
# Architectural Support for Cooperative Multiuser Interfaces

**Richard Bentley, Tom Rodden, Pete Sawyer, and Ian Sommerville**
**Lancaster University**

Computer support for cooperative work requires the construction of applications that support interaction by multiple users. The highly dynamic and flexible nature of cooperative work makes the need for rapid user-interface prototyping a central concern. We have designed and developed a software architecture that provides mechanisms to support rapid multiuser-interface construction and distributed user-interface management. Rapid prototyping requires mechanisms that make the information determining interface configuration visible, accessible, and tailorable.

We developed the architecture described here as part of a project investigating support for the cooperative work of air traffic controllers. Extensive use of prolonged ethnographic investigation helped to uncover the nature of cooperation in air traffic control.[1] The aim of the architecture is to support an environment in which a multidisciplinary team can experiment with a wide range of alternate user-interface designs for air traffic controllers. Thus, we use examples from this domain to illustrate the architecture.

**Developing prototypes for multiuser-interface designs can be complex. It helps if the developer can work with a set of simple mechanisms that determine the interface properties.**

## Multiuser interfaces

Many computer systems already support simultaneous interaction by more than one user; examples include multiuser databases, operating systems, and office information systems. But these systems support multiuser interaction in a way that prohibits cooperation. Most existing multiuser systems give each user the illusion that he or she is the only one using the system, thereby maintaining a "protective wall" to hide other users' activities. To support and encourage cooperation, cooperative applications must allow users to be aware of the activities of others. The purpose of a cooperative multiuser interface is to establish and maintain a common context, allowing the activities of one user to be reflected on other users' screens. This common context is achieved by sharing application information; the real-time presentation and manipulation of shared information is the main function of cooperative multiuser interfaces.

The extent to which multiuser interfaces support sharing through the propagation of activities is termed *interface coupling*; the greater the level of awareness between users, the closer the interface coupling. The following three levels of sharing correspond to different degrees of interface coupling:


0018-9162/94/$4.00 © 1994 IEEE

## Status report

The user display model and agent architecture described here form the basis of a multiuser-interface prototyping environment called MEAD (Multiple Electronic Active Displays). The latest version of MEAD (version 2.0) exists as a working prototype developed in Objectworks\Smalltalk Release 4.1 for Sun workstations. It is being used in several research projects at Lancaster University and elsewhere. One of these is Comic (Computational Mechanisms of Interaction for Cooperation), an ESPRIT research project involving 10 partners that is developing supporting theories and techniques for CSCW. Within Comic, the system is being used to investigate sharing in cooperative systems and, in particular, the relationships between sharing information and sharing awareness of other users' activities.

For more information on the availability of MEAD or the Comic project, contact Tom Rodden at the address shown at the end of the article.

- *Presentation-level sharing* (tight coupling): Each user is presented with the same display of the same information from a common information space. When this presentation is changed in any way, all display screens are updated. This level of sharing is also known as "what-you-see-is-what-I-see" (WYSIWIS).
- *View-level sharing* (medium coupling): Each user has presentations of the same information, but the presentations may differ. For example, different users may simultaneously interact with tabular or graphical displays of the same data.
- *Object-level sharing* (loose coupling): Each user has presentations of different information. For example, several users may each edit different sections of the same document.

Lauwers and Lantz[2] describe two approaches to developing multiuser interfaces. The first allows single-user applications to be shared in a collaboration-trans-

## Collaboration-transparent and collaboration-aware user interfaces

It is important to distinguish between two broad classes of multiuser interface. The first lets multiple users work cooperatively with existing single-user applications. The second involves the development of special-purpose applications that handle collaboration explicitly. Lauwers and Lantz[1] identify these approaches as *collaboration transparency* and *collaboration awareness*, respectively.

Some applications can be modified to run in a multiuser setting. This modification provides geographically dispersed participants with access to sophisticated tools to facilitate group work. Thus, systems have been developed that support transparent sharing of applications, often called shared applications.

This approach was developed to allow each user's screen to be shared with others. As windowing systems developed, this shared-screen approach was extended to permit sharing of individual windows. The figure shows the logical structure of such a shared-window system.

A central conference agent is responsible for multiplexing display output and demultiplexing user input so that the application deals with a single stream of events. This sharing is transparent to the application, a condition achieved by allowing only one user to interact with the application at any given time (part b in the figure). Borrowing some business-meeting terminology, this user is said to "have control of the floor." Floor control must be passed to other users before they can interact.

Floor control is the responsibility of the central conference agent and is identified as its *chair management role*.[2] Not surprisingly, much of the work in supporting collaboration transparency has focused on floor control — for example, the development of different turn-taking protocols and floor control policies.

Collaboration-aware solutions provide facilities to explicitly manage information sharing, allowing information to be presented in different ways to different users. Often, the application itself manages each user's sharing. Applications shared in this way are multiuser applications.

The logical centralization of user interface management embeds within the application a set of decisions as to how information is presented and modified. These decisions form the policy by which information is shared; the embedding of this sharing policy in the application inhibits tailoring. In addition, the lack of a supporting infrastructure requires most collaboration-aware applications to be constructed from scratch. As a result, this approach to developing cooperative systems tends to be less popular than the collaboration-transparent approach.

**Multiuser interfaces.** The problems of developing multiuser interfaces were initially highlighted in the CoLab project at Xerox's Palo Alto Research Center,[3] which examined the development of appropriate supporting facilities for real-time co-located meetings. Xerox PARC researchers developed several applications that allowed the effects of each user's actions to be shared across several screens. The CoLab project called this principle "what-you-see-is-what-I-see" (WYSIWIS).

