Notification Servers for Synchronous Groupware

John F. Patterson
Lotus Development Corporation
55 Cambridge Parkway
Cambridge, MA 02142-1295
+1 617 693 4236
john_patterson@crd.lotus.com

Mark Day
Lotus Development Corporation
55 Cambridge Parkway
Cambridge, MA 02142-1295
+1 617 693 0576
mark_day@crd.lotus.com

Jakov Kucan
M.I.T. Laboratory for Computer Science
545 Technology Square
Rm. NE43-334
Cambridge, MA 02139
+1 617 253 5866
kucan@theory.lcs.mit.edu

ABSTRACT
We introduce the Notification Service Transfer Protocol (NSTP), which provides a simple, common service for sharing state in synchronous multi-user applications. A Notification Server provides items of shared state to a collection of clients and notifies the clients whenever one of the items changes. The division between client and server in this system is unusual; the centralized state is uninterpreted by the server. Instead, the responsibility for semantics and processing falls on the clients, which collude to implement the application. After describing NSTP, we differentiate it from other systems in terms of the four design principles that have guided its development.

Keywords
Synchronous groupware, multi-user applications, group-ware infrastructure, client/server architectures, notification, protocol, design principles, performance, state sharing.

INTRODUCTION

The problem of maintaining consistent state is central in network-based synchronous multi-user groupware. In each such system, the user interfaces (UIs) of the participants must be kept consistent to promote the impression of playing or working together. For example, players in multi-user games like Doom must be aware of each other's presence, as must visitors in shared virtual spaces like AlphaWorld. Users of shared editors must be able to see changes and gestures made by coeditors, as must users of shared whiteboards. Greif [7] and Baecker [3] have useful background readings on such synchronous groupware.

Often, the technique for maintaining consistency is specific to the application being implemented. Effectively, each new synchronous groupware application (re)invents some mechanism for achieving consistency. Our approach, the Notification Service Transfer Protocol (NSTP), abstracts out the problem of state consistency from any particular application. It provides a simple, general, open, centralized service to support the construction of synchronous groupware.

In this paper, we describe NSTP and discuss the design principles that differentiate it from similar systems. Although NSTP is the collection of messages by which a client obtains service from a Notification Server, we do not discuss detailed message formats for NSTP. Instead, we present a high-level view of how a Notification Server looks via NSTP. This is essentially the conceptual model that the programmer of a client needs to implement an application; end users do not need to know anything about NSTP.

A simplified view of the service supported by NSTP is presented in Figure 1. In this toy example, the server has five "pieces" of state, represented by the different shapes. Only two values of the state are possible: filled or unfilled. When Client1 sends a message to the server indicating that the circle should be filled, the server changes the shared state appropriately and then sends each of the clients a

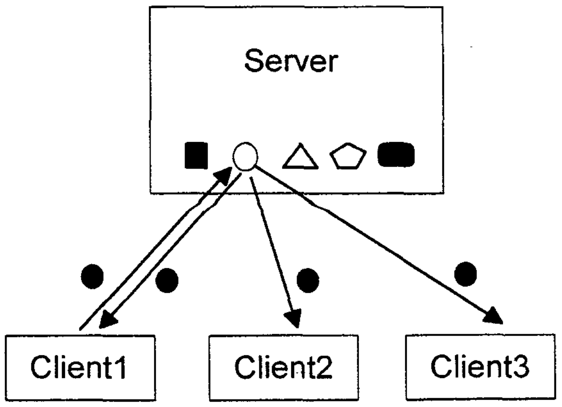

Figure 1: Schematic of a Simplified Notification Server.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copy-right notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
Computer Supported Cooperative Work '96 , Cambridge MA USA
© 1996 ACM 0-89791-765-0/96/11 ..$3.50

notification of the state change. The notifications permit the clients to maintain a consistent account of a common fact, namely the state of the circle.

Although we believe NSTP does well at solving certain problems of reuse and performance, NSTP does not attempt to solve all problems of synchronous multi-user groupware. There are two areas that we explicitly do not address: persistent state and streaming traffic.

