MIT/LCS/TM-264

SOFTWARE FOR INTERACTIVE ON-LINE CONFERENCES

Sunil K. Sarin

Irene Greif

July 1984

Software for Interactive On-Line Conferences

Sunil K. Sarin
Irene Greif

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge, Massachussets

ABSTRACT:

A layered architecture for the implementation of *real-time conferences* is presented. In a real-time conference a group of users, each at his or her own workstation, share identical views of on-line application information. The users cooperate in a problem solving task by interactively modifying or editing the shared view or the underlying information, and can use a voice communication channel for discussion and negotiation.

The lower layer in this architecture, named *Ensemble*, supports the sharing of arbitrary application-defined *objects* among the participants of a conference, and the manipulation of these objects via one or more application-defined groups of commands called *activities*. Ensemble provides generic facilities for sharing objects and activities, and for dynamically adding and removing participants in a conference; these can be used in constructing real-time conferencing systems for many different applications. An example is presented of how the Ensemble functions can be used to implement a shared bitmap with independent participant cursors.

The relation between this layered architecture and the ISO Open Systems Interconnection reference model is discussed. In particular, it is argued that Ensemble represents a plausible first step toward a Session-layer protocol for "multi-endpoint connections", a neglected area of communication protocol development.

Keywords: desk-to-desk conferencing, distributed systems

To appear in *Proceedings ACM-SIGOA Conference on Office Information Systems*, Toronto, Canada, June 25-27, 1984.

This research was sponsored by the Defense Advanced Research Projects Agency, and was monitored by the Office of Naval Research under contract number N00014-83-K-0125.

Table of Contents

1. Introduction

Interactions among people fall into two categories: *simultaneous* interactions such as face-to-face meetings or telephone conversations, and *asynchronous* interactions such as correspondence via the postal service. Each is appropriate for different situations, and neither is likely to completely replace the other. Asynchronous interaction can be extremely useful and efficient in that each communicator can act at a time and rate of his own choosing. However, it is often the case that after a sequence of asynchronous exchanges of information, proposals, and counterproposals it is necessary to negotiate in real time in order to resolve outstanding issues. Rapid group decision-making in a crisis situation also requires simultaneous interaction.

In the area of computer support for group interactions, asynchronous communication can be accomplished in several ways: electronic mail [11], computer conferencing [13], or forms management [26]. With a few exceptions, simultaneous interaction in such "meetings at a distance" has been largely neglected. Our research is concerned with implementing *real-time conferences*, in which groups of users at interconnected workstations can collectively view and manipulate on-line information. A real-time conference is not intended to simulate or replace every kind of face-to-face meeting. For meetings where the social and political content is dominant, the computer cannot help much; if face-to-face interaction is impossible or inconvenient, some form of video teleconferencing might be appropriate. However, for meetings where problem-solving using on-line information is the primary objective, a real-time conference allows a group of problem solvers to exploit the full power of the computer to retrieve, edit, process, and store information, without leaving their desks to convene a face-to-face meeting.

A real-time conference consists of an *informational* component and a *discussion* component. For the purpose of this research, we assume that the two components are implemented separately and independently, and focus our attention on the informational component. A voice communication channel is typically sufficient to allow spirited discussion and negotiation [1], and we will assume that the participants in a given real-time conference will set up the discussion component by some unspecified external means, e.g., a telephone conference call. We note, however, that the ideas presented in this paper are equally applicable to the voice channel if the hardware is available for building an integrated conferencing system that includes voice as well as data.

An example real-time conferencing system is a prototype named RTCAL (for *Real-Time CALendar*) that we implemented at the MIT Laboratory for Computer Science. RTCAL allows a group of users in a conference to exchange information from their personal calendar databases [10] in order to schedule a future meeting. The participants in a conference also typically set up a telephone connection to allow them to discuss their schedules and the planned meeting.

```
+-----------------------------------------------------------------------------+
|RTCAL 3.2 ctrl-↑ for control commands 12-4-82 11:52:07 Load=8.7         SARIN|
+-----------------------------------------------------------------------------+
|scheduling meeting "thesis"    uncommitted    (2hrs, 12-25-82 to 12-31-82)|
| With SARIN                LICKLIDER         GREIF              HAMMER       |
|      IN-Session           IN-Session        IN-Session         Absent       |
| session Running           chairperson: SARIN      controller: SARIN         |
+-----------------------------------------------------------------------------+
|LICKLIDER joined session - all replies received                             |
+--------------------------------+--------------------------------------------+
|Monday 27 December 1982         |Private calendar: 27 December 1982          |
|Merge of SARIN LICKLIDER GREIF  | Joe's birthday                             |
| 9:00   XXX                     | 9:00                                       |
| 9:30   XXX                     | 9:30                                       |
|10:00                           |10:00                                       |
|10:30                           |10:30                                       |
|11:00                           |11:00                                       |
|11:30                           |11:30                                       |
|12:00                           |12:00                                       |
|12:30   XXX                     |12:30    lunch                              |
|13:00                           |13:00                                       |
|13:30                           |13:30                                       |
|14:00   XXX                     |14:00    Arpa meeting                       |
|14:30   XXX                     |14:30     xx                                |
|15:00   XXX                     |15:00                                       |
+--------------------------------+--------------------------------------------+
| COMMAND> propose 10:30_                                                     |
+-----------------------------------------------------------------------------+
```

Figure 1-1: Example RTCAL Screen

The format of a participant's display screen in RTCAL is shown in Figure 1-1. The top of the screen presents "status" information such as the meeting subject and the names of the participants present in this conference. A *shared* "window" on the left shows blocks of free and busy times, obtained by "merging" information from the participants' calendars; all participants see this information. A *private* window, on the right, shows detailed information about the participant's own appointments, not visible to the other participants. The shared space can be "scrolled" to show a different date and time range (which causes participants' private windows to scroll in unison), or the set of participants whose schedules are merged can be changed. Specific times for the planned meeting can be *proposed*, and participants' *votes* are collected and tabulated. Multiple alternative proposals can be generated and reviewed, and a final meeting time selected by *committing* any one proposal;

commitment can be "undone" to recover from mistakes. These *calendar commands* are "echoed" on all participants' screens as they are typed.

Only one person at a time, referred to as the *controller*, can enter calendar commands. The conference *chairperson* (in RTCAL, the person who initiated the conference) may designate a new controller (including himself) at any time. This is done using a separate set of commands referred to as *control commands*. Control commands also allow participants to "request" control. Participants may leave the conference temporarily and receive an up-to-date display when they return. The chairperson may terminate the conference whenever he chooses; a meeting time may or may not have been committed.

RTCAL was our initial experiment in real-time conferencing; it supports a particular kind of problem-solving in a specific application. The objective of our research is to extend these ideas to other applications (such as computer-aided design, joint authorship of documents, financial planning using "spreadsheets", or on-line tutorials and instruction), as well as to allow more varied "styles" of conferencing (such as concurrent commands from participants, and more flexible interaction than with a single fixed "chairperson"). The design of a joint document editing system illustrating these goals is described in [23].