Initially, each user's complete screen was shared; however, this was found to be confusing and distracting. The solution was to share only portions of the screen, and a separation was established between shared and private windows. This arrangement is called *relaxed WYSIWIS*, while the initial Colab setting is termed *strict WYSIWIS*. Collaboration-transparent systems replicate display output and adopt an approach based on sharing an application's presentation. Multiuser interfaces for such systems, therefore, support the WYSIWIS sharing of applications, giving

parent manner, so that no facilities for handling collaboration are embedded in the application. The second involves development of special-purpose collaboration-aware applications that explicitly manage cooperation. Each approach has different implications for sharing (see sidebar).

Developers of a collaboration-aware application must determine how users require information to be presented and how they interact with information representations. Although design decisions can provide flexibility to support different forms of presentation and interaction for each user, they are often embedded within the application and become difficult to amend. While collaboration awareness is necessary for cooperative systems, it is better not to have it embedded in the application.[3] Therefore, supporting architectures that manage information visualization and manipulation outside the application are required. These architectures hide the physical distribution of components from the application developer and allow visualization and interaction policies to be tailored independently of the application.

# Multiuser interface requirements

Two key requirements constrain development of the supporting infrastructure for multiuser-interface development.

**Support for multiple displays.** Cooperating users access shared information through individual workstations, often supplemented by informal communication. Since cooperating users must be aware of each others' activities, a multiuser interface architecture should allow

- visualization of shared information on different users' screens,
- manipulation of shared information on different screens, and
- propagation of user interaction between screens.

To support rapid prototyping, these facilities need to be provided through a set of robust and readily understood mechanisms that allow interface reconfiguration.

**Support for different views.** Cooperating users may require information to be

each user a common frame of reference.[1] Removing this common context can cause problems when users engage in tightly coupled group work.

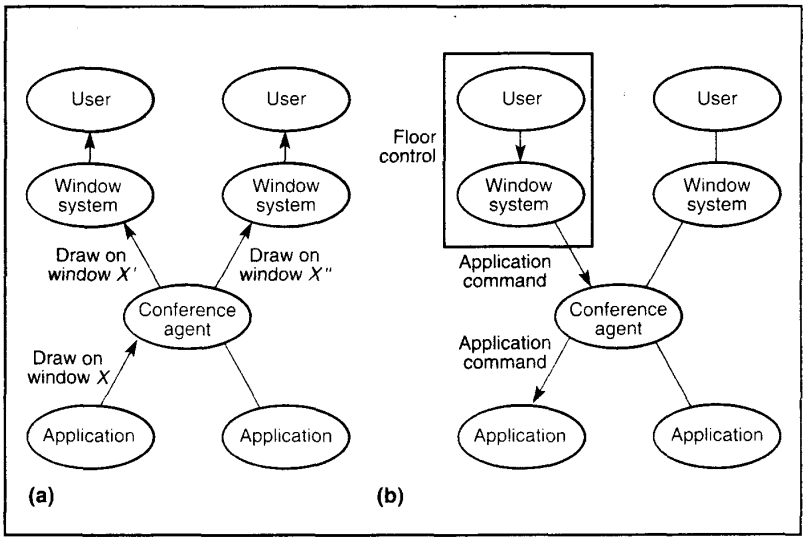
Although some shared-window systems relax interface coupling, a fundamental problem is their inability to support anything other than WYSIWIS information sharing. Where users have widely differing knowledge, roles and attributes, this can be overly restrictive. A view used by a technical person, for example, may have too much detail for effective viewing by a manager.[2]

Different users may require different representations of the information, or may be concerned with entirely different information. This level of flexibility cannot be provided by collaboration-transparent applications, since it requires the system to exploit knowledge concerning the shared task being undertaken. This form of interface is provided by collaboration-aware applications.

Although they can support strict WYSIWIS interaction (as with the Colab system, for example), collaboration-aware systems offer such evident advantages as a range of alternative application presentations across a community of users. To contrast this arrangement with the work of Colab, Dewan[4] calls this style of sharing "what-you-see-is-not-what-I-see," or WYSINWIS.

### References


1. J.C. Lauwers and K.A. Lantz, "Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared-Window Systems," *Proc. CHI 90, Conf. Human Factors in Computing Systems*, ACM Press, New York, 1990, pp. 303-311.
2. S. Greenberg, "Sharing Views and Interactions with Single-User Applications," *Proc. ACM/IEEE Conf. Office Information Systems*, ACM Press, New York, 1990, pp. 227-237.
3. M. Stefik et al., "WYSIWIS Revised: Early Experiences with Multiuser Interfaces," *ACM Trans. Office Information Systems*, Vol. 5, No. 2, Apr. 1987, pp. 147-167.
4. P. Dewan, "Principles of Designing Multiuser User-Interface-Development Environments," in *Engineering for Human-Computer Interaction*, North-Holland, Amsterdam, 1992.


**Logical output architecture (a) and input architecture (b) of a shared-window system.**

presented in different ways that correspond to different levels of sharing. For tightly coupled cooperative work, users may share information at the presentation level. However, sharing may also be required at both view and object levels to support different user or task requirements. As a result, the supporting architecture should allow

- definition of different interactive representations of shared information entities,
- maintenance of these representations as underlying information changes, and
- updating of information entities through interaction with their representations.

## Support for multiuser-interface development

Most existing approaches to developing cooperative systems consider user-interface development to be a task for application developers and therefore provide little support for user-centered development and tailoring. Because the focus of cooperative interface development is on information sharing, we need support for experimentation with different visualization and interaction techniques. Consequently, any architecture to support rapid prototyping should make visualization and interaction details explicit and decouple them from application concerns.