Persistent state outlives the execution of an application, generally by making a record on a nonvolatile medium. Asynchronous groupware requires sharing via persistent state, since the users cannot rely on being simultaneously present during a communication. Synchronous groupware often requires some of both persistent and non-persistent (ephemeral) state. For example, sharing via persistent state may be best for any large, while sharing via ephemeral state is appropriate for echoing shared pointers during the discussion. A Notification Server is designed to be useful for ephemeral state, not persistent state[1]. Insofar as an application requires persistence, some other infrastructure will be required. Some examples of such infrastructure are a distributed database, a Web server, Lotus Notes[2],or a distributed file system.

Streaming traffic is data such as video and audio. Conferencing applications usually transmit continuous streams of information among user stations. Other applications, like multi-user editors and many multi-user games, only transmit brief bursts of information when some element of the application changes and other clients need to be informed. Notification servers are useful for such bursty traffic and less stringent real-time demands, but they are not suitable for routing streaming traffic among clients. If an application needs data streams for audio or video between users, then some additional infrastructure will be required.

In summary, NSTP allows clients to communicate with a centralized mechanism for maintaining consistent state among those clients. It is not intended as a means either for accessing persistent state or for streaming information among clients.

This paper explores the design of NSTP by examining its four major design principles, namely:

- Support for Awareness,
- Centralized State Sharing,
- Client-based Semantics, and
- Place Browsing.

Each principle is explained, justified, and compared to the design of other groupware infrastructure. At the end, we ask how NSTP is distinctive and consider extensions for improving NSTP. First, however, we begin with a brief description of the conceptual model underlying NSTP.

CONCEPTUAL MODEL OF NSTP

The conceptual model of NSTP is a description of the service from the vantage point of a developer who must implement a client. Unlike some services (notably Internet Relay Chat [9]), NSTP is not intended as an end-user protocol. Client software uses NSTP to obtain service on behalf of a user. The purpose of a Notification Server is to share state information among a set of clients that work together to implement a synchronous multi-user application. The user's experience of the application is determined by the clients, not NSTP.

We would like NSTP to be simple, generally applicable, and efficient. Two senses of efficient compete here. On the one hand, the time required to receive a state change request, make the change, and send notifications to all the clients must be minimized. On the other hand, it is inefficient to devote a process or processor to each new multi-user application. The tension is between supporting as many applications as possible and optimizing the notification performance for each application. Since this tradeoff is demanding enough, we have resisted the temptation to add features to NSTP unless they facilitate the management of state and the ensuing notifications. As a result, NSTP is somewhat austere.

NSTP deals with four kinds of object: Places, Things, Facades, and the server itself. Figure 2 illustrates the relationships among these entities. Within the server, the larger box with the dark outline corresponds to a Place. The thicker outline is intended to imply that these objects are opaque until the client becomes associated with ("enters") them. The geometric forms within the Place are the Things. The box with the thinner outline in front of the Place is its Facade. The geometric forms within the Facade are views of the Things from the Facade. The thin outline of the Facade is intended to convey that it is accessible by any client associated with the server.

A Place is the most important kind of object. It contains the shared state and partitions the server's resources among multiple applications. Each application uses at least one Place within which its shared state is kept. Clients join the application by entering the Place. Once they are in the Place, they receive notifications about any change to the

[1]There is actually nothing in NSTP that prevents a server implementation from storing state in files before generating notifications; we simply assume that such implementations will be avoided because performance is a primary objective. There is also nothing that prevents a server from backing up state during idle moments as a hedge against system crashes; this would be prudent. Clients of Notification Servers cannot rely on these forms of persistence, however, and must behave accordingly.
[2]Lotus Notes is a registered trademark of Lotus Development Corporation.

