exporter object when not a member. However, the following privileged call 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.

```
session.markForExport(MeventFilter f);
```

The server that hosts `MeventHandler` 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.

In a similar vein, we allow those with sufficient privilege to mark events for import:

```
session.markForImport(MeventFilter f);
```

The instance of `MeventFilter` 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.

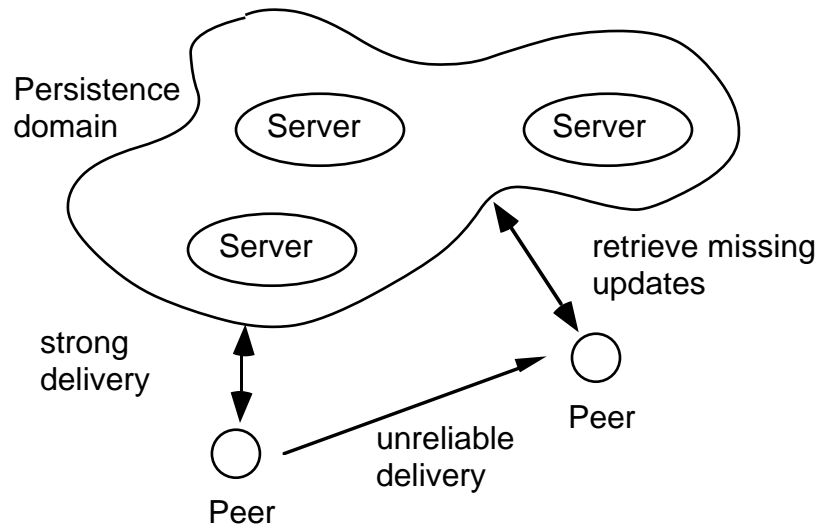


Figure 2. Event delivery.

3.3 Event delivery

The event delivery system has to trade off high Internet latencies against potentially expensive delivery guarantees, such as ordering. A given event may be consumed by different mushlets with different requirements. For example, one mushlet may not suffer from missing an event that is critical to another mushlet; one mushlet will break if event ordering constraints are violated, whereas another does not care. The event delivery system has to accommodate this heterogeneity of requirements as far as possible.

The event delivery architecture reflects the unreliability of peers that we mentioned above, as Figure 2 shows. An event is deemed not to exist unless the members of the session's persistence domain agree it to exist. The semantics of that agreement includes storage and ordering guarantees. They may range from storage at any one of the servers (for eventual delivery to the others), to storage at all servers and agreement on the total ordering of the event. A member of the persistence domain transmits an agreement message once agreement has been reached. This identifies (rather than contains) the event. As Figure 2 shows, delivery to the persistence domain is in general 'strong'; that is, it fulfils the guarantees necessary for agreement to be possible.

In order to accommodate differing latency/guarantee trade-offs, we allow transmission of events to proceed concurrently with agreement. Depending on the semantics of the application, a mushlet may decide to handle the event before agreement. It could, for example, be used to represent a tentative state of affairs, such as the introduction of a mushlet into the session with a tentative name. A session transmits events to peers unreliably. A peer which receives an agreement message for an event that it has not seen before can obtain the event.

As well as the default of FIFO event ordering semantics, and total ordering (with respect to a session, not the collection of all sessions!), we believe that application-specific ordering will be required. The announcer of an event can declare a dependency:

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

– correspondingly, a recipient can call:

```
event.await();
```

– to block the caller until those events on which it depends have been handled.

4. Discussion

We have motivated the use of events as the principal means of interaction for an architecture to support shared, replicated, persistent objects. We have also advocated sessions as the unit of clustering for shared objects and the events that pertain to them. Sessions maintain directories of shared objects – for example, those used for the purposes a particular collaboration – and they act as a channel for the associated events.

Is it possible to express all policies for replication, persistence, concurrency control and security usefully in terms of events? Evidently, we cannot express these policies in terms of events alone – we also have to express constraints regarding mushlet state location and transmission. However, an event-based approach seems promising in that we can:

- coordinate (potentially heterogeneous) mushlet ‘replicas’ by distributing events to them
- express a persistence policy for a mushlet in terms of the events that it consumes
- apply concurrency control by ordering event processing, and by merging streams of events that users generated during disconnected operation
- implement access control by controlling which events may be legitimately announced and observed by which principals.

There are options for who or what should specify policy:

- those who define the event
- those who announce the event
- those who handle the event.

There may be tensions between these requirements and options. Also, an open system is one that exports a model of its own behaviour, which programmers can amend in a principled and dynamic way (and not by hacking source code). We do not yet have answers to the question of what the model should be, or how to resolve tensions between requirements.

Finally, many engineering issues remain to be resolved. In particular, we shall investigate security and data integrity (including concurrency control) aspects further in a forthcoming project. As well as intra-session security and integrity, we shall investigate further suitable mechanisms for importing and exporting mushlets and events between sessions.

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] Terry, D., Theimer, M., Peterson, K., Demers, A., Spreitzer, M., and Hauser, C. (1995). Managing update conflicts in a weakly connected replicated storage system, *Proc. 15th ACM symposium on operating systems principles*, pp. 172-183.

- [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] T. Harrison, D. Levine, and D Schmidt (1997), The Design and Performance of a Real-time CORBA Object Event Service. To appear, proc. OOPSLA '97, Oct. Available at <http://www.cs.wustl.edu/~schmidt/oopsla.html>.
- [14] M. Shapiro (1997), A seamlessly-integrated large-scale Java system. Proc. Workshop on Persistence and distribution in Java, Lisbon, October, <http://www.perdis.esprit.ec.org/events/java-wkshp-971020/>.
- [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] F. Cosquer, and P. Verissimo (1995), Large-scale distributed support for cooperative applications. Proc. European Research Seminar on Advances in Distributed Systems, pp. 105-110.
- [17] G. Coulouris, J. Dollimore, T. Kindberg, and M. Roberts (1997), PerDiS security vs Java. Proc. Workshop on Persistence and distribution in Java, Lisbon, October, <http://www.perdis.esprit.ec.org/events/java-wkshp-971020/>.
- [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.