Along with rapid refinement of interface designs, prototyping also involves evaluation of interface prototypes in realistic settings. Hence, in addition to development tools supporting interface construction, we also need mechanisms to execute prototype interfaces. These mechanisms should make visible the policy for visualizing and interacting with information (the sharing policy) to allow high-level tailoring and rapid refinement of interface prototypes. Some general design principles for single-user interfaces have been identified, but few have emerged for multiuser-interface development. However, multiuser interfaces must support a number of key features.

(1) *Separation.* An accepted feature of single-user interface architectures is the separation of user-interface and application components. This arrangement has many advantages:

- *Reusability.* Both user interface and application can be reused independently.
- *Customization.* The interface can be tailored in isolation by both developer and user.
- *Portability.* The application may be portable while the interface is device dependent.
- *Multiple interfaces.* The same application can be manipulated by different interfaces.

Logical separation is desirable for single-user interfaces, but the multiuser case requires such separation to support the alternative representations needed for view-level sharing.[4] In addition, physical separation provides a degree of fault tolerance; an interface process can fail without affecting other interface or application processes. Physical separation also allows execution of interface and application processes on different machines, providing local feedback and supporting a high degree of user adaptation.

**Cooperating users may work better with interface representations they can tailor individually.**

(2) *Feedback and feedthrough.* Most user actions require display feedback; the form of such feedback may depend on the semantics of the application. When the application and user-interface components reside on different machines, the feedback loop involves transmission over a network. It may therefore be hard to achieve acceptable response times.

Feedthrough is the updating of users' screens in response to actions performed by other users working on different machines. Multiuser interfaces must support rapid feedthrough. The importance of this feature depends on the granularity of the updates broadcast to other users. In tightly coupled cooperative activities, such as group drawing, the process of creating an object and the associated explanation and gesturing are often as important as the resulting object.[5] Where such tight coupling is required, the granularity of updates is small, and rapid update feedthrough is vital. For more loosely coupled activities, courser grained updates may be acceptable.

(3) *End-user tailoring.* Cooperating users may adopt different working methods even when performing similar tasks. Therefore, the interface designer cannot provide interface representations appropriate to all users in all contexts. One solution is to let users tailor their interfaces to suit their requirements.

These features, combined with the requirements of cooperative interfaces, motivated the design of our architecture, which supports multiuser-interface prototyping. Multiuser interfaces exist within distributed environments and support simultaneous interaction by several users. Interface requirements must therefore be sensitive to properties of the supporting distributed infrastructures. Consistency between information displays and the information itself must be maintained, and mechanisms must be provided to handle change propagation.

## Supporting infrastructure

Architectures for multiuser interfaces originate from distributed-systems research and must address problems such as network delay, loading, and latency. Between the two extremes of centralization and replication, lies a continuum of hybrid architectures.

**Centralized architectures.** In a centralized (or client-server) architecture, a central server program handles all user input and display output events, which are routed via local client programs. Local workstations act as graphical terminals and window servers. The master-slave architecture is a variant in which one client is merged with the server and all other nodes run as clients.

The primary advantage of the client-server approach is simplicity: The application and all data are held centrally, simplifying access management and data consistency. Implementation is easier still with a networked window system such as

X Windows. Shared-window systems such as shared X[6] use this approach, which has been widely adopted by computer-conferencing systems.[7] It is relatively easy to support presentation-level sharing because the server can replicate display directives to all clients.

The client-server approach can also support view-level sharing. For example, Rendezvous[3] is based on a client-server architecture with all user interaction and display management handled centrally. Each user has an associated view process that interprets input events and display directives. The sharing policy is detached from the information being shared, since each view process can interpret events and display directives differently, supporting alternative information representations.

Because the sharing policy is embedded in the central server, the system developer is responsible for interface tailoring. Rendezvous does not make the sharing policy visible, which severely limits end-user tailoring. The centralized architecture is also vulnerable to delayed feedback and failure of the central node (or the network connections to it), since all events must travel over a network.

**Replicated architectures.** At the other extreme, replicated architectures maintain exact copies, or replicas, of the application on each workstation. To maintain consistency, each replica handles screen management and feedback locally and broadcasts any change in application data to all other replicas. Local display management means that different views are easily supported. End-user interface tailoring is relatively easy to provide, as each replica can adapt its sharing policy to the user's preferences.

The major difficulties with replicated architectures concern synchronization and data consistency. Users can perform actions simultaneously that are executed locally before being broadcast to other machines. If these actions conflict — for example, one user deletes the selected object in a group drawing program just as another user changes the selection to a different object — inconsistent interfaces can result from events arriving in a different order at each machine.

Preventing such race conditions requires complex synchronization algorithms. The standard distributed-systems solution is to use a global clock to timestamp each event and then execute rollback should inconsistency arise, replaying events in temporal order. This is unacceptable for multiuser interfaces where screens may have been updated, so alternatives based on transforming updates to prevent rollback have been developed.[8]

A further problem occurs when users wish to join a group session already in progress. This dynamic registration is straightforward with a centralized approach, since new clients need only contact the central server. The server can then broadcast the current state of the application to bring the new client up to date. With a replicated approach, however, a new replica must contact all other replicas to tell them that it needs to receive updates. This means that new replicas must know or be able to learn the locations of all other replicas.

