BUILDING FLEXIBLE GROUPWARE THROUGH OPEN PROTOCOLS

Mark Roseman
Saul Greenberg

Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4
(403) 220-6015 roseman,saul@cpsc.ucalgary.ca

ABSTRACT

This paper presents a technical approach to building flexible groupware applications. Flexibility provides the promise of *personalizable groupware*, allowing different groups to work with the system in diverse ways which best suit the group's own needs. An implementation technique called *open protocols* is described, which is a variation of client/server architectures. Open protocols facilitate the addition of group-specific modules long after the system has been created. Three examples illustrating the use of open protocols are presented: floor control, conference registration, and brainstorming. Finally, a number of issues facing the groupware developer using open protocols are addressed, along with strategies that can help in dealing with these issues.

KEYWORDS

Personalizable groupware, expandability, open protocols, implementation technique.

INTRODUCTION

Conventional single-user applications are designed to capture a profitable market share. While the product may not be to everyone's tastes, the vendor's goal is to have it acceptable to enough customers to make its production an economically worthwhile venture. Those customers with different requirements or preferences simply go to another product, or do without. Since one person's choice of product rarely affects anothers, the resulting heterogeneous application environments are quite tolerable.

Designers of groupware face more rigid criteria. Of course, the product must satisfy enough groups to be commercially viable. But unlike single user software, the product chosen by the group should be acceptable and usable by nearly *all* its members, for groupware acts as a common medium through which people communicate, coordinate, and pursue their common goals.

If the groupware cannot accommodate its potential users, then conditions can occur that contribute to its failure [7].

- A critical mass of system adopters may not be reached if too many people opt out of using the groupware product, e.g. see Markus and Connolly's [15] discussion of payoff criteria for adopting technology.
- Participants who cannot or will not use the technology face the danger of becoming second class citizens within their own group [1].
- New people joining an established but evolving group must quickly become proficient at using the system, otherwise cliques of expertise may evolve.
- Participants in a group may have quite different roles that are not recognized by the groupware product, e.g. see Austin, Liker et al [1].
- The disparity between who does the work and who gets the benefit of groupware leads to varying degrees of its acceptance [3,8].
- Group needs evolve rapidly, not only from meeting to meeting but within the course of a meeting. The groupware must keep pace, e.g. see Stefik, Bobrow, Foster et al's [22] discussion of the rapid formation and dissolution of subgroups within a meeting.
- Forcing groups to use a single groupware system may trigger both individual and organizational resistance, and may cause users to do things in undesirable and unproductive ways [9]. At its worst, users will perceive such systems as "fascist software," e.g. see Bair and Gale's [2] report on the COORDINATOR.

Some of these conditions could be partially avoided if the environment was homogeneous, that is, if all people were content with a single groupware product. This can occur when groupware is so transparent that almost anyone can use it (such as teleconferencing), or when the service provided is so valuable or so entrenched in an organization that all users are effectively forced to use it.

In reality, homogeneity is almost impossible to achieve. The differences present between group members—their varying and evolving roles, needs, skills—and the differences between groups as a whole are serious obstacles to achieving uniform acceptance of a single groupware product or style, especially if the product treats all users and groups identically.

This paper presents a compromise solution by arguing that homogeneous use of a groupware product is possible in a heterogeneous environment by making the groupware

flexible enough to accommodate individual roles and group differences.

Personalizable and Flexible Groupware

Personalizable groupware is defined as groupware that users can tailor match their needs (each member of the group may observe a different behavior), and that groups as a whole can change to match their overall needs (each group may observe a different collective behavior) [7]. We see personalizable groupware offering its users a range of behaviors that reflect a complementary range of the groups' requirements.

The catch with personalizable groupware is that it is difficult, if not impossible, for a designer to predict ahead of time the complete range of behaviors the system should exhibit. One solution is to make the groupware *flexible* as well, so that new behaviors can be created and old ones modified in ways that the original designer had not anticipated. This is not advocating the chaos of a completely customizable or unstructured system. We minimally expect that the groupware designer would determine what parts of the groupware system should remain immutable, what parts should be flexible, and would then set reasonable constraints on the personalization allowed.

After briefly describing some related work, this paper will present what we believe to be a fairly general approach to building flexible and personalizable groupware called *open protocols*. Because it is more of a concept or architecture than an algorithm, we ground the idea by illustrating how three different groupware components—floor control, conference registration, and brainstorming—can all be built in a flexible manner. However, flexibility comes at a cost, and we raise issues such as synchronization and conflict control, how open protocols can be designed when there is uncertainty on how a system will be used, who constructs the new components, and security.

RELATED WORK

A handful of groupware systems are personalizable, and a few are described here. By personalizable we mean that users of each system can select from a set of different behaviors. QUILT is a multi-user asynchronous document editor that allows its users to assume software-defined roles such as reader, commentor, and co-author [11]. Each role gives the user greater powers of annotation and revision. CRUISER is a media space that lets its users set privacy permissions that limit how others can observe and/or interact with them [18]. The VIRTUAL LEARNING COMMUNITY is an asynchronous conferencing system that lets a conference facilitator tailor the groupware to support the purpose and the variety of the group's activities [9]. For example, the boundaries that define group membership can be adjusted to either enforce equal participation of all group members, or to allow "lurkers"—people who follow the group's discussion but who never express themselves.