This paper presents an architecture for real-time conferences and a common set of functions that can be used to support conferencing in many applications, such as the above, using many different conference styles. In this model, the primary purpose of a conference, for whatever application, is the sharing of one or more *objects* (e.g., documents, circuits, agendas, proposals, or screen images) among a group of participants, allowing manipulation of these objects in a controlled way via one or more groups of commands called *activities*. The structure and meaning of the objects and activities in a conference will be specific to the given application, but the dynamics of sharing the objects and setting up the activities is performed by a common software utility that we have named *Ensemble*.

In Section 2 we describe a "layered" architecture, not unlike that of the ISO Open Systems Interconnection (OSI) reference model [14], in which the Ensemble "layer" provides conference control functions to the application "layer". The functions provided are conference initiation and termination, participants joining and leaving a conference, and the sharing of application-defined objects and activities. In Section 3 we illustrate how these functions can be used to construct particular kinds of conferences, by presenting the design of a "shared bitmap" facility that allows participants to view an identical bitmapped image on their workstation screens and move independent cursors over this image using their pointing devices. We conclude in Section 4 by noting that the Ensemble functions correspond very closely with the stated function of the OSI

"Session" layer, and that Ensemble is therefore a plausible starting point for developing a "multi-endpoint" Session layer protocol.

2. Ensemble Conference Architecture

This section introduces the conference control functions provided by *Ensemble*, in the context of a layered distributed system model.

2.1. System Model

We assume that the system consists of a collection of *nodes* that do not share memory but communicate only via *messages* sent over a communications network or internetwork; we assume that messages between any two nodes are reliably received in the order sent. We will assume in the following discussion that each node is running on a separate processor, with the understanding that multiple nodes might in fact be invisibly multiplexed on a single physical machine. (This model is thus in many ways similar to other contemporary models of distributed computing, such as "guardians" [18].)

We assume that the functions within a given node are arranged into *layers*. While discussing Ensemble, we shall assume that there are just two layers:

1. The *Ensemble* layer, which provides the conference control facilities we describe here.
2. The *application* layer, which uses the Ensemble facilities in order to implement real-time conferences for one or more applications.

In actual practice, the Ensemble and application layers may be further subdivided into layers, as in the ISO Open Systems Interconnection reference model [14]. For example, the application "layer" may have a Presentation layer performing window management and command parsing, while the Ensemble "layer" will have a Transport and lower layers that provide the underlying communication services.

The Ensemble layer (perhaps via a lower "Physical" layer) is the only layer that interfaces directly with attached hardware devices, namely disks, networks, and workstation display and input devices. Services not directly related to conference communication, such as shared file storage and "window management", are assumed to be implemented by the "application" layer, with Ensemble providing only a fairly low-level interface to the associated devices.[1]

As shown in Figure 2-1, interaction between the Ensemble and application layers within a node takes place in the form of two kinds of calls that each makes of the other:

- *Downcalls* from the application layer to the Ensemble layer, typically "requesting"

[1] We make this assumption only so as to concentrate on the specification of Ensemble's conferencing functions. In actual practice it would probably be desirable to standardize these other services across all applications (using any of several well-known techniques) and place them in a lower layer than the application.

```
          +----------------+
          |                |
          |  APPLICATION   |
          |                |
          +----------------+
               |         ^
               |         |
      Downcalls|         |Upcalls
               |         |
               v         |
          +----------------+
          |                |
          |    ENSEMBLE    |
          |                |
          +----------------+
```

Figure 2-1: Layer Interface within a Node

conference-related or other i/o device services.

- *Upcalls* from the Ensemble layer to the application layer, typically reporting "events" related to some previous request for service.

(This terminology is due to Clark [2].) Each call, up or down, is a kind of "message", that causes a *handler* in the called layer to be invoked. Unlike message-passing systems, however, we assume that the layers interact as cooperating "coroutines": downcalls and upcalls are *synchronous*, in that the layer making the call does not continue to run in parallel with the other layer's handler, but is blocked until the handler returns.

At any given time, a node is processing one *event*, which may be one of the following:

- An event from a hardware device, such as a disk, network, user input device, or a timer.
- A special *background* event that is defined to occur when there are no other events waiting to be processed.
- A special *initialize* event, perhaps triggered by hitting the machine's "boot" button, which is used to start the node.

Each event causes a handler in the Ensemble layer to be invoked, which may make upcalls to the application layer, which may in turn make downcalls to the Ensemble layer. (To avoid mutual recursion, we require that Ensemble's downcall handlers not make upcalls to the application layer.) The node must dispose of one event, i.e., Ensemble's event handler must return, before the next event can be processed.[2] This structure implies that the lower layer, Ensemble, is "in charge", and application layer functions are invoked as required by the Ensemble layer. This "upside down" view runs counter to the traditional view of the "application" being in control and invoking "system" functions, but does in fact correspond better with the reality of how asynchronous events are processed by a system. This inverted approach is beginning to gain wide acceptance. For example, in the VisiOn^{TM} [17] system the VisiHost^{TM} is in charge of all activity. The VisiHost^{TM} invokes specific application functions at the appropriate times, and blocks until the application functions, corresponding to our "handlers", complete. The advantages of such an approach are discussed in more detail in [2].

We find it convenient to distinguish two kinds of functions that a given node might perform:

- *Front-end* (or *FE*) functions: interfacing with a user's display and input devices and managing the set of objects that he is working with at any given time.
- *Server* functions: shared file storage, mail facilities, name lookup, and conference control.

This distinction applies to both the Ensemble layer and the application layer. We shall call the respective sets of functions in the two layers *Ensemble-server*, *Ensemble-FE*, *application-server*, and *application-FE*.

2.2. Conferences, Objects, and Activities

Within the context defined above, a *conference* is an agreement among a collection of front-end nodes, representing the *participants* of the conference, and a server node, that controls the conference, to share a specified collection of *objects* and to allow manipulation of some or all of these objects via specified *activities*.

We assume a *centralized* architecture for any given conference, in which the front-ends of the participants communicate only with the server controlling the conference; this is illustrated in Figure 2-2.[3] Note that we do not assume a centralized architecture for the system as a whole; different server nodes may control different conferences.

Objects in a conference are controlled by the conference's server node. The most common type of

[2]It is assumed that hardware device events, which usually must be caught within a very short time interval, are not lost, by having them queued by code running at the machine's "interrupt level". It is still desirable for handlers to return quickly in order for the node to remain responsive to incoming events; a handler that needs to perform lengthy computation or i/o can return quickly by setting up some state information that will cause the required processing to be performed by some later handler, e.g., in "background" or in response to a timeout.