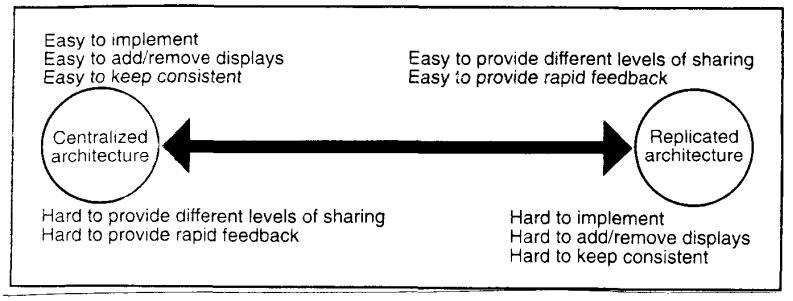



**Figure 1. Summary of distributed-architecture characteristics.**

**Hybrid architectures.** Both centralized and replicated architectures offer benefits and limitations. Because neither architecture fully meets multiuser interface requirements, a hybrid solution is needed wherein components of the cooperative system are either centralized or replicated, depending on the application requirements. A continuum of such hybrid arrangements exists between the extremes of centralization and replication. Figure 1 summarizes the characteristics of different distributed-systems architectures.

Our objective was to develop an architecture that supports multiuser interfaces and also provides runtime support and facilities for interface tailoring. This architecture is based on autonomous agents that encapsulate details of the sharing policy to manage multiuser interfaces independently of the application's behavioral semantics. The architecture is a hybrid in which shared information is kept consistent in a centralized component while the presentation and interaction semantics are replicated in the distributed agents.

# User display agents

Our architecture considers users' information displays as autonomous entities with properties that can be tailored by interface developers and users. The states of these entities characterize the way information is presented to users, who interact directly with the entities to update the underlying information. These updates are immediately propagated to other users' screens to maintain consistency. We refer to these entities as *user display agents*; the parts of a user's screen managed by an agent are called the *user display*.

Users browsing and manipulating a shared information space each hold a working set of UD agents. Each agent manages one display of the shared information and can present this UD in multiple screen windows (see Figure 2 on next page). An agent can be a member of several working sets, so that the UD it manages can be displayed on multiple screens. Agents can be added to and removed from working sets as required.

Since agents can display their UDs on multiple screens, it is possible to support presentation-level information sharing. Different UDs can be designed that represent the same information in different ways to support sharing at the view level. Object-level sharing is possible because users can have completely different UD agents in their working sets. The mechanism separates what is being shared (the application information) from the sharing policy (the presentation of information and the means of interacting with it). The sharing policy can be tailored independently of the application, since UD agents support collaboration-aware sharing without requiring the application to handle the sharing itself.

**Properties of a user display.** UDs support information sharing between multiple users and allow updating of informa-

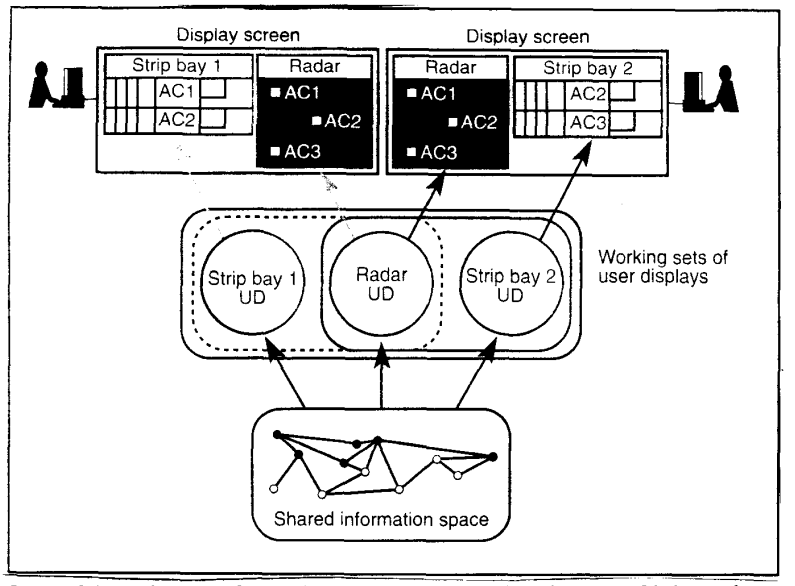



**Figure 2. Working sets of user display agents cooperating via a shared information space. (AC1, AC2, and so forth, in this and subsequent figures are used to distinguish abstract objects representing, in this case, different aircraft and the associated visualizations.)**

tion through interaction with local representations. The need to support different levels of sharing and interface coupling highlights three effects that updates have on shared information:

- *Focus.* Cooperating users may be interested in only a subset of information from a shared information space. Information representations may need to be added to or removed from users' displays dynamically.
- *Representation.* While cooperating users may require the same information, the way it is represented may vary, depending on the tasks being supported and the user's level of experience and domain knowledge.
- *Position.* The spatial arrangement of representations on users' screens may provide information about relationships between information entities. This arrangement may have to be adjusted to reflect changes in these relationships.

An example of a UD is an interactive radar display showing the geographical position of aircraft in a portion of airspace on a two-dimensional screen. Aircraft longitude and latitude data is mapped onto the *x-y* position of a "blip" representing aircraft location. Blips can contain information such as identification codes and height data and can be manipulated by the user to display additional information about the aircraft they represent.

In terms of *focus*, *representation*, and *position*, modeling the radar display is quite straightforward. Focus is concerned with calibration (that is, the portion of airspace represented), representation with the type of blip used to display each aircraft, and position with the mapping from aircraft location to the position of the blips on the screen. Any change in location may cause changes in focus, representation, and/or position.

