A Stake in Cyberspace

Tim Kindberg

Dept. of Computer Science, Queen Mary & Westfield College, University of London
Mile End Road, London E1 4NS, UK
timk@dcs.qmw.ac.uk

1. Introduction

Consider the information 'terrain' in which we browse and interact over the Internet, principally the World Wide Web but also USENET, email and IRC. This differs in several basic ways from the territory that we are used to:

- We can be – and often want to be – in several places at once.
- The information terrain is hard to navigate. It is infinite and heterogeneous. There are only rudimentary maps. It is in a state of flux, and is not sharply distinguished from the artefacts it sustains.
- Currently we spend most of our time wandering alone where there is much evidence of human activity, but where the natives themselves and their tools are nowhere to be seen. When we do encounter others, we pass low-bandwidth messages back and forth, often knowing little about our interlocutors.

The thesis of this position paper is:

- The most useful notion of the terrain is the combination of users and the shared information that they interact with, not the information alone.
- Information is where users should be able to encounter other users with similar interests. We need to support collaboration and other forms of social interaction between users, who either meet while browsing information or who are already members of a group.
- The key to collaborative information sharing and interaction on the Internet is the concept of *boundary*, which encompasses naming, security and integrity of shared data, and user communication and awareness. The navigation problem will not go away, but boundaries enable us to impose structure on the sprawl.

We now present an early snapshot of a design that addresses these points.

2. A framework for sharing objects across the Internet

Mushroom[1] [10] is a framework for collaboration and interaction across the Internet. This framework supports the dynamic creation and management of *Mrooms*, which are environments for collaborative activity and containers for shared objects. We give an overview of the users' model in this section, and go on to describe the system issues in the next section.

Although we base our work on a 'room' metaphor [6, 9], we recognise the need to provide a task-based representation of Mrooms. Users access Mrooms through links on World Wide Web pages, and they navigate between them by following links; however, access control is applied. In Mrooms, users can share applications and information objects such as documents, multimedia presentations and whiteboards. Users also share tools for awareness of and communication with other users in the Mroom. Mrooms are persistent, and users can either interact and communicate synchronously (in the same Mroom at the same time) or asynchronously (users occupy the Mroom at different times, but observe one another's changes to information objects). Objects within Mrooms may be active. For example, in distance-learning a lecturer may create interactive objects for student experiments. Equally, we can use Mrooms to store conventional objects such as PostScript files and word processing documents.

---

[1]UK EPSRC grant GR/K73674 'Support for Collaboration and Interaction via the World Wide Web'.

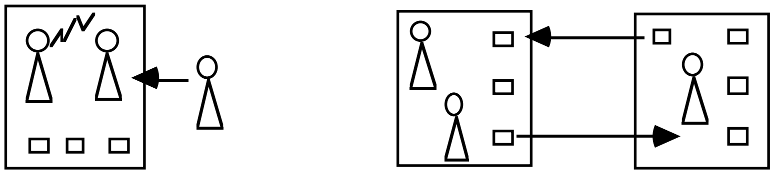

Figure 1. Boundary provides privacy and access control, concurrency and integrity control between Mrooms

Mrooms

To the user, an Mroom consists of a boundary surrounding a collection of shared information objects and objects representing the users who are inside the Mroom. Users observe the Mroom through one or more view objects.

The *boundary* provides security: privacy of communication and access control over the import and export of objects. It also applies concurrency control and more general integrity constraints when importing objects (see Figure 1). The boundary is represented in the user interface. Users drag user representations and information objects across it to import or export the objects themselves, or ones derived from them. (Conceptually, objects enter or leave Mrooms.) An Mroom has one or more owners, who alone are able to configure the controls exercised by the boundary.

Users outside an Mroom can, subject to access control, browse the Mroom in order to find out attributes of the objects inside it and read copies of them. In this case, the Mroom serves a function that is a cross between a Web page and a directory.

A user enters an Mroom in order to interact: other users in the same Mroom can see a representation of the user in the Mroom and can communicate with him or her; and the user can interact with the objects in the Mroom, subject to access control and concurrency control constraints. In particular, users in the Mroom may join *sessions*, which are groups of users and information objects that they update, asynchronously or in real time. Sessions allow several related activities to take place without the user or system overheads of creating several Mrooms. For example, a group of users could collaboratively edit a shared document, using a whiteboard to explain their ideas to one another. Several such groups could work on alternative versions of the document in parallel sessions in the same Mroom. Sessions may also be shared between Mrooms, for interactions between users with limited mutual trust or interests.