There are also several examples of flexible groupware. By flexible we imply some degree of constrained open-ended tailorability, where the system can be changed in ways not foreshadowed by the original developer. INFORMATION LENS is an information manager for mail and news. It is flexible because end users can construct their own semi-structured templates representing different types of mail, and can create their own rules to filter and view incoming information in quite sophisticated ways [13]. Next, there are a variety of "conversation-based" systems that provide flexibility in how different speech act units and the protocols between them are defined; the idea is that conversation and the rules that define it are tailored to the group needs. Examples are STRUDEL [21], CHAOS [4], OVAL [14], and CONVERSATION BUILDER [10]. Another approach gives a programmer the ability to configure low-level system constructs, such as the degree of "sharing" or coupling exhibited by interface components [5], or the degree of access control [20].

Our own approach of *open protocols* differs from the above work because it is a general implementation technique that can be applied to a variety of domains and situations. Its only requirement is that the system behavior can be configured by altering state information, and that personalization is achieved by constructing different modules to control state changes through a client/server architecture. The following sections describe the technique in detail.

OPEN PROTOCOLS

Open protocols are an implementation technique that can be used to build flexible groupware systems that accommodate the diverse needs of different groups. The technique allows a designer to add new modules into systems long after the original system has been implemented. Modules are tailored to fit user and group needs, alter the way the system behaves, and present an interface to the user appropriate to the behavior.

The general idea is that the core system is a server containing state information and obeying a primitive, pre-defined protocol for determining how its state is changed. Modules are external client processes that encapsulate high-level concepts as permutations of these primitive protocols; each module can then control the behavior of the core system by transmitting directives for changing the system state. Modules also can present an interface to the user that is specific to that behavior. Because open protocols are a variant of a client/server architecture, they can be implemented on any machine and software architecture supporting multiple processes or objects—potentially on different machines—that can communicate with each other, for example through TCP/IP stream sockets.

More specifically, open protocols have three components: a *controlled object* (server) that maintains state, a *controller object* (client), and a *protocol* describing how the two communicate. Figure 1 shows the relationship between these three components. The controlled object does not maintain any policy regulating how its state can be manipulated. Instead, a protocol for changing the state—typically allowing extreme changes—is defined. The controlled object obeys any external requests made using the protocol to change its state. The controller, external to the

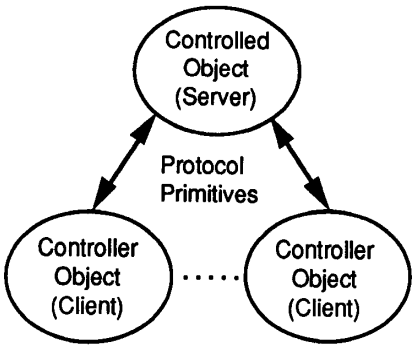

Figure 1. Relation of controlled object (server), controller objects (clients) and protocol in open protocols.

controlled object, implements a particular interface and groupware behavior by the requests it sends to the controlled object.

This model is in sharp contrast to most client/server architectures, where policy traditionally resides in the server, not the client. Using open protocols, the responsibility is for the client (controller), not the server (controlled) to implement policies; the server merely obeys a set of generic commands from clients according to its protocol.

The designer of an open protocol system must decide what state changes are reasonable to embed within the controlled object, and what protocol it should obey. This both standardizes the system's behavior and sets the constraints on its flexibility. A variety of clients can be created at any time afterwards and transparently integrated into the system. Adding a new client is as simple as opening up a connection (e.g. TCP socket) to the controlled object and transmitting the appropriate protocol. Provided the client uses the defined protocol, the controlled object needs no other information about the client.

As a result, new and diverse clients, each with their own unique interface and behavior, can be created at any time and added on to the main system, without the need to make any changes to the controlled object, not even recompilation. It is possible—within bounds—to create clients with behaviors that were never thought of when the protocol and controlled object were initially designed and implemented. This results in clients designed to suit the needs of particular users groups, accommodating their preferred working styles.

Because open protocols is a fairly abstract idea, the next three sections ground the concept in a variety of examples. We being by showing how a variety of floor control policies can be integrated with a simple shared screen system. A second, more sophisticated example illustrates conference registration and the various ways people can create, join and leave a conference. The final example illustrates how a generic brainstorming tool can be constructed that facilitates a variety of uses in different social situations. All show the diversity of the clients that can be created from simple open protocols.

FLOOR CONTROL

Open protocols are a generalization of work on floor control in the SHARE system [6,7]. Under that system, floor control is used to mediate access to a shared terminal. "Floor holders" can type to the shared screen, while those not holding the floor can only observe.