[3]*Decentralized* architectures, in which front-end nodes can communicate directly, are discussed in [23].

```
   SERVER                          Front-End              Front-End

 +--------+                       +--------+              +--------+
 | Applic.|                       | Applic.|              | Applic.|
 | Server |                       |   FE   |   . . . .    |   FE   |
 +--------+                       +--------+              +--------+
     ↑                                ↑                       ↑
     |                                |                       |
     v                                v                       v
 +--------+                       +--------+              +--------+
 |Ensemble|                       |Ensemble|   . . . .    |Ensemble|
 | Server |                       |   FE   |              |   FE   |
 +--------+                       +--------+              +--------+
    ↑    ↑                            ↑                       ↑
    |....|                            |                       |
    |    v                            v                       |
    v    ------------------------------                       v
    -----------------------------------------------------------
               (Communication Network)
```

Figure 2-2: Conference Architecture

"object" that might be shared in a conference is an *image* that can be directly displayed on the participants' workstation screens; examples of images are text images, graphs, line-drawings, raster images, and so on. A shared image is thus the means by which the illusion of a common "blackboard" can be presented. (Unlike a blackboard, however, an image will often present a view of (part of) a more abstract application object, e.g., a document or a circuit.) This notion of "image" is purely a convention involving the application-server and application-FEs. Ensemble is not aware of the semantics of an object, and many other kinds of objects can be shared in a conference, such as:

- High-level application objects, e.g., documents or circuit designs.
- Objects used to facilitate participant interaction in the conference, e.g., agendas, minutes, proposals and votes, or a queue of "requests" for permission to enter commands.
- Objects exchanged by the server and front-ends to determine run-time parameters such as the maximum bandwidth and size of objects that can be supported.

Objects shared in a conference are updated only by the application-server; the application-FEs' copies are "read-only" in that an application-FE cannot modify its copy except in response to updates received from the application-server. (Again, we assume that the server and front-end nodes are cooperating properly in realizing a real-time conference, so we do not concern ourselves with nodes behaving incorrectly or maliciously.) Such updates to objects are generated by the application-server according to its own criteria. While it is quite possible for updates to be spontaneously generated by the application-server (e.g., when running a simulation, or responding to events outside the conference), more typically the purpose of a conference will be to let the participants themselves initiate updates (e.g., to edit a document or circuit) via some set of defined commands. Commands initiated by a participant are forwarded by the participant's application-FE to the application-server (via facilities that Ensemble provides) for processing; the application processes these commands, updating one or more shared objects or generating error messages as appropriate.

Commands in a conference are grouped into *activities* such that permission to enter the commands in a given activity can be independently given out to a selected subset of the conference participants. This allows for different logical groupings of commands, e.g., "application" commands versus "conference control" commands, as in the example system RTCAL. The application-server can give different participants permission to enter different types of commands, and can dynamically give and revoke such permission during the course of a conference.

Each object in a conference is "shared" among some subset of the conference participants in the sense that the given participants' application-FEs are provided with copies of the given object and receive updates from the server as they occur. A participant's application-FE can present him with a view of some or all of these objects, under control of the participant. Most objects in a conference will be "universally" shared in that they are displayed to, or made available for display to, all of the participants in the conference. Other objects might be available to only a single participant, e.g., feedback messages informing the participant that he does not have permission to enter certain commands; such feedback would not be meaningful to the other participants. In most conferences, we expect that each object will fall into one of these two classes, i.e., shared with all participants or available to just a single one. In some conferences, however, it may be necessary to support "subgroup" discussions, and Ensemble provides the flexibility for defining objects that are shared with any subset of a conference's participants.

The application-server indicates its desire to "share" an object with a given participant's application-FE by issuing a "Give-Object" downcall to the Ensemble-server; an encoded description of the object's current value is passed to the participant's application-FE in an upcall from the Ensemble-FE. Subsequent updates to the given object, generated by the application-server, are transmitted by Ensemble to the application-FE only if the latter indicates willingness to receive and process them; the application-FE does this using an Accept-Object downcall. The application-FE may instead decline to receive updates, using the Decline-Object downcall, for whatever reason (e.g., insufficient local storage or processing power); updates to the object will not transmitted by Ensemble to this particular application-FE.

Every conference has a distinguished object, called its *description*, that is always shared among all of its participants. The purpose of the conference description is threefold:

1. To allow the application-server to properly set up its state information when it receives an Initialize-Conference upcall. The conference description might include, for example, the name of a file containing the document to be edited in the conference.
2. To instruct the participants' application-FEs to supply information that will be used by the application-server in setting up some of the objects in the conference. In the example that follows, in the next section, the size of a bitmapped image is to be selected based on available screen space information received from the application-FEs; the conference description should specify what information of this kind is needed.
3. To help a user decide whether to join the conference or to leave, e.g., by providing brief textual statements about the "purpose" of the conference and the estimated time and duration of the conference. (Some of the information used by the application-server and application-FE, above, might also be meaningfully presented to the user.)

It is of course up to the application to supply a conference description that accomplishes the above goals; Ensemble is not aware of the structure and semantics of the conference description, or of any other object.

A participant is brought into a conference by "sharing" the conference description with his application-FE:

- The application-server issues an Add-Participant downcall, which gives the participant's application-FE a copy of the conference description.
- The participant's application-FE issues a Join-Conference downcall, "accepting" the conference description. (The participant may instead choose not to join the conference; the Leave-Conference downcall "declines" the conference object.)

When the application-server receives an upcall indicating that the participant has joined, it then determines which shared objects the participant should receive (using its own access control criteria), and proceeds to "give" these objects to the participant as described above. Note that the protocol used for sharing the conference description is the same as that for sharing any other object in the conference; the Add-Participant, Join-Conference, and Leave-Conference downcalls are simply specialized versions of Give-Object, Accept-Object, and Decline-Object, respectively.

Ensemble provides the basic facilities for setting up a conference among a group of participants (by sharing the conference "description", as above) and for sharing objects and activities within a conference. There are several important services that Ensemble does not provide, such as permanent file storage, access control, naming and authentication, and user interface management. These functions are assumed to be implemented "above" Ensemble, by the "application" layer, in whatever manner the system or application designer deems appropriate.

The Ensemble architecture also does not directly address performance issues, e.g., the bandwidth and processing power required to achieve a desired response time for a given application and given number of participants; these must be considered by the application designer in the context of his particular system. Clearly, a richer interface and faster response time will be attainable on a high-bandwidth local area network than over a long-haul network or internet, and performance will in both cases degrade as the number of participants increases. (For example, the response time of the shared bitmap facility of the next section may not be adequate with a slow network.) If performance information (about communication bandwidth and delay, and front-end processing capabilities) is available at run-time, the application designer can incorporate their values into shared "objects" and can program the application-server to tune the interface and underlying update and activity protocol to the run-time parameters.