**Components of a user display.** The UD model, that is, the concepts of selection, presentation, and composition, directly realizes the concepts of focus, representation, and position. A UD is described as a triple, comprising a *selection*, a *presentation*, and a *composition*:

- A selection is a set of information entities dynamically chosen from an information space according to selection criteria. Selection criteria are predicates over entity attributes; they act as a filter to pick out entities to be displayed. As the state of entities is updated, the selection may change.
- A presentation is a set of views used to represent entities in the selection. A *view* is a graphical representation that defines the appearance of a single entity, the position and representation of that entity's attributes, and the interaction with that entity. Views are dynamically selected for each entity through the application of the presentation criteria. These criteria define a filter for each view that an entity must pass through to be represented by that view. Changes in the state of an entity may require changes in the presentation, that is, the selection of a different view to display the entity on the screen.
- A composition is a set of positions representing the spatial arrangement of views in the UD. These positions can be either absolute or relative to other views. As an entity's state changes, these positions may also have to change to remain consistent with the arrangement defined in the composition criteria.

Using this model, one abstract definition of the radar UD described above might be

- *Selection criteria*: aircraft longitude from $x'$ to $x''$, latitude from $y'$ to $y''$.
- *Presentation criteria*: large blip for passenger aircraft, small blip for private aircraft.
- *Composition criteria*: Map longitude to $x$ position, latitude to $y$ position.

A change in an aircraft's longitude may require a change in selection or composition of the radar UD.

The encapsulation of both the definition criteria and the state of an information display in an autonomous UD entity allows the tailoring of information displays without system reconfiguration. Any changes to a UD's definition criteria result in immediate computation of the new state and updating of users' screens. This tailoring, recomputation, and display management is managed by the associated UD agent.

**Maintaining user displays.** A UD agent can be a member of more than one user's working set, supporting presentation-level information sharing. Each screen representation of the UD managed by such an agent is affected by updates to the shared information in the same way, and thus the effects of updates on selection, presentation, and composition need only be calculated once.

To support this capability, a copy, or *surrogate*, of each UD agent can be held in each user's working set. Surrogates are minimal agents that hold only the state of the selection, presentation, and com-

position of the UD. The UD's definition is held by a *master* UD agent, which receives notification of relevant updates to information entities, uses the definition criteria to compute the effects on the UD, and informs each surrogate of the new state (see Figure 3).

In our implementation, shared information entities are held as objects within an *object store*. All updates to the objects in the object store are handled by the *object store server* (OSS), illustrated in Figure 3. This component also holds the master UD agents, allowing straightforward registration and deregistration of new users. Machines can be added without other users' machines being informed; the new machines need only contact the OSS to register their existence, create the required surrogates, and establish links to the master agents. A machine can deregister by informing the OSS that it no longer wishes to receive update information.

The master/surrogate arrangement of UD agents allows local tailoring of information displays without system reconfiguration. Local tailoring operations performed on a window presenting a UD are retained by the surrogate agent. When the master informs surrogates of updates to the UD's state, they can take any tailoring into account before displaying the new state.

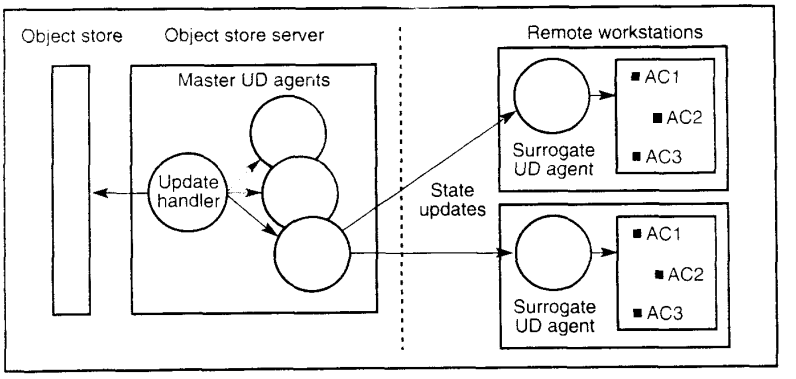



**Figure 3. Master/surrogate user display agent arrangement.**

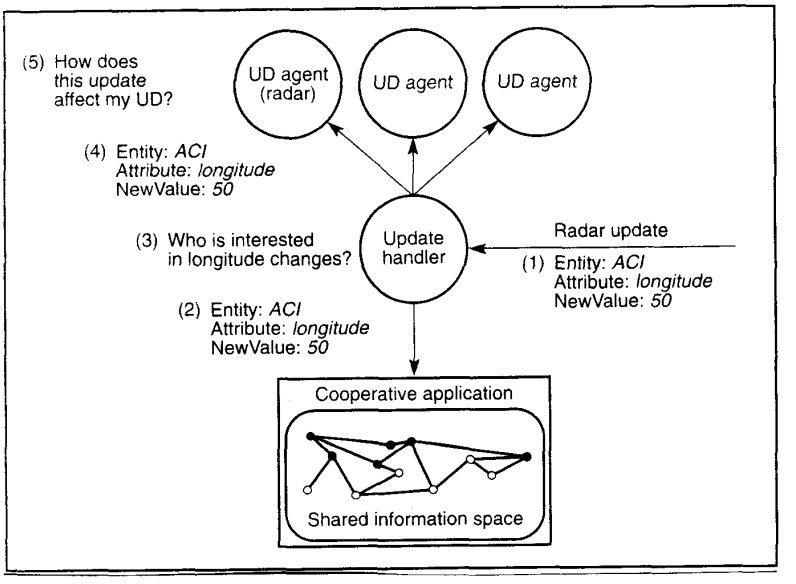



**Figure 4. The dispatching role of the update handler.**

## Consistency maintenance

Updates to shared information objects may affect the selection, presentation, and composition components of each UD. For example, consider the definition of the radar UD described above; a change in an aircraft's position (latitude or longitude) may require it to be added to or removed from the display, or its blip representation may have to move. UD agents must be aware of such updates so that they can use the definition criteria to calculate effects on the state of their UDs and maintain consistency with the shared information space.