A *view* gives a task-based representation of users and information objects inside the Mroom. Typically it represents a desktop-like arrangement; but it could equally be a document or a 3D graphical representation of a virtual world. Users select their *perspective* on the Mroom through the view – for example, a user can look at a plan perspective showing all the objects (including sessions), and later at the objects and users in a particular session.

Mroom contents

In principle any type of object may be brought into an Mroom for sharing. Mushroom provides a concurrency control and integrity framework but leaves the choice of mechanisms and policy to the user. An Mroom is a unit of consistency and, more generally, data integrity. By definition, users within an Mroom can perform no operations that could render its contents inconsistent. This is because in the Mroom:

- individual (group-aware) objects keep themselves internally consistent
- to avoid inter-object inconsistency, the users who manage an Mroom are responsible for setting configuration parameters to rule out *in situ* operations that could potentially make one object inconsistent with another.

Users needing to transform an object (or several objects) in a way that may make the Mroom's contents (temporarily) inconsistent drag it into another Mroom to work on it, and then drag it back into the original Mroom to attempt to re-integrate it. So users attempt to install new *versions* of objects into Mrooms. Users can create a *cloned* (c.f. 'forked') copy of the source Mroom to work in, in order to make a new object-version in a familiar and complete environment. For example, I am not allowed to modify source code *in situ* in a software configuration Mroom, so I create a clone of the Mroom, and update and test my files there. I then drag the new files back for re-integration.

When the user attempts to drag an object between Mrooms, each boundary object (which can consult the dragged object) decides whether this is possible, and if so what the effect is. The following are some possible effects:

- the object is copied and the copy is placed in the destination Mroom
- the object is migrated to the destination Mroom
- a modified copy is produced: I try out a full version of Word v.8 in the Microsoft demonstration Mroom; but when I drag it to my home Mroom I obtain a limited-lifetime demo version
- a proxy to the object is placed in the destination Mroom – so that users in that Mroom can join in a session in which it is updated
- the object becomes locked, and will refuse to be dragged across the source boundary if another attempt is made before it has been dragged back into the Mroom
- the destination Mroom refuses to admit the dragged object, because of access control settings
- the destination Mroom refuses to accept the dragged object because of a consistency conflict or other integrity conflict with the existing contents.

Dragging several objects into an Mroom is atomic. When an attempt to import one or more objects fails, the Mroom may invoke users to assist in integrating them, should this be possible. Integration may involve modification of the object itself or its attributes (including its name).

Mrooms are not simple shared directories or desktops. Jim is motivated to enter an Mroom in order to interact with objects (including users) within it; other users then become aware of Jim. By clicking on Jim's representation, Mary can see a summary of his activities and can communicate with him. When Jim drags an object to his Mroom to work (optimistically) on a copy of it, the system places a link to his Mroom next to the object in the source Mroom, so that users may view Jim's Mroom (and hence at least a snapshot of his activity). They can communicate with him if necessary by entering his Mroom.

3. System architecture

An Mroom's state consists of the following objects:

- the Mroom *directory* -– which maps names onto attributes for objects in the Mroom
- the objects (user representations and information objects) within the Mroom:
    - immutable object *versions*, which may be freely replicated
    - mutable, non-group aware objects, which are copied entirely when updated
    - mutable, group aware objects, which are dynamically updated in place.

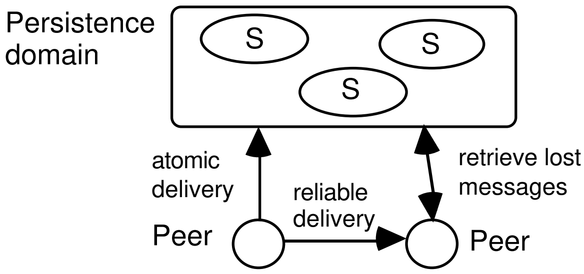

Figure 2. Distributed system architecture (*S* = server).

The Mushroom architecture is made up of servers and peers (Figure 2). Servers maintain persistent Mroom state and naming information; peers maintain volatile replicas of the state and update it – typically, a peer runs at a user's workstation. It is expected that there will be one or more server machines per DNS domain. Objects are replicated at servers in several participating DNS domains to increase their availability.

The main architectural unit is a *session* (which is directly related to but not to be confused with the user-level concept of the same name). A session is specified by a group of peer and server processes, and a collection of shared objects, including an encryption key for privacy. The objects are replicated at the group members, and updated by sending events to the group. Every Mroom has at least a ‘main’ session, containing the Mroom’s directory and a collection of core objects. The core objects include representations of the users in the Mroom, and an object that controls the import and export of objects (when users drag objects into and out of the Mroom), and other aspects of the Mroom’s shared configuration. Sessions other than main sessions can include any selection of objects; and they may span Mrooms. Whether or not a session spans Mrooms is a question of key distribution: group membership (needed for user participation) requires knowledge of the key.