A complete list of the "downcalls" and "upcalls" for setting up conferences, shared objects, and activities, is presented in the Appendix. Ensemble deals with objects, updates thereof, and activities, and commands therein, as uninterpreted "blocks" of data that are transmitted between application-server and application-FEs as necessary; the syntax and semantics are imposed by the application (server and front-ends). Ensemble also does not specify how the application (server or front-end) should respond to a particular upcall, e.g., what processing it should perform and what downcalls it should make; it is up to the application designer to program handlers in the application server and front-ends that will accomplish the functionality and interface desired in the given real-time conference. Techniques available to the application designer are discussed in [23].

3. Example: Sharing a Bitmap

To illustrate how the functions provided by Ensemble can be used to implement real-time conferences, we present a facility that allows participants in a conference to share a virtual bitmapped screen. The shared bitmap facility also allows for keyboard and pointing device input from some or all participants.

```
      SERVER                  Front-End(s)

    +--------+
    | Applic.|
    | Server |
    +--------+
        ^
        |
        v
    +--------+               +--------+
    | Bitmap |               | Bitmap |
    | Server |               |   FE   |
    +--------+               +--------+
        ^                        ^
        |                        |
        v                        v
    +--------+               +--------+
    |Ensemble|               |Ensemble|
    | Server |               |   FE   |
    +--------+               +--------+
        ^                        ^
        |                        |
        --------------------------
```

Figure 3-1: Architecture of Shared Bitmap System

The shared bitmap facility can be used in conferences by many different applications. It is thus an intermediate-level abstraction that lies between the Ensemble layer and the high-level application code. (The shared bitmap is an advanced form of *virtual terminal* [5, 16]; in terms of the Open Systems Interconnection reference model, it corresponds to the Presentation layer.) This layering is shown in Figure 3-1.

The application-server manages some set of application objects, e.g., documents, circuits, spreadsheets, etc., and presents some view of these objects on the shared bitmap. (A simple application-server could also be written to provide an "electronic blackboard" on which participants can draw figures and enter text.) These application objects are internal to the application-server and are not "shared objects" from Ensemble's point of view; they are visible to the participants only indirectly via whatever view the application-server presents in the shared bitmap, and may be manipulated by participants only by entering commands that are parsed and executed by the application-server according to its own syntax and semantics. The interface between the bitmap-server and the application-server is implemented in the same way as the Ensemble interface, in terms of downcalls and upcalls that we will not list in detail here. Ensemble is not aware of this separation of function between the bitmap and application "layers"; the combination of these two layers appears to Ensemble to be a single "application" layer.

Note also that Figure 3-1 does not show an application-FE layer at the participants' front-end nodes. Instead, the application program runs in the server node only, with the participants' user interface being taken care of completely by the bitmap-FE layer (as described below). This allows easy development of applications, which can be written to manipulate the shared bitmap in the same way that they would manipulate a user's directly-connected screen bitmap; there is no need to write a "distributed" application program.[4] It should also be fairly straightforward to convert an existing application program, that interfaces directly with a user's bitmapped screen, to use the shared bitmap instead. This architecture also assumes minimal support from the participants' workstations, only requiring that they be able to run the bitmap-FE software.

3.1. Object Specifications

The shared bitmap facility provides the following objects for sharing in a conference:

- The conference *description* (which as we have described every conference must have), which consists of the following components:
  - A specification of what application is to be run; this is used to initialize the application-server for this conference.
  - A brief text statement of the "purpose" of the conference.
  - The current size (height and width) of the shared bitmap; this is "undefined" if the bitmap has not yet been initialized.

- The *shared bitmap* itself. This is a rectangular array of intensity bits that can be displayed, in whole or part, on a raster screen [20, 8].[5] The shared bitmap is made available to all participants in a conference, i.e., their bitmap-FEs receive copies of the bitmap for display on their screens. The application-server can freely read and update the contents of the bitmap, either bit-by-bit or using

[4]Note that the shared bitmap is just one example of how Ensemble can be used. More sophisticated application programs that are distributed between the server and front-ends are supported equally well by the Ensemble architecture, as described in [23].

[5]The ideas presented in this section can be easily extended to raster images that allow multiple bits per pixel, designating gray-levels or color; additional storage and bandwidth will of course be required to support this.

higher-level constructs such as lines, rectangles, characters, arcs, and so on; all such updates are transmitted to the participants' bitmap-FEs for display.

• One *pointing-cursor* for each participant in the conference. Each pointing cursor is specified by its current *position* (x- and y-coordinates) and a small rectangular bit-pattern called its *shape*. The set of pointing cursors are shared among all participants, to allow each to observe the movements of the others' pointing devices; assigning different shapes allows the different participants' cursors to be distinguished.

• A bit pattern, called *own-cursor-shape*, that every participant will use on his own screen to distinguish his pointing cursor from those of the others.

• One *text-feedback* object per participant, each available only to the given participant. These can be used by the application-server to give participant-specific feedback, and also to allow participants to send text "messages" to each other (as described under the conference activities, below).

• For each participant, an *input-stream* consisting of a sequence of keystrokes and mouse button events from the participant's workstation; these are passed to the application-server for processing.

• For each participant, an *available-screen-space* object that specifies how much space (height and width, in pixels) the participant has available on his screen for displaying the shared bitmap. This will be used, as described below, in choosing a suitable size for the shared bitmap that all participants can accommodate.

3.2. Activity Specifications

The following activities are defined in a conference that uses the shared bitmap facility:

• One *stream* activity per participant, which allows the participant's bitmap-FE to add keystrokes and mouse button events to the participant's input-stream object (described above). The application-server receives an upcall whenever new stream input arrives, and can remove and process input from the participants' input streams according to its own command syntax and semantics. The application-server might implement not only application commands (e.g., to edit a document), but also "conference control" commands to selectively disable, and re-enable, individual participants' stream activities. Thus, the application-server may accept input from only one participant at a time if it wishes, which is useful for sharing existing single-user application programs.

• One *pointing* activity per participant, for tracking movements of the participant's pointing device. The purpose of these activities is to allow participants to "point" at information that they may be referring to in their voice channel discussion, without involving the application program or disturbing the contents of the shared bitmap. Each participant's bitmap-FE sends periodic mouse position reports (relative to the "virtual" coordinate system of the bitmap, not the participant's physical screen coordinates), at some predetermined frequency, to the bitmap-server which updates the participant's pointing-cursor position and forwards the updates to all participants' bitmap-FEs.[6]

• A *message* activity for each participant that allows the participant to send brief text messages, to the entire group or "privately" to individual participants.

• An activity for each participant that allows the participant's bitmap-FE to set or modify the height and width in his "available screen space" object.

3.3. Participant Interface

```
+--------------------------------+
|                                |
|    1                           |
|                                |
|          Shared                |
|          Bitmap                |
|                        X       |
|                                |
|             4                  |
+--------------------------------+
|         Text Feedback          |
+--------------------------------+
|                                |
|           Private              |
|                                |
+--------------------------------+
```

Figure 3-2: Participant's Screen when Sharing a Bitmap
View of participant number '2'.