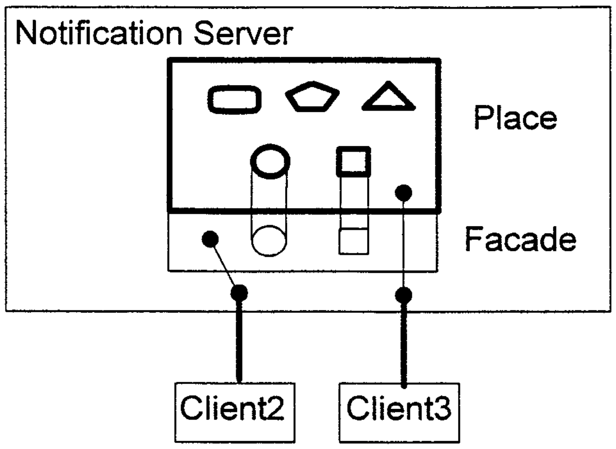

Figure 2: A Place, its Facade, and some Things

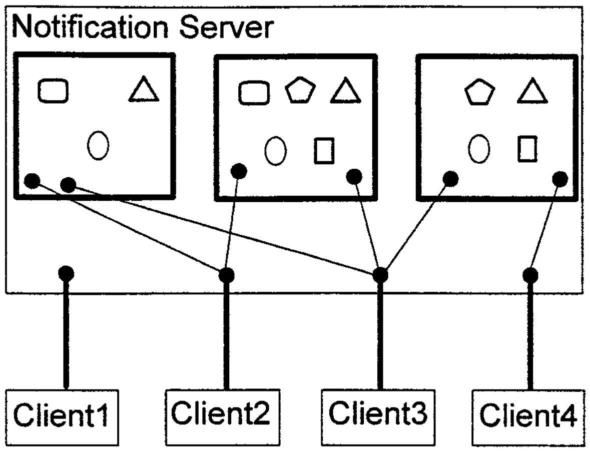

Figure 3: Each client can be in zero, one, or more Places (Facades omitted for clarity)

state of Things in that Place; they may also make changes of their own. Once the client leaves a Place, it no longer receives notifications from that Place. In Figure 3, Client1 has not entered any Places even though it has registered with the server. The other clients are all in one or more of the three Places. Note that Client2 and Client3 are in more than one Place at a time; entering a new Place does not require leaving the current Place.

Things are the keeper of state within the Places. They have three parts: name, value, and attributes. Disregarding attributes momentarily, the Things are essentially property-value pairs for the Place. The name may be any string; the value may be any byte array. The attributes control access to the Things and indicate the type of notifications that a Thing will cause. Attributes are set at creation, and may not be modified thereafter. Since only the value of an object can change, not its attributes, we avoid the problem of whether attribute changes should cause notifications. Things may be created, changed (value only), locked, unlocked, and deleted. If specified by the attributes, Things will cause notifications on creation, change, or deletion.

A Facade is the external view of a Place. It comes into being when the Place is created and is destroyed when the Place is destroyed. Facades serve two purposes. First, they provide access to any Things with attributes indicating that they are publicly accessible. In Figure 2, the Place has two publicly available Things, the square and the circle. Distinct attributes control reading and writing of a Thing, so a Thing can be publicly readable without being publicly writeable or vice-versa. The other purpose of a Facade is to provide entry to the Place.

Finally, the server itself provides some services to the client. It authenticates clients and ensures that they should be allowed to use the server. Also, it provides the means to create new Places and retrieve existing Places.

SUPPORT FOR AWARENESS

In this section we start to explain and justify the principles of our design. Our first principle is that the server must support an application class we refer to as "awareness". The Notification Server grew out of earlier work on a service that made it possible for users of Lotus Notes databases to know who else had the same database(s) open at the same time. This service was built around a server where clients registered whenever they opened or closed a Notes database. Once registered, the clients received change notifications for entering or departing clients.

Other systems that provide similar functionality use the same technique, but often use different terminology. Rendezvous[3] [10], Touring Machine [2], and GroupKit [11] use the term session; LambdaMOO [4] and more recently Jupiter [5] [8] have Places; Internet Relay Chat (IRC) [9] has channels. We use the term *session* as a general term for the mechanism that identifies and binds the distributed elements of an application into a collection of cooperating processes for a period of time.

While many different session models are possible, the two most common involve either "calling-out" or "calling-in". A standard phone call and some conference calls exemplify the calling-out model, in which conference participants are brought into the conference by calling out to them from the conference. A "meet me" telephone conference exemplifies the calling-in model, in which participants call into the session to indicate their interest in joining.