Rather than implementing a single floor control policy or even a small, fixed set of policies into the system (as done in most shared view systems [7]), a protocol is defined in SHARE whereby independent modules can be attached to the system to manage floor control. The shared terminal itself maintains a two-state flag for each user in the conference. If the flag is set to "write", the user can type to the shared terminal, while if the flag is reset to "observe", the user's input is ignored. External floor control modules can attach to the shared terminal and set or reset the flag for any user. State changes are sent from the shared terminal back to all attached floor control modules.

This scheme allows a wide variety of floor control policies to be built, such as round-robin, free floor, preemptive, explicit release, and central moderator (see [7] for full details). The floor control modules are developed independently from the main shared terminal, and others can be added dynamically.

Under this scheme, the shared terminal acts as the controlled object, maintaining state information in the form of an observe/write flag for each user. The floor control modules act as the controller objects or clients, specifying the pattern of state changes of the flags in the shared terminal. Finally, the protocol between the shared terminal and the floor control modules is open in the sense that it supports a wide variety of different policies, as defined by the floor control modules and not the shared terminal. A subset of the scheme implemented in SHARE is illustrated in Figure 3. Set_write and set_observe both direct the shared terminal to change the status of the indicated user, while state (user) returns a particular user's current status.

Figure 2 suggests how a number of different floor control policies could be implemented. Each policy is controlled by the user through an interface, such as the one shown in Figure 4. Pre-emptive, round robin and moderator controlled policies are all quite straightforward. However, the explicit release policy brings up the problem of conflicts. Assume the current floor holder releases the floor and two other users attempt to grab the floor simultaneously. Both first check that the floor is empty (it is). They both set their own flag to allow writing. The end result will be both users allowed to write at once. This problem is considered later.

CONFERENCE REGISTRATION

Open protocols are the basis for conference registration—the process of creating, deleting, joining and leaving groupware applications—in our groupware toolkit, GROUPKIT [19]. GROUPKIT provides a registration system

Pre-emptive	Round Robin	Explicit Release	Moderator Controlled
Description: Any user may at any time grab control from any other floor holders.	*Description:* When done with the floor, it is passed along to the "next" user, assuming a known, circular list of users.	*Description:* The floor is released explicitly by the floor holder. When empty it can be grabbed by any user.	*Description:* A moderator controls which users can hold the floor (none, one, several or all).
on grab_floor: foreach user if (user=me) set_write(user) else set_observe(user)	*on release_floor:* foreach user if (user=next) set_write(user) else set_observe(user)	*on release_floor:* foreach user set_observe(user) *on grab_floor:* foreach user if (state(user)=write) return set_write(me)	*on allow_write(user):* set_write(user) *on disallow_write(user):* set_observe (user)

Figure 2. Four different floor control policies implemented using open protocols.

decoupled from the main groupware application. The goal here is to be able to accommodate different styles of registration, to best fit the needs of different groups. GROUPKIT registration schemes could be constructed to support structured meetings, informal contact, meetings in the same room or at a distance.

In the system, a central Registrar server maintains lists of conferences and their users, while different Registrar Clients can connect to the Registrar to alter these lists. Here, the Registrar is the controlled object, and the Registrar Client is the controlling object. The two objects, and the protocol by which they communicate, are shown in Figure 5.

The Registrar responds to any request from its clients, broadcasting the result to all attached Registrar Clients. This allows any client to ask the Registrar to add a new conference, or conceivably even to delete any user from any existing conference. While this does make it possible to create a "super-user" version of the Registrar Client, it also provides the flexibility to create any number of other Registrar Clients interfacing to the Registrar, without making any changes to the Registrar itself.

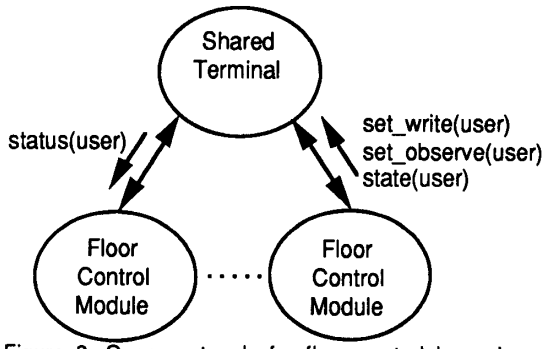

Figure 3. Open protocols for floor control in a shared terminal.

As an example, consider the implementation of an *open door* registration policy, where any user may create a conference, and users can join any existing conference. Its interface is shown in Figure 6. The underlying implementation is straightforward. To join an existing conference, the Registrar Client sends an "add-user" message to the Registrar. This is broadcast to the other Registrar Clients in the selected conference which update their own users lists (right top window, Figure 6). The Registrar Client also creates a new application Conference, because under the policy, the user is guaranteed entry to the conference. The Conference makes connections with the other users, the application appears on the display, and interaction proceeds normally.