A participant can instruct his bitmap-FE to display the shared bitmap and pointing cursors anywhere on his workstation screen (with the bitmap-FE "clipping" the bitmap if the display region is not large enough), and may use parts of his screen for private information. A typical screen configuration is shown in Figure 3-2. The bitmap-FE superimposes the participants' pointing cursor "shapes" at their respective positions. (This superimposition is done by the bitmap-FE software, but it could use multiple hardware cursors if the workstation has such a facility.) In the figure, we have used numbers

[6] In an actual implementation, it is possible to speed this up by having a bitmap-FE directly send mouse position reports to the other bitmap-FEs as well as to the bitmap-server. This optimization is permissible because different participants' position reports update different cursor objects, and can therefore be received and processed in any order.

to indicate the different shapes; this need not be the case in actual practice, e.g., "arrows" with different orientations could be used. In addition, this participant's own cursor is displayed using the defined "own-cursor-shape" (**X** in the figure) rather than the shape (which would be **2** in this example) that the others see.[7] In the figure, the cursor of participant **3** is not visible because that participant is using his mouse for local interaction with his bitmap-FE.

When in a conference, a user's keyboard and mouse input are normally forwarded by his bitmap-FE to the server node in order to move the participant's cursor or enter application commands. A participant can instruct his bitmap-FE, in a number of ways (e.g., via an "escape character" or by moving his mouse outside the region displaying the shared bitmap), to "dissociate" his keyboard and mouse from the conference temporarily. This allows him to perform local interactions such as changing his screen configuration, and retrieving or editing private information. At some point, the participant may instruct his bitmap-FE to "return" to the conference, i.e., resume forwarding keyboard and mouse input.

3.4. Starting a Conference

The following sequence of steps is taken when setting up a conference sharing a bitmap. The objective here is to defer creating the bitmap, and starting the application-server, until information is obtained from "enough" participants as to how much screen space they can allocate for displaying the shared bitmap. The bitmap-server then selects the minimum height and width from the information received and initializes the bitmap accordingly, thus ensuring that all participants will be able to see the entire bitmap on their screens. Note that Ensemble does not require that the steps be followed in this exact order, or even that all of the steps below be taken. The selection of the shared bitmap size, for example, could be done unilaterally by the application-server, ignoring the participants' available screen sizes and letting the participants worry about clipping and scrolling the bitmap if it is too large. Or, information about available sizes could be presented to a participant designated as "chairperson", who chooses between accommodating more participants and being able to present more information in a larger bitmap. All of these options, and others, can be programmed by invoking the Ensemble functions in the appropriate order.

The following description uses "upcalls" and "downcalls" that are listed in the Appendix to this paper.

- First, some user "creates" the conference by instructing his bitmap-FE to issue a Create-

[7]The participant's "own" cursor position is tracked and displayed locally by his bitmap-FE, which does not wait for or display the "echo" from the bitmap-server. An option is provided to display both the local and remotely-tracked mouse positions, which can be useful for debugging or estimating the communication delay.

Conference downcall. The "address" of the server node that will control the conference is specified; we assume that the address is already known or is obtained using some unspecified lookup service. The user enters the information needed in the conference "description": the "purpose" of the conference, and a program name and arguments to be passed to the application-server. The user also supplies a list of addresses of the front-end nodes of the desired conference participants; these may be known in advance (or even reported over the phone), or obtained from some kind of "name server". The user need not enter his own address, the bitmap-server will be supplied with it when it receives the Initialize-Conference upcall, below. (The user might instead, or in addition, supply the addresses of one or more "lookup servers" which will be sent conference descriptions, so that users not explicitly invited to the conference may find out about it by inquiring of a lookup server.)

• The bitmap-server at the given server-address receives an Initialize-Conference upcall carrying the above information. For each front-end address provided, including that of the initiatin user, the bitmap-server includes that front-end in the conference by issuing an Add-Participant downcall; and Add-Activity downcall is also issued to allow each front-end to set its "available screen space" object. The bitmap-server finally issues a Sync-Object downcall indicating that it wishes to be notified when all participants have acknowledged, positively or negatively, and a Set-Timer downcall with some suitably chosen interval to protect against the possibility of one or more participants not responding or taking too long. The Ensemble-server processes these downcalls as a batch when the bitmap-server's handler returns, sending messages carrying the conference description and new activity to the specified participants' front-ends.

• The bitmap-FE of each participant receives a New-Conference and an Activity-Received upcall. It shows the conference description to the user and asks whether or not he wishes to join. If he does, a Join-Conference downcall is issued. The user is also asked to allocate a region on his screen for displaying the shared bitmap, and the size of this is encoded in a Send-Input downcall. If instead the participant does not wish to join, a Leave-Conference downcall is issued.

• When the Ensemble-server receives notification of every participant agreeing to join or leave, it issues a Sync-Complete upcall to the bitmap-server. If instead the previously-specified timeout interval expires, a Timer-Expired upcall is issued. We assume that in this case the bitmap-server does not wish to wait for the remaining participants to join (and therefore issues an Abort-Sync downcall), and proceeds to select a bitmap size as described here. The bitmap-server examines the available screen sizes returned by the participants who agreed to join, and computes the minimum height and width. The shared bitmap is initialized with the selected height and width (and some standard initial contents, e.g., all zeroes). The Add-Object downcall and a set of Give-Object downcalls are used to make the bitmap available to all participants. The bitmap-server also sets up other objects (text-feedback objects and pointing-cursors) and activities (stream, pointing, and message) for each participant, using the Add-Object, Give-Object, and Add-Activity downcalls. As a final step, the bitmap-server issues an Initialize-Conference upcall to the application-server so that the latter may set up its own state information in readiness for conference activity. From the point of view of the users and the application, the conference officially "starts" at this stage, although to Ensemble the conference started the moment it was created.

• The participants' bitmap-FEs receive New-Object and Receive-Activity upcalls for each object (including the shared bitmap) and activity given to them. Each such upcall carries an encoded description of the object or activity. (For the shared bitmap, it is only necessary to send the height and width, because the bits themselves are known to be initially all zeroes and need not be sent.) The bitmap-FE thus displays the objects in the screen regions designated by the participant, and sets up some state information to "associate" the participant's keyboard and mouse with the conference activities. (The devices can be temporarily "dissociated" by the participant, as described earlier.)

Once a participant's workstation is set up as above, keyboard and pointing device input are transmitted by his bitmap-FE to the bitmap-server; this is done using the Send-Input downcall, which results in an Input-Received upcall to the bitmap-server. The bitmap-server receiving input under a pointing activity, i.e., mouse position reports, updates the participant's pointing cursor position and sends the update to all participant's bitmap-FEs (using an Update-Object downcall, as a result of which every participant's bitmap-FE receives an Object-Updated upcall). Stream input received by the bitmap-server is passed up the application-server for processing. The application-server may perform one or more update operations on the shared bitmap, e.g., to echo a character, scroll the contents of a document, or display the effect of some other application command; these are similarly transmitted to all participants' bitmap-FEs, using Update-Object and Object-Updated, for processing and display.