[3]Rendezvous is a trademark of Bellcore.

Another perspective on this distinction between calling-out and calling-in is the type of network addressing that is involved. In the calling-out model, clients are the entities that may be addressed; the session is implicit. In the calling-in model, sessions are the addressable entities. In the world of groupware, some systems, e.g. ProShare[4],support calling-out, while others, e.g., Internet Relay Chat, MUDs, and AlphaWorld, support calling-in.

Awareness applications work best with a calling-in model with explicitly addressable sessions. If only clients are addressable, then awareness can only be achieved by having clients notify one another directly. For large numbers of clients, this will only work if one is willing to devote the bandwidth needed for the large number of broadcast notification messages. In general, the number of notifications required in such a system grows as $O(n^2)$. The more efficient choice is to use a session server as a clearinghouse or filtering point for matching client notifications with those clients that are interested in receiving the notification.

CENTRALIZED STATE SHARING

A system that only supports awareness (as described in the previous section) is essentially a session server that permits clients to join a session and then receive notifications whenever any other clients join or depart. Such a server facilitates state consistency among its clients, but the only kind of state is the presence or absence of a client in the session. We next asked whether this might be usefully generalized to support client-defined shared state. From earlier experiences with Rendezvous [10], we knew that many synchronous multi-user applications (e.g., games, whiteboards, etc.) have only modest throughput requirements. It seemed reasonable to centralize the shared state.

We should point out that this was not an obvious choice. GroupKit [11], for example, has introduced a session server, called the Registrar, to support awareness, but avoids encumbering it with anything other than the simple awareness requirements. Once the clients learn of each other's presence via the Registrar, further communications are pursued on a client-to-client basis. The GroupKit designers chose to centralize only where centralization was truly required.

So why have we chosen to centralize these interclient interactions? Why not simply have each client send its messages to all other clients in the session?

Figure 4 illustrates our reasoning on this point. In the first approach to interclient communication (A), each client sends messages to all other clients. The designers of

[4]ProShare is a trademark of the Intel Corporation.

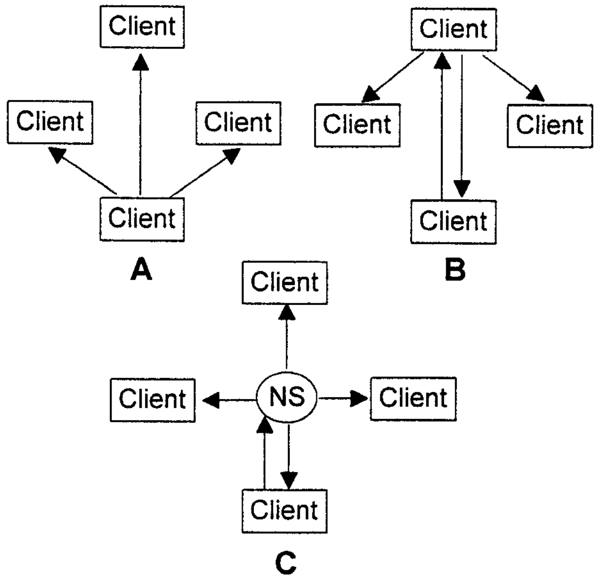

Figure 4: Three approaches to interclient interaction

GroupKit do this whenever possible, but they concede that it is sometimes necessary to ensure that client requests are serialized, i.e., placed in a consistent order. When this is needed, one client (the upper one in approach B) assumes the role of serialization point. It receives all the messages and redistributes them (even to the original sender). Finally, in approach C, we see the Notification Server acting as this central serialization point, not the client.

Our primary reason for preferring a central serialization point is that clients are typically less reliable than servers. A client is usually thought of as supporting one person's work, while a server provides services for a number of people. Because a server failure can affect more people, a sensible manager will devote more resources to keeping servers up. In addition, the pragmatics of machine placement mean that servers are often in machine rooms and relatively well-protected; clients are in the offices of the users, where they are subject to being powered down, unplugged, having coffee spilled in them, and the like.