There are two options for detecting changes in the state of information entities: Agents can periodically poll the information space, checking to see what has changed, or information objects can notify agents when changes occur. The first option is inefficient when there are many information objects, and the second requires each object to either record which agents are interested in it or broadcast updates to all agents.

The agent architecture uses a variant of the second option that does not require the information objects to notify UD agents of updates. All updates to shared information objects are delivered to an agent called the *update handler*. This agent forwards updates to the object store, as well as notifying potentially effected UD agents. Figure 4 illustrates the dispatching role of the update handler, which ensures that only UD agents interested in an update are notified.

Whenever the interface developer creates a new UD, a UD agent is automatically created to manage it. This agent's first task is to register with the update handler to receive updates. Agents must inform the update handler of their interest set, which it then uses to determine which updates the agent should receive; for example, the radar UD agent defined above will register changes in aircraft longitude and latitude as its interest set. Agents may have to reregister their interest set if the interface developer modifies the definition criteria of their UDs and deregister if their UDs are removed.

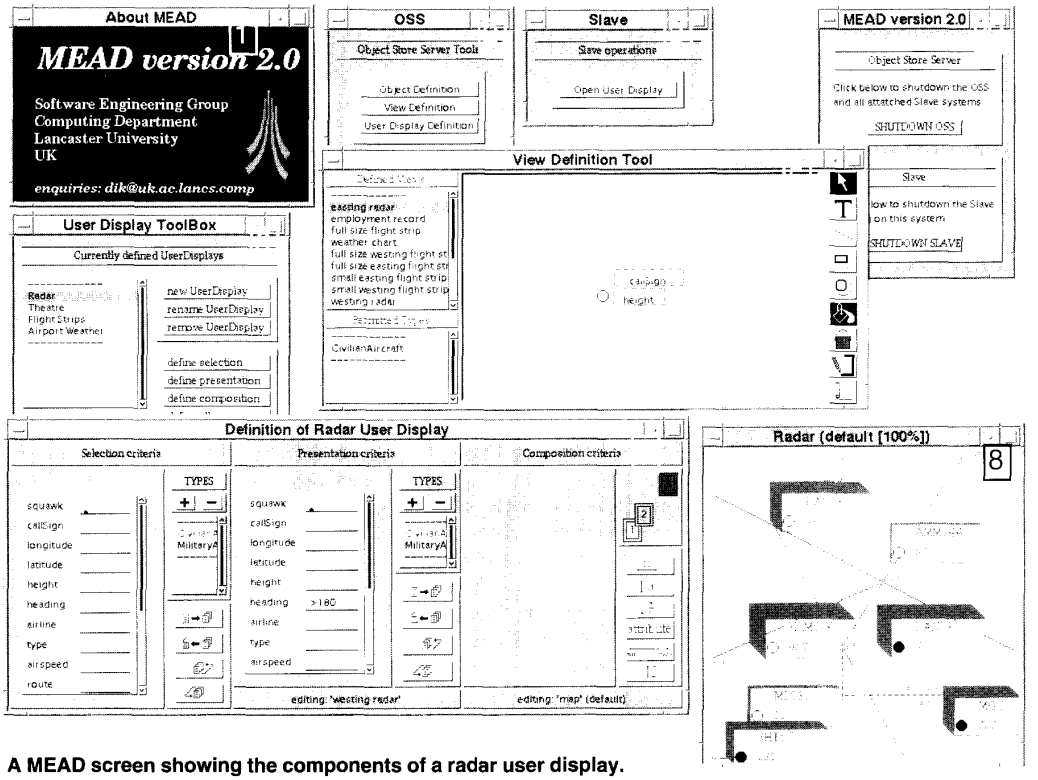
This mechanism ensures that agents can maintain consistency between their UDs and the shared information without being overloaded with irrelevant update notifications. The filtering of updates by the update handler minimizes communication overhead, and therefore the cost of notification; it also allows applications to remain unaware of the agents that manage representations of their data.

**Supporting multiple views.** Shared information objects can be represented in

## The MEAD multiuser-interface prototyping environment

MEAD is a prototyping environment that allows the construction and refinement of cooperative displays. (Version 2.0 is developed in Objectworks\Smalltalk r4.1 for Sun workstations.) It makes visible the model of a user display (UD), discussed in the main text, while hiding the complexity of the update handler, master/surrogate, and local caching mechanisms. The figure, which illustrates the definition and realization of a radar UD, shows the tools MEAD provides.The numbers for the MEAD components listed below correspond with those superimposed on the windows in the figure.