In contrast, a *closed door* registration policy (not illustrated) does not permit new users to join an existing conference unless "sponsored" by an existing conference participant. Here the Registrar Client again sends an "add-user" message, which is broadcast to the other users. At this point, the local Registrar Client does *not* create a new Conference. The remote users are asked by their Registrar Clients if the new user should be accepted. If a remote user

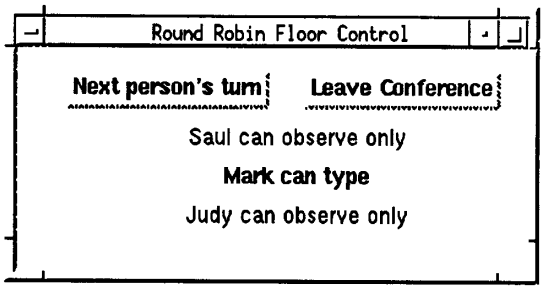

Figure 4. One possible interface for a floor control module.

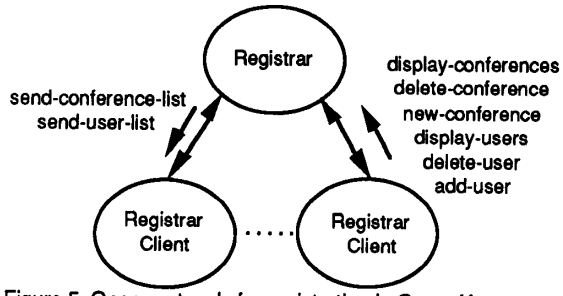

Figure 5. Open protocols for registration in GROUPKIT.

sponsors the new user, their client sends a message directly to the new user's client, prompting them to create the conference as before. If rejected (either explicitly or by timeout), the rejecting client sends a "delete-user" message to the Registrar.

Cooperating Clients

It is also possible to build different clients that cooperate with each other. Some conference users would work with one type of client, while other conference users work with a different client.

For example, a *facilitated* Registrar Client can be created, emulating the mediated registration policies often found in group support systems meetings [16]. A central facilitator running one client can create several conferences

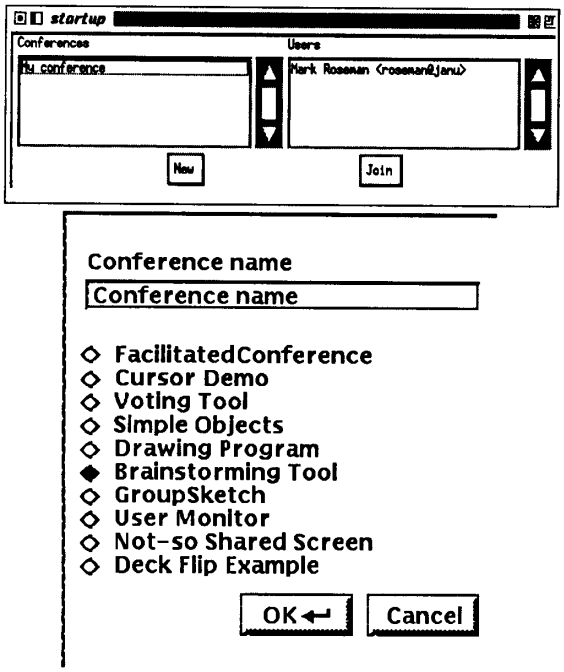

Figure 6. Interface for an open door registration policy. The top window shows the list of conferences and users in the selected conference, while the bottom window allows users to create new conferences, selecting from the available list of groupware applications.

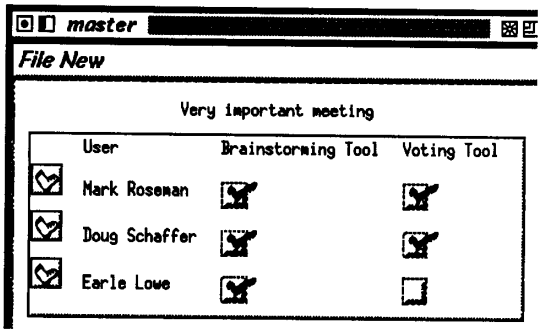

Figure 7. Interface for a facilitated registration policy. Facilitators can create new conferences (under the "New" menu), add or delete users from these conferences (using the check boxes) or remove users from the facilitated meeting (using the boot icon beside each user).

(applications) for the group, such as brainstorming sessions or voting. The client interface is shown in Figure 7. Users in the facilitated meeting use a different client, which merely obeys the requests from the facilitator's client. This client presents no interface other than an initial dialog box prompting the user to join a facilitated conference. The two clients are designed to cooperate with each other, as shown by their protocols in Figure 8.

The facilitator could cause a user to join a brainstorming session by sending an "add-user" message for the user and conference to the Registrar. The user's client, receiving the

Facilitator Controlled	User in Facilitated Meeting
on initiate-new-conf: send new-conference	*on new-conference:* do nothing
on new-conference: if (initator=me) create_conference(conf) add-user(conf, me) join_conference(conf)	*on delete-conference:* delete_conf(conf)
on initiate-add-user: add-user(conf, user)	*on new-user:* if (user=me) create_conference(conf) join_conference(conf)
on initiate-delete-user: delete-user(conf, user)	*on delete-user:* delete_user(conf,user) if (user=me) delete_conf(conf)

Figure 8. Open protocols for conference registration. These cooperating clients are used within the same meeting, the first by the meeting's facilitator, and the other by the remaining meeting participants. This figure shows only a subset of the possible actions and messages that occur with these clients.