After the bitmap has been initialized and the conference "started", Join-Conference downcalls may be received from additional participants. This may happen with a participant who was originally added to the conference but who did not reply in time (i.e., the bitmap-server timed out and went ahead with the conference), or in the case of a participant who found out about the conference from some "lookup" server. In the latter case, the application-server must decide whether or not to allow the given participant into the conference, based on its own criteria; it may make an immediate decision to accept or reject the participant, or may ask some participant (e.g., the designated "chairperson") to decide. Once the decision is made, manually or automatically, to allow the participant into the conference, Give-Object and Add-Activity downcalls are issued by the bitmap-server to get the participant started, just as described earlier. Because the shared bitmap may have been updated since the conference started, the new participant's bitmap-FE cannot assume that the shared bitmap is clear. The current contents of the shared bitmap must be sent to newly-joining participants, and can be encoded in some efficient way for such transmission, e.g., sending the positions and lengths of contiguous "runs" of one bits.

4. Conclusion

We have proposed a layered software architecture for real-time conferences that allows different applications to use a common set of conference control functions named *Ensemble*. These functions support the sharing of application-defined *objects* among the participants of a conference, and the manipulation of these objects via one or more application-defined groups of commands called *activities*. We used the example of sharing a bitmap and multiple pointing cursors to illustrate how objects and activities can be defined in order to realize a particular kind of conference. Other kinds of useful functions can be implemented in the form of appropriately-defined shared objects and activities, such as meeting "agendas" and "minutes" and "votes", or a queue of "requests" for permission to enter commands via some activity. It is also possible for high-level application objects to be shared between servers and participants' front-end nodes, allowing for more compact transmission of information and local viewing of the application objects. An example system designed along these lines, which supports real-time joint document editing using the concepts presented here, is described in [23]; this uses and extends ideas that have appeared in recent "distributed editing" protocols [9, 24]. It is even conceivable to include voice communication in this framework, by treating the voice stream as a shared "object", with the act of speaking being the "operation" that participants can perform on this object. Implementation of this idea, which we have not attempted, will of course require special hardware and communication protocol support [3].

In [23] we describe how other useful functions, not described in this paper, can be implemented within the same framework:

- *Concurrency control* among commands from multiple activities is supported by Ensemble using *timestamps* [15, 25] on command messages and object updates. Ensemble's centralized conference architecture ensures "mutual consistency" in that all participants' front-ends will see conflicting updates in the same order. However, stronger forms of consistency (e.g., "serializability" [7]) are not automatically guaranteed; these are left as options that the application implements if it wishes using the timestamp information that Ensemble provides.
- Conference and participant "lookup" facilities can be implemented by allowing nodes to release conference description information in response to queries from users.
- It is possible to add server nodes to a conference as "participants" and give them copies of some or all of the objects shared in the conference. This makes it possible to increase the availability of conference description information for queries, or to "move" a conference to a different server in case of a crash or impending shutdown of the original one.
- Recovery from front-end node crashes is supported by allowing a participant to "rejoin" a conference, at which point he receives up-to-date copies of all objects available to him in the conference. In addition, if the participant's front-end had previously "checkpointed" his copy of these objects, the timestamps on the checkpointed copies can be used by the

> server to send a log of intervening changes rather than complete new copies of the objects; no changes will need to be transmitted for an object that has not been updated in the meantime. (The same technique is used to restart a participant who has been absent from the conference for a long time.)

In [23] we also explore alternatives to the centralized architecture presented here, in which participants' "front-end" nodes can communicate directly; this can improve response time at the cost of more complex synchronization.

4.1. Implementation Status

An implementation of the shared bitmap facility is now complete (as of April 1984). To attract users for testing the system, our first application will be a conference in which participants cooperate to solve a crossword puzzle. Because the shared bitmap facility does not use the full flexibility of Ensemble, and passes many "upcalls" and "downcalls" between the application and Ensemble with little or no processing on its own part, we have chosen to implement the bitmap and Ensemble functions in a single "layer". The interface to the application is still layered, and closely resembles the Ensemble specification presented in the Appendix.

The underlying message transport mechanism is being implemented directly in terms of datagrams (using the DoD internet datagram protocol IP [22]) rather than virtual circuits.[8] When the same information must be sent to multiple receivers, datagrams allow for more efficient implementation than separate virtual circuits (between the server node and each front-end) because only one retransmission queue and timer need be maintained for all receivers. Also, retransmission of lost messages can be avoided in certain cases; for example, with mouse position reports it is more important to process the latest report quickly than it is to reliably receive all previous ones. (The use of timestamps in this case allows delayed obsolete reports to be discarded, as in packet voice transmission [3].) Datagrams would be even more efficient if *broadcast* or *multicast* facilities were available; while multicast is not currently included in DoD-IP, it is supported by the Xerox Network System protocols at the datagram level [4].

4.2. Related Work

Real-time conferencing in the form of "terminal linking" has been in existence since at least the early days of NLS [6], now marketed by Tymshare as Augment™. Similar features, with enhancements such as multiple "windows" and "virtual terminal" support for dissimilar display terminals, now exist on many systems. Terminal linking, whether physical or virtual, allows an

[8]The transport mechanism is based on a protocol designed by David Reed for coordinating a bitmap between a server and a single workstation; we have made several extensions to the protocol to deal with multiple workstations in a conference.

arbitrary existing program to be shared among a group of users by redirecting its output to multiple terminals and accepting input from any number of them. This has often been found useful, e.g., for debugging a program with which some other user encountered a problem. (Such joint debugging is often accompanied by a telephone conversation.) This kind of sharing, however, is accomplished at the lowest possible level of abstraction, namely the input and output character streams of the program; this does not allow for useful functions such as direct transfer of application information (as opposed to character streams) between different programs. The character stream model of program behavior is also somewhat out of date, considering the proliferation of directly-addressable bitmapped screens and pointing devices; the shared bitmap facility we have defined is an attempt to remedy this last problem.

More recently, some application-specific conferencing systems have appeared, e.g., as additions to the computer-aided design systems TOPES [21] and Palette [19]. While such systems have been useful in their particular application areas, they are typically specialized and inflexible, e.g., have a fixed definition of the "chairperson's" role or are restricted to groups of two users. Our objective has been to expose the underlying principles that are involved in designing such conferencing systems for any application. The Ensemble architecture presents a general framework in terms of which these existing conferencing systems, as well as terminal linking, can be described as special cases. In addition, it allows for extension to new applications and for experimenting with different conference "styles".