(1) *MEAD information window.*
(2) *Object store server tools.* From this toolbox, the user interface developer can open tools on the object store (not illustrated), the view definitions (window 5), and the master UD agents (window 6). These definitions are all held in a central component called the object store server (OSS).
(3) *Slave toolbox.* To open a window such as window 8, which presents a UD, a slave that registers itself with the OSS described above must be started. The slave toolbox is then accessible, allowing UDs to be opened on the local workstation.
(4) *MEAD launcher.* This panel allows OSS and slave modules to be started and shut down. Typically, one machine will run the OSS in any single session, with slaves being started on a number of machines (which can include the machine running the OSS).
(5) *View definition tool.* This tool supports the definition of different entity representations, called views. Using a set of primitives, the user interface developer constructs the representations, indicating how entity attribute values are to be positioned, how they are to be presented, and how the user can interact with them to change the state of a represented entity.
(6) *User display browser.* This tool allows UDs to be created, renamed, and removed. In addition, definition tools can be opened on the selection, presentation, and composition criteria components of the selected UD (window 7).
(7) *User display definition tools.* These tools capture the definition of the selection, presentation, and composition criteria for a UD. Each set of criteria is created and modified using a separate form. The selection and presentation criteria are specified by editing a condition template. A single entity must pass all the conditions on a template to pass the guard, which the template defines. Composition axes are defined to specify the layout of the representations in the UD. This composition criteria definition tool allows several different arrangements to be defined so that the user can change layouts; in addition, it supports the association of different backdrops, depth effects, and so forth. The layout shown here is a three-dimensional arrangement, with aircraft longitude and latitude mapped on to *x-y* position and height used to calculate the depth.
(8) *User display window.* This window presents the UD defined in window 7. Aircraft entities are taken from an information store created from actual flight-plan data. The attribute values shown in each view can be edited to update the underlying entities, with all changes propagated to all other representations on all slaves. In addition, users can change the view representation being used for each entity and also the layout they wish to use (if alternatives have been defined). Any changes made to the view or UD definitions are immediately propagated to all open windows that present affected UDs.




**A MEAD screen showing the components of a radar user display.**

different UDs by different views. As updates occur, views must also be updated to remain consistent with the entities they represent. To maintain consistency, views are linked to the objects they represent for each UD (see Figure 5). These links allow update information to flow between views and shared information objects when the user modifies a view and when objects are changed through other end users' interactions or by external updates.

As Figure 5 shows, each object in a UD's selection may have links between itself and more than one view. To minimize communication between the OSS and remote machines, the agent architecture adopts a mechanism whereby shared information objects are cloned and held in a local cache (see Figure 6). Updates to information objects are not routed through the UD agents but are sent to the caches by the update handler, which maintains a table of locations of object clones. The links from the update handler to the caches are bidirectional so that the update handler can receive updates to clones resulting from user interaction with object views. The update handler can then notify the application of the update, selectively multicast updates to caches that maintain a clone of the affected object (and update all the object's views), and inform the relevant master agents in case the update requires changes in the states of their UDs.

The UD model and agent architecture described here form the basis of a multiuser-interface prototyping environment called MEAD (Multiple Electronic Active Displays) described in the sidebar. The architecture is flexible enough to allow view and UD definitions to be added, removed, and modified in the OSS while other machines are connected — without requiring suspension of user activities. The architecture directly supports the rapid prototyping facilities provided by MEAD through the set of simple, robust mechanisms described in this article.

Because we have limited knowledge about the nature of group work and still lack proven interface principles, rapid prototyping is essential for the development of effective cooperative systems. Prototyping of this sort demands a robust set of mechanisms to support user-interface construction and execution.

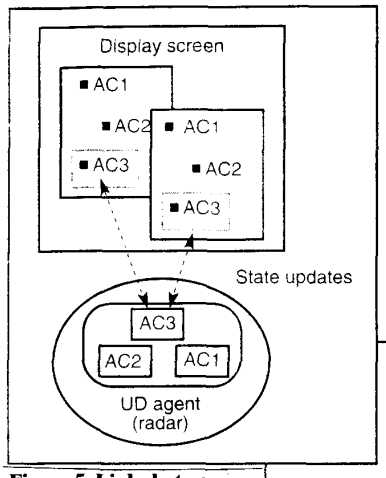



**Figure 5. Links between views and shared information objects.**

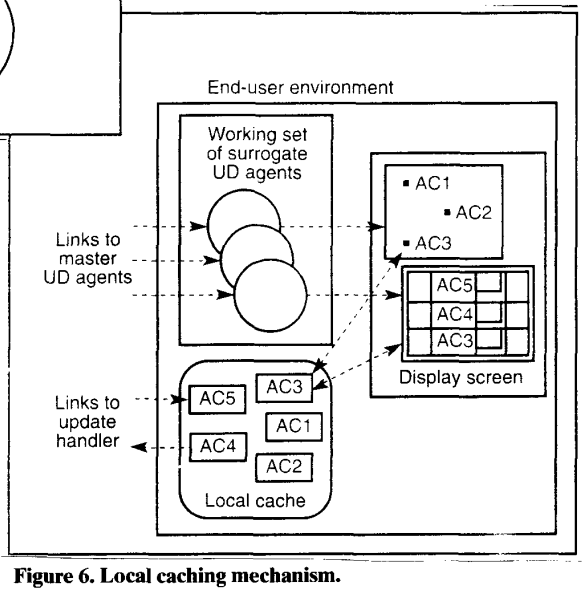



**Figure 6. Local caching mechanism.**

Our architecture represents an initial exploration into what a suitable set of mechanisms for the construction of multiuser interfaces might be. The needs of multiuser-interface support require consideration of how interface architectures work in tandem with the supporting distributed infrastructure. One consequence of this interdependency is that the supporting mechanisms for multiuser interfaces need to be designed to execute within a distributed environment. This requires a careful reassessment of the structure of user- interface software.