Anonymous	Public	Selective Delete (Own ideas identified)	Full Delete (All ideas identified)
on user_idea: write_idea(new_idea_id, owner_id, posn_last, the_idea)	*on user_idea:* write_idea(new_idea_id, owner_id, posn_last, the_idea)	*on user_idea:* write_idea(new_idea_id, owner_id, posn_last, the_idea)	*on user_idea:* write_idea(new_idea_id, owner_id, posn_last, the_idea)
on update_idea: update_display(the_id, the_posn, the_contents)	*on update_idea:* update_display(the_id, the_posn, Lookup(the_owner):: the_contents)	*on update_idea:* if (the_owner=MY_ID) owner = Lookup(the_owner) else owner = "" update_display(the_id, the_posn, owner::the_contents)	*on update_idea:* update_display(the_id, the_posn, Lookup(the_owner):: the_contents)
on user_delete: do_nothing	*on user_delete:* do_nothing	*on user_delete:* if (the_owner=MY_ID) delete_idea(the_id)	*on user_delete:* delete_idea(the_id)

Figure 9. Open protocols for brainstorming. Four different clients respond to several events: (a) an idea is entered by the user, (b) server, and (c) the user elects to delete an idea. Note this last option is not available in two of the clients.

message via the Registrar, would obey the request, creating the application Conference and connecting to other users. The facilitator could then remove the brainstorming session by sending a "delete-user" message for the user and conference. The user's client would again receive this message and obey it, deleting the brainstorming conference.

To reiterate, the point is not that GROUPKIT provides several registration policies, but that its use of open protocols permit a wide variety of registration schemes to be implemented.

BRAINSTORMING

As a final example of the use of open protocols, Figure 10 suggests a possible architecture for a brainstorming system, where ideas are stored on a central "idea server" connected to a number of brainstorming client programs. A variety of different clients are possible here, suitable for different users in a group or for different group dynamics.

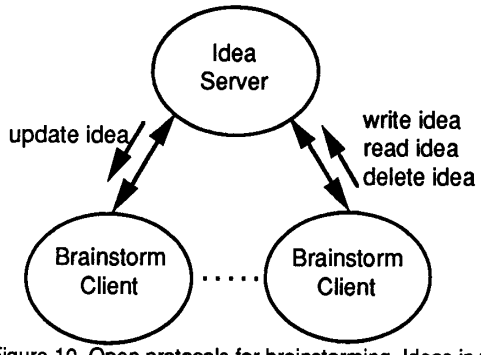

Figure 10. Open protocols for brainstorming. Ideas in the server consist of a unique ID number assigned by the Idea Server, an owner ID, the position in the idea list, and the actual text of the idea itself.

Figure 9 shows the open protocol implementation of four such clients. One client is a standard anonymous brainstorming client, where the identity of the idea's owner is hidden (illustrated in Figure 11). Alternatively, the owner of ideas may be displayed publicly. If ideas are to be deleted, as might occur in later stages of a brainstorming meeting, the group may decide that any user can delete only their own ideas, or on the other hand, that anyone's idea can be deleted by a user. Note that a facilitator, using the registration interface described previously, might control the use of different interfaces during the course of the meeting.

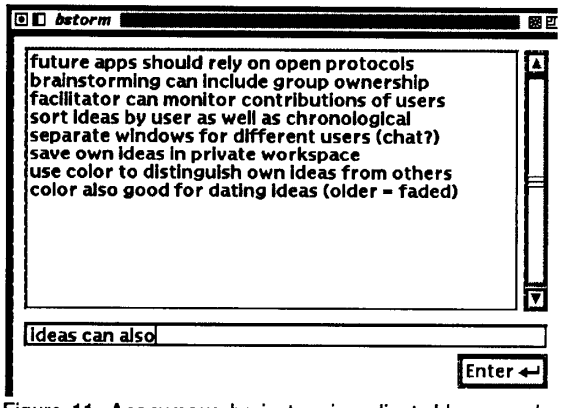

Figure 11. Anonymous brainstorming client. Ideas can be entered by any user, and are displayed without attribution for all users to see. No deletion of ideas is possible with this particular client.

Possibilities for Future Clients

To illustrate the variety of possibilities that the open protocols strategy can provide, consider the following "brainstorming clients" which could be easily integrated into the brainstorming architecture described.

Idea organizers. Because the idea server maintains a "position" in the list for each idea, idea organizing applications can be built. Users can rearrange ideas in the list to group related ideas. As with deleting ideas, users may be restricted to moving their own ideas or they may be allowed to move any other user's ideas.