4.3. Open Systems Interconnection

We observe that the Ensemble "layer" that we have postulated corresponds very well to the OSI "Session" layer. A real-time conference is essentially a multi-party "session", and is independent of the underlying transport-level protocol being used to implement it. (For example, we do not require that the same transport-level connections be used for the duration of a conference, or that transport-level connections always be open. A conference can be created well in advance of the time when participants join, without setting up any transport-level connections until they are needed.) Ensemble supports the transmission of uninterpreted blocks of data ("session-service-data-units"), leaving it to the higher layer(s) to impose some structure and meaning (as objects and commands) to these blocks, and also supports the "recovery" of participants when their nodes or transport connections fail; this again matches the description of the OSI Session layer very well.

Multi-party network "connections", such as real-time conferences, have some characteristics that make them very different from two-party connections. In particular a multi-party connection can dynamically grow and shrink as parties leave and join; this distinctive feature is reflected in the structure of the Ensemble functions. Although the OSI reference model makes some mention of "multi-endpoint connections" (cf. Appendix A of [14]), there is little current activity in defining protocols for this important area. (This remains so despite considerable progress in distributed databases, for example.) The Ensemble architecture is an attempt to meet the need for a multi-endpoint Session layer protocol; we hope to stimulate further discussion toward the development of multi-endpoint protocol standards.

REFERENCES

[1] Chapanis, Alphonse, Ochsman, Robert B., Parrish, Robert N., and Weeks, Gerald D.
Studies in Interactive Communication: I. The Effects of Four Communication Modes On the Behavior of Teams During Cooperative Problem-Solving.
Human Factors 14(6):487-509, 1972.

[2] Clark, David D.
An Alternative Protocol Implementation.
Request For Comments 223, M.I.T. Computer Systems Research Group, May, 1982.

[3] Cohen, Dan.
A Protocol for Packet-Switching Voice Communication..
Computer Networks 2(4/5):320-331, September/October, 1978.

[4] Dalal, Yogen K.
Use of Multiple Networks in the Xerox Network System.
IEEE Computer 15(10):82-92, October, 1982.

[5] Davidson, J., Hathaway, W., Postel, J., Mimno, N., Thomas, R., and Walden, D.
The Arpanet TELNET Protocol: Its Purpose, Principles, Implementation, and Impact on Host Operating System Design.
In *Proc. Fifth Data Communications Symposium*, pages 4-10 to 4-18. September, 1977.

[6] Engelbart, Douglas C., and English, William K.
A Research Center for Augmenting Human Intellect.
In *Proc. Fall Joint Computing Conference*, pages 395-410. AFIPS Press, December, 1968.

[7] Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L.
The Notions of Consistency and Predicate Locking in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.

[8] Foley, James D. and van Dam, Andries.
Fundamentals of Interactive Computer Graphics.
Addison-Wesley, 1982.

[9] Goldberg, Robert N.
Software Design Issues in the Architecture and Implementation of Distributed Text Editors.
Technical Report DCS-TR-110, Rutgers Univ. Dept. Computer Science, 1981.

[10] Greif, Irene.
The User Interface of a Personal Calendar Program.
In *Proceedings of the NYU Symposium on User Interfaces.* New York University, May, 1982.

[11] Henderson, D.A., and Myer, T.H.
Issues in Message Technology.
In *Proc. Fifth Data Communications Symposium*, pages 6-1 to 6-9. September, 1977.

[12] Herlihy, Maurice, and Liskov, Barbara.
A Value Transmission Method for Abstract Data Types.
ACM Transactions on Programming Languages and Systems 4(4):527-551, October, 1982.

[13] Hiltz, Starr Roxanne and Turoff, Murray.
The Network Nation: Human Communication via Computer.
Addison-Wesley, 1978.

[14] International Standards Organization.
Reference Model of Open Systems Architecture, version 3, November 1978.

[15] Lamport, Leslie.
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of the ACM 21(7):558-565, July, 1978.

[16] Lantz, Keith A., and Rashid, Richard F.
Virtual Terminal Management a Multiple Process Environment.
In *Proc. Seventh Symposium on Operating Systems Principles*, pages 86-97. ACM, November, 1979.

[17] Lemmons, Phil.
BYTE West Coast: A Guided Tour of VisiOn.
BYTE 8(6):256 ff., June, 1983.

[18] Liskov, Barbara, and Scheifler, Robert.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[19] Mitchell, Neal B., and McLean, Michael.
Demonstration of Palette Teleconferencing Software, American Society of Civil Engineers Annual Conference, New Orleans, October 1982.

[20] Newman, William M., and Sproull, Robert F.
Principles of Interactive Computer Graphics, second edition.
McGraw-Hill, 1979.

[21] Pferd, W., Peralta, L. A., and Prendergast, F. X.
Interactive Graphics Teleconferencing.
IEEE Computer 12(11):62-72, November, 1979.

[22] Postel, Jon (editor).
Internet Protocol - DARPA Internet Program Protocol Specification.
Arpanet RFC 791, USC/Information Sciences Institute, September, 1981.

[23] Sarin, Sunil K.
Interactive On-Line Conferences.
PhD thesis, M.I.T. Department of Electrical Engineering and Computer Science, May, 1984.

[24] Stallman, Richard M.
A Local Front End for Remote Editing.
Memo 643, Massachusetts Institute of Technology Artificial Intelligence Laboratory, February, 1982.

[25] Thomas, Robert H.
A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases.
ACM Transactions on Database Systems 4(2):180-209, June, 1979.

[26] Tsichritzis, D.
Form Management.
Communications of the ACM 25(7):453-478, July, 1982.

APPENDIX: Ensemble Specification

We present here a list of "downcalls" and "upcalls" between the Ensemble and application layers, in the form of procedure names and argument types. We prefix each procedure name with an "arrow" to indicate the direction of the call, down (↓) or up (↑). We also prefix the call with an "F" or "S" to indicate whether it is a front-end or server call, respectively; calls that do not have either prefix (such as for "timers") apply to both front-end and server nodes.

We briefly explain here the different argument types accepted by the downcalls and upcalls:

- Server nodes and participants' front-end nodes are referred to by their network *addresses* (e.g., network number, host, and socket). How these addresses are determined, e.g., by name lookup, is not visible to Ensemble.
- Unique *ids* (identifiers) are used for referring to many kinds of entities: conferences ("confid"), participants ("pid"), objects ("objid"), activities ("actid"), and timers ("timerid"). Unique ids can be generated by standard techniques, e.g., by using the "timestamp" (below) at the time the entity in question is generated. Some kinds of entity identifiers need to be unique only within a given context, e.g., objects and activities in a conference; for these, it is possible to have much more compact ids, e.g., indexes into an array. (Note that users can be globally identified by their "fe-addresses", but within a conference a shorter "pid" is used to identify the participants.)
- *Timestamps* are used on all object update and command input messages. These are generated using a local clock value concatenated with the node's identifier in order to ensure global uniqueness. The clocks of different nodes are only approximately synchronized (e.g., using Lamport's method [15]) if at all.
- All objects, updates, and commands, are passed between the application and Ensemble in the form of programming language objects. It is assumed that some method (such as [12]) exists for "encoding" an object at the sending node into a linear sequence of bytes for transmission, and for "decoding" such a linear sequence at the receiving node into a copy of the original object; subroutines for doing such encoding and decoding are automatically invoked when objects are transmitted.