These observations about client reliability are critical to deciding which functions to centralize. In approach B, should the process or machine fail, or should the user simply decide to power down, then the other clients are either stuck or a rather complicated recovery protocol must ensue. With a server-based serialization point, like approach C, clients are free to be unreliable without disrupting the other clients' access to the application. Furthermore, more attention can be devoted (e.g., better hardware, more reliable code, failover protection) to ensuring that this critical bottleneck does not fail.

Arguably, the difference between GroupKit's approach and the Notification Server approach is a judgment about how often serialization is required. Over all the interclient messages, if serialization is rarely required, then GroupKit accepts little risk and avoids the unnecessary delays associated with serializing messages that do not require it. If serialization is frequently needed or if clients fail often, then a Notification Server will be simpler and more reliable. It is perfectly reasonable, however, to use both, writing clients that use a Notification Server when serialization is required and that communicate directly when it is not.

Another system that supports awareness, but does not support centralized state is Internet Relay Chat (IRC) [9]. IRC does go further than GroupKit by providing multicast messaging via its servers, but it does not have a concept of shared state and notifications about that state. Unlike GroupKit, the absence of centralized shared state in IRC was less an aversion to centralization than an indifference to state sharing as a concern.

So why do we centralize state? The need for a central serialization point does not imply a need for keeping state centrally. *Zephyr* [6], for example, provides a service similar to ours without keeping any client-defined state at the servers; instead, the servers accept messages from clients and multicast those messages back out to interested clients. We offer two advantages for centralized state: it simplifies lock-keeping and it is useful for introducing latecomers to an application.

Whenever clients are permitted to change the same state variables or collection of variables, it may be necessary to ensure that only one is doing it at any given time. To accomplish this, some form of locking is needed. Assume there are several lockable tokens, but there is no centralized keeper of locks. Then every client must respond to queries from other clients about which clients have which tokens. For any tokens they possess, they must process requests to yield the token, send the token as needed, and notify all relevant clients that the token has been passed. In contrast, when there is a centralized lock keeper, clients do not process lock requests and there is no need for clients to keep track of where the locks are being held. Nichols, et.al. [8] make similar arguments for the manner in which centralized processing simplified their concurrency control.

Our other reason for preferring centralized state is that it eases the problem of introducing a latecomer to a synchronous application. If all the essential state in a shared application is kept in the server, then the client that comes late can determine what it needs to know without encumbering any other client. When there is no such server, as in GroupKit, then some client must take on the role of communicating the current state to the latecomer. If this client is not the serialization point, then various problems of intermixing initial state messages with change notifications are possible. If it is the serialization point, then this client is even more encumbered.

One point of view about our decision to put shared state in a centralized server is that we are exploring which services are best centralized. We have already concluded that awareness will compel some centralization. Given that we will have a session server, we are trying to determine what else it can usefully do. Clearly, some activities should be done on a client-to-client basis, for example file transfer, but we are unwilling to go along with GroupKit's avoidance of any additional centralization. In this regard, we are more in line with LambdaMOO [4] and Worlds, Inc's VRML+ [12].

CLIENT-BASED SEMANTICS

Two of our primary design objectives were to ensure that Notification Servers were application-independent and that they would perform well. We were aiming for a "lean, mean, notification machine" that could be used in a wide variety of synchronous groupware applications. The best way to achieve these objectives was to avoid the temptation to have the server "know" anything about the semantics of its Places. *Zephyr* [6] pursues a similar design strategy, but does not provide a state sharing service.

The reasoning behind this client-based semantics is quite simple. In aggregate, clients have more available computing resources than a server. Thus, the more that processing can be shifted off the server and into the clients, the less the server will need to do and the better its overall performance will be.

This decision to keep the application semantics out of the server had two identifiable consequences for the protocol. First, the protocol includes no mechanisms for programming a Place. There is no way, for example, to inform the server that if a certain value is set then certain activities should follow. This type of conditional processing might be nice to have, but NSTP leaves it to the clients to perform.