Group membership. A number of the interfaces differentiate between ideas generated by the "local" user and those of other conference users (for displaying names, deleting or moving). This idea could be extended to support subgroups within a larger meeting. Ideas generated by a subgroup member could be moved and perhaps deleted by any subgroup member, not just the original creator. Subgroup maintenance would need to be handled by the clients themselves, perhaps even using an additional "subgroup" open protocol server.

User monitors. A client could be built that simply monitors the generation of ideas, noting the number of ideas generated by different group members or the group as a whole and displaying this as a bar graph. A facilitator could use such a client to monitor individual or group progress through a task and then react accordingly.

Different interfaces. Because the protocol does not depend on the user interface of the clients but rather the operations the interfaces invokes, a variety of interfaces are possible. Textual or graphical interfaces, monochrome or color, or interfaces built with different toolkits or languages are trivially incorporated into the architecture.

Portability. Similarly, interfaces running on different platforms are possible. The main requirement is that the underlying communications mechanisms (i.e. low level sockets) are the same. If the communications is based on widely available protocols such as TCP/IP, clients could be written for Unix machines, Macintoshes, PCs, or other systems.

ISSUES

The previous sections have described several examples of using open protocols to build flexible, open-ended groupware applications. This section now turns to the issues that can arise when new applications are designed with open protocols.

Synchronization and Conflict

Recall the problem of conflict that can arise with the "explicit release" floor control protocol. Under this protocol, the floor may be grabbed by any user only when it has been explicitly released by the previous floor holder. Problems can arise if two users attempt to grab the floor simultaneously.

This can be illustrated by looking at the following implementation of the grab floor method:

```
foreach user
 if (state(user) = write)
  return
set_write(me)
```

This method first checks whether any user is currently holding the floor, and if not, the floor is grabbed. Yet this can result in two users holding the floor. There will be a delay between the time the floor is identified as empty and the "set_write(me)" is propagated to all users. During that time, the floor appears empty so another user can call "set_write(me)" resulting in both users gaining the floor.

A more sophisticated solution would not only set the local user's flag, but also reset other users' flags:

```
foreach user
 if (state(user) = write)
  return
foreach user
 if (user=me)
  set_write(user)
 else
  set_observe(user)
```

Yet again there can be synchronization problems. This method works fine assuming that all "set" requests from a particular user are processed sequentially. However, if requests from two users are intermixed—some requests from the first user are processed, then some from the second, the remainder from the first, etc.—problems can arise. While this can result in a single floor holder (the desired result), it is also possible to have two floor holders (if the "set_write" of each user comes after the corresponding "set_observe" of others) or no floor holders (if the "set_observe" requests override the "set_write" requests).

This problem can be solved by treating the entire set of "set" requests sent by a single user as a complete *transaction.* All the requests in the transaction must be processed together. A very simple way of achieving this is by *batching* the requests together into one larger request. Rather than sending several smaller messages—which may be interspersed with messages from other clients when they reach the server—a larger message is sent and processed together.

To illustrate, let us say that the low level floor control protocol communicated to the server maps "set_write(user)" to the command "Wuser" and "set_observe(user)" to "Ouser" where "user" is a unique integer identifying each user. Assuming the floor requestor is user number 3 and other users in the conference have user numbers 1,2, and 4, rather than sending the protocol:

```
O1
O2
W3
O4
```

where each line is a separate message, a composite message batching all the requests can be sent:

```
O1|O2|W3|O4
```

Batching messages together into a single transaction is easily handled by the client/server model, and can solve many of the synchronization and conflict problems that can arise when using open protocols. The only assumption is that the server will process an entire message at once.

If even more sophisticated synchronization schemes are necessary, it is also possible to build them directly into the protocol. For example, a locking scheme is easily built.

Predicting the Future

One of the inherent difficulties in effectively using open protocols is that the benefit gained (in terms of the number of different clients that can be accommodated) directly depends on the original designer's ability to predict the future—not a particularly exact science! While the designer does not have to foretell exactly what modules will be created, the protocol primitives that are defined do constrain the system's flexibility.

Though difficult to predict all the different uses that users may have for the data maintained by the server, some general guidelines can help a designer create a widely usable protocol.

Think generally of the data's uses. While it is easy to think in terms of particular clients, try to think in terms of all the uses the server's data could be put to. In the brainstorming example discussed previously, the natural tendency would be to think of the data as usable just for brainstorming. Yet if the data is considered as "ideas" more generally, other possibilities begin to suggest themselves. Clients for organizing, grouping or ranking the ideas can be seen as possible options.

Design primitive operations. Another advantage of thinking in terms of the data rather than particular clients is that primitive operations become more apparent. Using floor control as an example, an alternative but poorer approach to the "flags" for each individual user would be a single "floor control token" which could be passed between users. Whoever "holds" the token holds the floor.

While this scheme works reasonably well, it is more restrictive than the flag per user scheme. In particular, the token passing scheme does not allow more than one user (but not all) to hold the floor at one time. The token passing scheme, by assuming some of the semantics of typical clients, results in a less general scheme than the flag per user scheme. The latter, being more primitive, is actually more powerful and expressive.