A few details have been skipped in the following presentation, such as calls that report errors (e.g., no server at the given address, or participant attempts to Send-Input in an activity that he no longer holds). We also do not specify subroutines that simply retrieve information or components of data structures, e.g., reading the local clock or determining the address of the participant with a given "pid" (or vice versa).

F↓*Create-Conference*(server-address,description)

Asks the server at the given address to start a new conference with the given "description" (see Section 2). The server will automatically add the front-end issuing this downcall as a participant in the new conference.

S↑*Initialize-Conference*(confid,description,timestamp)

In response to a Create-Conference, the application layer at the given server address is asked to initialize its state information for a new conference with the given description.

The description carries the given initial timestamp; its timestamp will change only if the description is updated.

S↓*Add-Participant*(confid,fe-address) returns(pid)
: The application-server requests that the participant at the given address be added to the given conference; Ensemble-server returns the new participant's "id" within the conference for future reference.

F↑*Added-To-Conference*(confid,description)
: The application-FE is informed that it has been added to the given conference, with given "description", as a participant. The application-FE is expected to either Join or Leave (below).

F↓*Join-Conference*(confid,description-timestamp)
: The application-FE indicates that the participant wishes to join the given conference. "Description-timestamp" is used by the server (below) to determine whether the participant has an up-to-date conference description object or whether a new version needs to be sent.

S↑*Participant-Joined*(confid,pid,fe-address,description-timestamp)
: Upcall received by application-server when the participant's application-FE issues a Join-Conference downcall.

F↓*Leave-Conference*(confid)
: Participant does not wish to remain in the conference. (Can be issued at any time, when added to the conference or later.)

S↑*Participant-Left*(confid,pid)
: Server upcall resulting from Leave-Conference.

S↓*Remove-Participant*(confid,pid)
: Remove the given participant from the conference.

F↑*Removed-From-Conference*(confid)
: Front-end upcall resulting from Remove-Participant, or from Terminate-Conference.

S↓*Terminate-Conference*(confid)
: Terminates the given conference; all participants' application-FEs get Removed-From-Conference upcalls.

S↓*Add-Object*(confid,obj) returns(objid)
: Add the given object to the conference; "objid" is returned for future reference. The object is not yet "shared" with any participants, but can be using Give-Object.

S↓*Give-Object*(confid,objid,pid)
: Make the specified object in the conference available to the given participant.

F↑*New-Object*(confid,objid,value,timestamp)
: Participant's application-FE is informed of Give-Object. Is expected to either Accept-Object or Decline-Object, below.

F↓*Accept-Object*(confid,objid,timestamp)
: Indicates willingness to process updates to the given object. "Timestamp" is used in the same way as "description-timestamp" in Join-Conference (above).

S↑*Object-Accepted*(confid,pid,objid,timestamp)
: Server upcall resulting from Accept-Object.

F↓*Decline-Object*(confid,objid)
: Participant's front-end does not wish to receive updates to the given object.

S↑*Object-Declined*(confid,pid,objid)
: Server upcall resulting from Decline-Object.

S↓*Revoke-Object*(confid,objid,pid)
: The given participant is to no longer receive updates for the given object.

F↑*Object-Revoked*(confid,objid)
: Front-end upcall resulting from Revoke-Object, or from Remove-Object (below).

S↓*Remove-Object*(confid,objid)
: The given object is no longer shared in the conference. Every participant's application-FE receives an Object-Revoked upcall.

S↓*Update-Object*(confid,objid,change-desc) returns(timestamp)
: Send a description of an update to the given object, to all participants with whom the object is shared. The timestamp associated with this update is returned.

F↑*Object-Updated*(confid,objid,change-desc,timestamp)
: Front-end upcall resulting from Update-Object. The application-FE should interpret the "change-desc" in order to update its copy of the given object.

S↓*Sync-Object*(confid,objid) returns(timestamp)
: Wait for an acknowledgement, that the front-end has received and processed all updates to the given object, from the front-end of each participant holding copies of the object. Acknowledgements are automatically generated by the participants' Ensemble-FEs when the application-FE's handler for Object-Updated returns; Decline-Object suffices as a "negative" acknowledgement.

S↑*Sync-Complete*(confid,objid,timestamp)
: Acknowledgements received from all associated participants.

S↓*Abort-Sync*(confid,objid,timestamp)
: Stop waiting for acknowledgements. Typically used in response to Timer-Expired (below), when not all acknowledgements have been received.

S↓*Add-Activity*(confid,info,pid) returns(actid)
: Create a new activity assigned to the participant with given "pid". "Info" carries a specification of the commands available in this activity.

F↑*Activity-Received*(confid,actid,info)
: Front-end upcall from Add-Activity, or Give-Activity.

S↓*Give-Activity*(confid,actid,pid)
: Give the specified activity to the specified participant, who will receive an Activity-Received upcall. The participant who currently has the activity receives an Activity-Revoked upcall.

F↑*Activity-Revoked*(confid,actid)

Front-end upcall resulting from Remove-Activity or Give-Activity.

S↓*Suspend-Activity*(confid,actid)
Temporarily disallow commands in the given activity.

F↑*Activity-Suspended*(confid,actid)
Front-end upcall resulting from Suspend-Activity.

S↓*Resume-Activity*(confid,actid)
Allow commands in the given activity once again.

F↑*Activity-Resumed*(confid,actid)
Front-end upcall resulting from Resume-Activity.

S↓*Remove-Activity*(confid,actid)
Removes the given activity from the conference.

F↓*Send-Input*(confid,actid,command-desc) returns(timestamp)
Send the description of a command under the given activity.

S↑*Input-Received*(confid,actid,pid,command-desc,input-timestamp,readset-timestamp)
Server upcall resulting from Send-Input. "Input-timestamp" is the value of the participant's clock at the time of Send-Input; "readset-timestamp" is the timestamp of the last object update that the participant's front-end had received at the time (can be used for concurrency control).

↓*Set-Timer*(time,obj) returns(timerid)
Set a timer to expire at the specified time. Can associate an arbitrary object, which is passed back by Timer-Expired, below.

↑*Timer-Expired*(timerid,obj)
The specified timer has expired.

↓*Postpone-Timer*(timerid,new-time)
Reset the given timer to go off at a different time.

↓*Abort-Timer*(timerid)
Discard the given timer, i.e., don't give a Timer-Expired upcall.

↑*Background*()
Upcall generated when there are no events waiting to be processed, i.e., the node is "idle".

↓*Set-Background-Interval*(interval)
Ask to receive Background upcalls only when the specified amount of "idle" time has passed. (This interval is initialized to some standard default value.)