Servers belong to the *persistence domains* of individual sessions. A server in the persistence domain for a session maintains persistent copies of the session’s objects. Persistence domains are similar to the notion of a core group in [1]. However, for the sake of efficiency we are considering an architecture in which we separate the notions of group addressing and delivery guarantees. Peers send updates to one group – the session’s group – but delivery guarantees to the persistence domain are stronger than to the peers (see Figure 2). We regard totally-ordered delivery to the whole group and atomic delivery to the persistence domain as being highly desirable – at least in the case of Mrooms’ main sessions, so that users can agree on the set of objects in the Mroom and their attributes. We are investigating the feasibility of providing these guarantees with acceptable performance in a wide-area collaborative setting.

Mushroom names, directories and attributes

The directory maps names to attributes as follows:

Mroom directory:	name <,version number> –> attributes
attributes =	type
	versionNumber
	MushroomResourceLocator
	integrityControlAttributes:
	entryHandler
	exitHandler
	concurrencyControlState (e.g. locks, notifications)
	accessControlList
	otherAttributes

An object has a textual name which is a path through Mroom directories, optionally qualified by a version identifier. Mushroom maintains versions of objects, which users need to develop documents [2], source code [7] and other types of object.

Mrooms can span organisational boundaries, which suggests that the high-level name space needs to be categorically rather than organisationally partitioned (c.f. the USENET name space). For convenience, however, DNS names are embedded in the name space. One of the chief attributes obtained when resolving a name is a *resource locator*. This specifies a globally unique object identifier and session identifier – any object is always to be found in some session. A session identifier is mapped locally to a session hint, a combination of an IP multicast address and individual host addresses, which sites use to locate the nearest copy of the object.

Objects have the potential to become inconsistent with respect to other objects (or to break other integrity constraints). The system invokes procedures for importing and exporting objects into and out of Mrooms. When a user tries to drag an object into an Mroom, the system calls an *entry handler*, which detects whether a consistency or integrity conflict would arise from installing the object and, optionally, handles the conflict. For example, a conflict can be detected by attempting to compile a new version of a source code file, when it is dragged to an Mroom containing the source code tree. A handler may advise the user about the conflict. It may even update the dragged object and perhaps other objects to resolve the conflict – similar to a *merge procedure* as used in the Bayou replicated storage system [13]. The *exit handler* is used, for example, to install a link to the destination Mroom so that others are aware that the object is being worked on, or to implement locking. We prefer optimistic techniques for collaborative working, coupled with mechanisms for keeping users aware of one another's concurrent activities.

Replication

For our model of replicated object interactions we adapt the standard object-oriented Model-View-Controller (MVC) architecture [11], normally used for single-user applications. At a single site, the state of the application belongs to a collection of objects called the model. The user-interface objects are kept in a separate collection called the view. When a user interface action occurs (for example, a user presses 'delete'), a controller object is created to handle the interaction by providing feedback through the view and by updating the model.

In adapting the MVC paradigm to a collaborative application we must take account of the fact that users do not necessarily share views or even models. For example, I could use a desktop-based interface to our shared objects, while you use a VR-based interface. Whatever the view, normally some application state is not shared but personal (it represents personal settings). Users may even use different models providing similar semantics. A further requirement, which arises particularly in wide-area collaboration, is to accommodate the high latencies of operations.

An *event* is a set of labelled attributes which describes a new state of affairs and can be transmitted in a message. It does not name a handler object or a method to be invoked, but is used to dispatch an invocation according to the attribute values. The need to accommodate heterogeneity between peers suggests using events as a means of updating other users' models in a session, instead of invocations on named replicated objects. Events are also attractive from the point of view of scheduling atomic updates to shared state: instead of scheduling invocations on individual replicated objects, we order events, and each site schedules event processing locally, according to ordering constraints. Despite the advantages of events, we face the problems of: deciding which events need to be propagated to other sites; translating local events into globally recognised event attributes before transmission; and eliminating all but one announcement of the same event originating from replicated peers.