Provide back doors. No matter how thoughtfully designed a protocol is, there will always be someone who wants to do something that cannot be accommodated within the protocol. The solution here is to design "back doors" into the system so that clients can bypass the server when there is a need.

In both SHARE's floor control and GROUPKIT's registration mechanisms, all clients know about each other (i.e. host and port number), and can establish direct connections. While communications between different clients are normally mediated by the server through the open protocols, clients may communicate directly to each other for extraordinary needs. While it is of course better if the protocol can cover the necessary cases, back doors are a practical solution to push beyond the constraints inherent in the protocol.

Enhance the protocol when needed. The protocols defined in the server should not change often, otherwise existing clients will break. Yet there will probably come a time for change. If numerous clients all use the same "back doors" to implement the same features, those features are probably best incorporated into the protocol. Preferably they can be added without disturbing the existing protocol, and servers will remain "backwards compatible." For example, a time-out mechanism was added to SHARE's floor control protocol because several clients had the need for pause detection, ie. they took some action after the floor holder was idle for a given period of time.

Ease of Building New Clients

Open protocols provide—at least in theory—the opportunity for end users to actually design their own clients for their particular needs. While system designers may provide a library of "common" client interfaces for end users to select from, it would be nice if end users could design their own clients for their own unique needs. How feasible is this?

While this question depends somewhat on the actual protocol—if it is very complex it is unlikely users will be able to express their needs in terms of the protocol—this difficulty is probably secondary compared to the problems of actually making the software tailorable by the non-programmer.

To take GROUPKIT's registration system as an example, it would be quite unreasonable to expect typical end users to design a new registration client interface. End users would need access to a C++ compiler, a reasonable knowledge of the C++ language, and familiarity with a sophisticated user interface toolkit (INTERVIEWS [12]).

We have begun experimenting with alternative ways of developing such interfaces. One promising approach utilizes the TCL language, which provides an embeddable interpreter similar to many shell programming languages [17]. Because the syntax is relatively straightforward and the need for compiling is removed, the level of sophistication needed to write code is dramatically reduced. To aid in the coding, generic procedures implementing the protocol primitives can be provided. For example in the floor control example, procedures for "set_write(user)" and "set_observe(user)" can be provided as TCL primitives.

The user interface portion of the program might be best specified using an interface builder, or at least a very simple interface library, and several exist for TCL. A tool such as HYPERCARD is another possibility, providing an easy way to build interfaces, and using XCMDs to provide generic procedures supporting the particular protocol.

Will such schemes be sufficient to allow end users to develop their own groupware interfaces? Although it is too early to tell, it would in principle considerably simplify the development of personalizable groupware interfaces. While this may not allow all end users to develop interfaces, it may be sufficient to foster "local experts" who, with some initial learning, may be able to help support a community of users.

Security

Because open protocols permit any client using the protocol to interact with the controlled objects, this raises the specter of malicious clients, and security violations. In offering the flexibility, open protocols do open security risks. For example, a malicious Registrar Client could easily delete users from conferences in the GROUPKIT registration example. We offer several reasons explaining why this may not be particularly troublesome in practice.

Physical and network security. There will be little need for security in the software if it is operating in a physically secure area, for example a computerized meeting room on its own secure network. As intruders will not be able to access the server, the only clients available will be those installed on the local computers by the system administrators.

Organizational pressures. In many organizations, social and organizational pressures will tend to prevent such abuses, because of the sense of community and trust present in many workgroups, or possibly the ramifications of such inappropriate actions. Security may be more of a concern in loose organizations such as the Internet, but even so there may be channels whereby problems with malicious users can be resolved.

Implementing security. Finally, nothing actually prevents security measures from being programmed into the systems. Security may take the form of restricted access lists, password protection, or other such authentication and approval mechanisms. Open protocols do not prevent such mechanisms from being implemented within a client/server dialogs.

CONCLUSIONS

We have argued that successful groupware should be flexible enough to accommodate the differences inherent in groups. Such personalizable groupware must therefore adapt to a wide variety of different group behaviors, including behavior not originally expected by the groupware developer.

Open protocols provide a technique that addresses this requirement. Implemented as a variation of the client/server architecture, open protocols specify that access to the server's data is controlled by the client, not the server. As a result, a large number of different clients can potentially control the server. The diversity of possible clients was illustrated in the domains of floor control, conference registration, and brainstorming.

To further aid the groupware developer working with open protocols, a number of important design issues were discussed, along with suggestions for dealing with the issues. These included issues of synchronization and conflict, predicting the future, end user client development, and security.

We have found open protocols to be not only a very powerful technique for building personalizable groupware, but also one that is very straightforward to use. Its power is its simplicity—complex operations can be built from simple primitives. We believe that open protocols can result in more flexible, easily expandable groupware systems.