MEAD is one of several research prototypes being used to explore the structure of future multiuser-interface systems. Multiuser interfaces have an important role to play in future interactive applications, and the agreement of common architectural principles is essential. Considerable work is still required in establishing such principles and in migrating them to standard applications. ■

## References


1. R. Bentley et al., "Ethnographically Informed Systems Design for Air Traffic Control," *Proc. Conf. Computer-Supported Cooperative Work*, ACM Press, New York, 1992, pp. 123-130.
2. J.C. Lauwers and K.A. Lantz, "Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared Window Systems," *Proc. CHI 90, Conf. Human Factors in Computing Systems*, ACM Press, New York, 1990, pp. 303-311.
3. J.F. Patterson et al., "Rendezvous: An Architecture for Synchronous Multiuser Applications," *Proc. Conf. Computer-Supported Cooperative Work*, ACM Press, New York, 1990, pp. 317-328.
4. J.F. Patterson, "Comparing the Programming Demands of Single-User and Multiuser Applications," *Proc. Conf. User Interface Software Technology*, ACM Press, New York, 1991, pp. 87-94.
5. J. Tang, "Findings from Observational Studies of Collaborative Work," *Int'l J. Man-Machine Studies*, Vol. 34, No. 2, Apr. 1991, pp. 143-160.
6. D. Garfinkel et al., "The Shared X Multiuser Interface User's Guide, Version 2.0," Research Report STL-TM-89-07, Hewlett-Packard Labs, Palo Alto, Calif., 1989.
7. T. Crowley et al., "MMConf: An Infrastructure for Building Shared Multimedia Applications," *Proc. Conf. Computer-Supported Cooperative Work*, ACM Press, New York, 1990, pp. 329-342.
8. C.A. Ellis and S.J. Gibbs, "Concurrency Control in Groupware Systems," *Proc. 1989 ACM SIGMOD Int'l Conf. Management of Data*, ACM Press, New York, 1989, pp. 399-407.

# CALL FOR PAPERS

RE '95

Second IEEE International Symposium
on
## Requirements Engineering

March 27-29, 1995 • York, England

Papers on all aspects of requirements engineering are welcome. However, all submitted papers must be classified according to the problems they are addressing and the contributions they are making toward solving them. The official classification scheme for the symposium can be obtained by requesting it from the program chair or by anonymous ftp from host *research.att.com (/dist/re95.cfp)*. Authors must submit six copies of each full paper (no email or FAX submissions) to the program chair. Papers should not exceed 6,000 words in length, and should be accompanied by full contact information including name, address, email address, telephone number, and FAX number.

**IMPORTANT DATES**
**August 1:** title, abstract, & classifications requested;
**September 1:** full papers due;
**November 1:** notification of acceptance;
**December 15:** camera-ready copy due.

**FOR A COMPLETE CALL FOR PAPERS:**
A complete call for papers, including information for authors, doctoral students and potential distributors, is available by anonymous ftp from *minster.york.ac.uk* in the directory *pub/re95*, filename *re95text*. A World Wide Web page is also available with the URL *http://www.cs.uoregon.edu/~fickas/re95.html*.

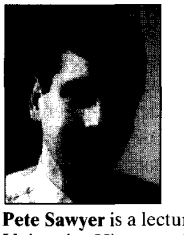
**FOR MORE INFORMATION, CONTACT:**
**General Chair:**
Michael Harrison, Department of Computer Science,
University of York, York YO1 5DD UK
(44) 904 432721; FAX (44) 904 432767
re95@minster.york.ac.uk

**Program Chair:**
Pamela Zave, AT&T Bell Laboratories, Room 2B-413,
Murray Hill, NJ 07974 USA
(1) 908 582 3080; FAX (1) 908 582 7550
pamela@research.att.com

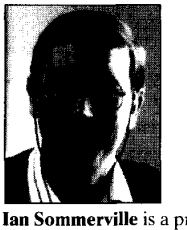
Sponsored by
IEEE Computer Society TC on Software Engineering
In cooperation with
ACM SIGSOFT (pending), IEE,
IFIP Working Group 2.9 (Software Requirements Engineering)

**Richard Bentley** is a research fellow with Rank Xerox Research Center, Cambridge, UK. His research interests include multiuser-interface development, computer-supported cooperative-work architectures, computer-augmented environments, and evaluation of cooperative systems. He has a BSc in computer science from Lancaster University, UK, and recently submitted his doctoral thesis. He is a member of ACM.

**Tom Rodden** is a senior lecturer in the Computing Department at Lancaster University, UK, and project manager of an ESPRIT basic research project on computer-supported cooperative work. His research interests include computer-supported cooperative work, software design, requirements capture, and supporting technology for cooperative applications. He has a BSc in computer science and microprocessor systems from Strathclyde University, UK, and a PhD in computer science from Lancaster University. He is a member of ACM, IEEE, and the IEEE Computer Society.



**Pete Sawyer** is a lecturer in the Computing Department at Lancaster University. His research interests include user interface design tools and environments, object-oriented databases, and requirements engineering. He has a BSc and a PhD in computer science, both from Lancaster University.



**Ian Sommerville** is a professor in the Computing Department at Lancaster University. His research interests include requirements engineering, cooperative system design, and safety-critical software systems. He has a BSc from Strathclyde University and an MSc and a PhD from St. Andrews University. He is a chartered engineer, a fellow of the IEE, and a member of the IEEE, the ACM, the British Computer Society, and the IEEE Computer Society.

The authors can be contacted at Lancaster University, Computing Department, Lancaster LA1 4YR, UK; e-mail {dik, tam, sawyer, is}@comp.lancs.ac.uk. Although the work described was carried out while Bentley was at Lancaster University, he is now with Rank Xerox Research Center, 61 Regent St., Cambridge, CB2 1AB. He can be reached at the e-mail address shown above or at bentley@europarc.xerox.com.