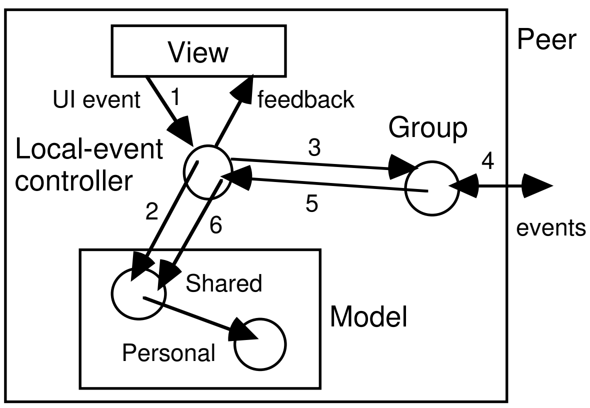

Figure 3. Handling an event originating locally (1); tentative update (2) is followed by a committed update (6) after communication via group (3,4,5).

To accommodate high latencies while meeting the standard user-interface requirement of effective feedback, we need to issue the effects of an operation locally as a tentative update, until the system has committed it (Figure 3). The system designates changes to the model as *tentative* initially, and later, when the event has been ordered and delivered to sufficient sites, as *committed* or *repudiated*. A repudiated update is one involved in a conflict: for example, you and I concurrently add different objects called "fred". The system represents changes as tentative through the view. For example, objects that have been dragged into the Mroom could appear faded until their incorporation in the directory is committed or repudiated; objects that are tentatively deleted could appear with a line through them.

4. Discussion

Compared to other systems employing similar room-based metaphors (e.g. MUDs and MOOs [5], CoDesk [9], Worlds [14] and BSCW [3]) the Mushroom design differs in its concern with the integrity of shared information objects, and in its focus on a scalable and flexible system architecture through using replicated state, group communication and event-based updates. Lotus Notes [8] provides for replication, but its support for collaboration is limited. There is no systematic way of providing awareness of users' activities. Support for conflict management is rudimentary. Update propagation occurs according to predetermined schedules, precluding synchronous working.

Many system issues remain to be resolved in the design of Mushroom, including the details of the design as we outlined it above, and the additional issues of security and dealing with partitions. A malicious or faulty site can arbitrarily change a local copy of an object, regardless of the user's supposed access rights. Correctly functioning sites need to ensure that only *bona fide* object versions, updates and communication are accepted for integration, application and audio or video rendering. The security issue for group working is under investigation [12]. The question of behaviour under partitions is addressed by [4], based on view-synchronous group communication. But more investigation is needed into user requirements for dealing with partitions.

We are currently engaged in a feasibility study investigating these challenges, including the construction of a prototype using the Java language and IP-Multicast-based software packages. Mushroom is being integrated with the Web as a 'helper' application for Web browsers.

5. References

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

[2] Bäker, A. and Busbach, U. (1996). DocMan: a document management system for cooperative support. *Proc. Hawaii Int. Conf. on System Sciences VIII*, Jan. 1996, IEEE Computer Society Press, pp. 82-91.

[3] Bentley, R., Horstmann, T., Sikkel, K., and Trevor, J. (1995). Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace system. *Proc. 4th International WWW Conference*, Boston, December 1995.

[4] Cosquer, F., and Verissimo, P. (1995). Large-scale distributed support for cooperative applications. *Proc. European Research Seminar on Advances in Distributed Systems*, pp. 105-110.

[5] Curtis, P. (1993). LambdaMOO programmer's manual. Xerox PARC.

[6] Henderson, A., and Card, S. (1985), Rooms: the use of multiple virtual workspaces to reduce space contention. *ACM Transactions on Graphics*, vol. 5.

[7] Jordan, M., and Van der Vanter, M. (1995). Software configuration management in an object-oriented database. *Proc. USENIX conference on Object-Oriented Technologies*, June 1995.

[8] Lotus (1993). System administration manual, Lotus Notes release 3, 1993.

[9] Marmolin, H., Sundblad, Y., and Pehrson, B. (1991). An analysis of design and collaboration in a distributed environment. *Proc. ECSCW 1991*, pp. 147-161.

[10] Kindberg, T. (1996). Mushroom: a framework for collaboration and interaction across the Internet. *Proc CSCW and the Web*, *5th ERCIM/W4G workshop*, Busbach, U., Kerr, D., and Sikkel, K. (eds), GMD, pp. 43-53.

[11] Krasner, G., and Pope, S. (1988). A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *JOOP*, vol 1, no 3, August/ September, 1988, pp 26-49.

[12] Rowley, A. and Dollimore, J. (1996). Replicated secure shared objects for groupware applications. Tech. report 716, Dept. of Computer Science, Queen Mary & Westfield College, University of London.

[13] 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.

[14] Tolone, W., Kaplan, S., and Fitzpatrick, G. (1995). Specifying dynamic support for collaborative work within WORLDS. *Proc. ACM Conference on Organisational Computer Systems*, August 1995, pp. 55-65.