ACKNOWLEDGEMENTS

Thanks to Ted O'Grady and Doug Schaffer for comments on early drafts of this paper. This research was supported by the National Sciences and Engineering Research Council of Canada.

REFERENCES

1. Austin, L. C., Liker, J. K. and McLeod, P. L. (1990) "Determinants and patterns of control over technology in a computerized meeting room." In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '90)*, pp. 39-52, Los Angeles, California, October 7-10, ACM Press.
2. Bair, J. H. and Gale, S. (1988) "An investigation of the Coordinator as an example of computer supported cooperative work." Hewlett Packard Laboratories, California. Unpublished.
3. Bullen, C. V. and Bennett, J. L. (1990) "Learning from user experience with groupware." In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '90)*, Los Angeles, California, October 7-10, ACM Press.
4. De Cindio, F., De Michelis, F., Simone, C., Vassallo, R. and Zanaboni, A. (1986) "CHAOS as a coordination technology." In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '86)*, pp. 325-342, Austin, Texas, December 3-5, ACM Press.
5. Dewan, P. and Choudhary, R. (1991) "Flexible user interface coupling in collaborative systems." In *ACM SIGCHI Conference on Human Factors in Computing Systems*, pp. 41-48, New Orleans, April 28-May 2, ACM Press.
6. Greenberg, S. (1990) "Sharing views and interactions with single-user applications." In *Proceedings of the ACM/IEEE Conference on Office Information Systems*, pp. 227-237, Cambridge, Massachusets, April 25-27.
7. Greenberg, S. (1991) "Personalizable groupware: Accomodating individual roles and group differences." In *Proceedings of the European Conference of Computer Supported Cooperative Work (ECSCW '91)*,

pp. 17-32, Amsterdam, September 24-27, Kluwer Academic Press.

8. Grudin, J. (1988) "Why CSCW applications fail: Problems in the design and evaluation of organizational interfaces." In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '88) (CSCW '88)*, pp. 85-93, Portland, Oregon, September 26-28, ACM Press.
9. Johnson-Lenz, P. and Johnson-Lenz, T. (1991) "Post-mechanistic groupware primitives: rhythms, boundaries and containers." *Int J Man Machine Studies*, **34**(3), pp. 385-418, March.
10. Kaplan, S. M., Tolone, W. J., Bogia, D. P. and Bignoli, C. (1992) "Flexible, active support for collaborative work with ConversationBuilder." In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, pp. 378-385, Toronto, Ontario, October 31 - November 4, ACM Press.
11. Leland, M. D. P., Fish, R. S. and Kraut, R. E. (1988) "Collaborative document production using Quilt." In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '88)*, pp. 206-215, Portland, Oregon, September 26-28, ACM Press.
12. Linton, M. A., Calder, P. R. and Vlissides, J. M. (1988) "InterViews: A C++ Graphical Interface Toolkit." Research Report CSL-TR-88-358, Stanford University.
13. Malone, T. W., Grant, K. R., Turbak, F. A., Brobst, S. A. and Cohen, M. D. (1987) "Intelligent information-sharing systems." *Comm ACM*, **30**(5), pp. 390-402, May.
14. Malone, T. W., Lai, K. Y. and Fry, C. (1992) "Experiments with Oval: A radically tailorable tool for cooperative work." In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, pp. 289-297, Toronto, Ontario, October 31 - November 4, ACM Press.
15. Markus, M. L. and Connolly, T. (1990) "Why CSCW applications fail: Problems in the adoption of interdependent work tools." In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '90)*, Los Angeles, California, October 7-10, ACM Press.
16. Nunamaker, J. F., Dennis, A. R., Valacich, J. S., Vogel, D. R. and George, J. F. (1991) "Electronic meeting systems to support group work." *Comm ACM*, **34**(7), pp. 40-61, July.
17. Ousterhout, J. K. (1990) "Tcl: An Embeddable Command Language." In *Proceedings of the 1990 Winter USENIX Conference.*
18. Root, W. R. (1988) "Design of a multi-media vehicle for social browsing." In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '88)*, pp. 25-38, Portland, Oregon, September 26-28, ACM Press.
19. Roseman, M. and Greenberg, S. (1992) "GroupKit: A groupware toolkit for building real-time conferencing applications." In *ACM Conference on Computer Supported Cooperative Work (CSCW '92)*, Toronto, Ontario, November 1-4, ACM Press.
20. Shen, H. and Dewan, P. (1992) "Access Control for collaborative environments." In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, pp. 51-58, Toronto, Ontario, October 31 - November 4, ACM Press.
21. Shepherd, A., Mayer, N. and Kuchinsky, A. (1990) "Strudel -- An extensible electronic conversation toolkit." In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '90)*, Los Angeles, California, October 7-10, ACM Press.
22. Stefik, M., Bobrow, D. G., Foster, G., Lanning, S. and Tatar, D. (1987) "WYSIWIS revised: Early experiences with multiuser interfaces." *ACM Trans Office Information Systems*, **5**(2), pp. 147-167, April. An earlier version appeared in CSCW '86.