Similarly, NSTP has no concept of different types of state. When we began designing, it seemed obvious to some of us that we would want strings, integers, enumerated types, and most standard types of variables. It became clear, however, that we were beginning to encumber Notification Servers with the need to process a language as well as issue notifications. Retreating from this approach, we opted for a single undifferentiated data type, a byte array. The encoding and decoding of these byte arrays is left entirely to the clients. The server does nothing but move the data around.

While we feel that client-based semantics is an essential element for ensuring the scalability of Notification Servers, other systems have emphasized different concerns. MUDs, in general, and LambdaMOO [4], in particular, are similar to Notification Servers in that they support sessions, called

Places, and notifications about changes to the state within a Place, but have made a notable commitment to server-based semantics. LambdaMOO provides a programming language with which users may create object behaviors. The programs are stored and processed on the server. This permits them to be shared among LambdaMOO's users.

With a Notification Server one would support programmable objects in a different manner. Clients would provide a UI for writing programs and would store the programs in a shared location. Then some state in the Place would be modified to indicate the location of the new program so that all clients could retrieve it. As needed, clients would retrieve the program and process it locally. This demonstrates that client-based semantics does not really prevent Notification Servers from doing what LambdaMOO can do; it just changes the location at which it is done.

NSTP's client-based semantics is one of its most striking characteristics. It is assumed that different clients will be needed for different applications. A whiteboard application will require one type of client, a chess game another. The collection of clients that enter a Place must work together to determine how the Place is used. In a sense, the clients collude in the interpretation of a Place.

Because NSTP assumes cooperating clients, it is open to certain abuses that are less likely with LambdaMOO. For example, one could write a generic client that could enter any Place permitting one to eavesdrop on or vandalize the Place. To inhibit this we have provided for:

- user authentication when registering with a server,
- mechanisms for controlling who may enter a Place,
- access control on a Place's state,
- and a policy that sends notifications to all clients within a Place whenever any other client either enters or leaves the Place.

Moreover, eavesdropping on or vandalizing a Place would probably be rather unrewarding, since one would only have access to uninterpreted byte arrays, which could themselves be encrypted. Nevertheless, the mechanisms that we provide are only helpful if applications are written to use them, which means that abuses remain feasible. When the semantics are server-based there are more opportunities to either provide or impose protective mechanisms.

Worlds, Inc's VRML+ [12] reveals another situation in which server-based semantics are in use. VRML+ is designed to support a large scale multi-user virtual reality. Many users are expected to be simultaneously browsing through 3D virtual worlds and they must all receive notifications about one another's movements. This is accomplished with a server that regularly sends notifications to its clients about any other clients that are within a certain radius.

VRML+ segments its notifications in a very different way from NSTP. NSTP imposes the rather harsh boundary of Places, which clients may either be in or not; VRML+ has a notion of a three-dimensional space and determines one's notification set as a sphere around one's location. While this approach is not very general, it is more efficient for very large Places and does demonstrate an advantage to shifting some semantics into the server.

PLACE BROWSING

Our final design objective for NSTP was that it should be possible to browse from one Place to another. For example, having entered a Casino Place where all sorts of games are available, it should be possible to see and then enter Places to play each of the different games. If the Places for playing particular games are thought of as tables, then sitting down at the poker table (entering the poker Place) need not mean that one leaves the Casino. One's client might be prepared to be in two or more Places at once.

Alternatively, the Casino might have doors that represent opportunities to leave the Casino and go to other interesting Places. These might be other arbitrarily different Places, much like the "see also" hyperlinks on a Web page.

This ability to browse among Places is an essential element of both LambdaMOO [4] and MERL's Diamond Park [1], but they do not support browsing to arbitrary Places. Our objective with NSTP was to ensure that links between Places could be as rich as hypertext links.

The first two steps in achieving this goal are directly analogous to HTTP. First, we borrowed the concept of a URL as the standard form of addressing for Places. This ensured that one Place could refer to another Place anywhere on the Internet. Second, we recommend that certain Thing names be reserved as indications of other Places to access from the current Place.

This permits a client to discover a link in one Place that leads to another and even to follow that link to another server. The real difficulty comes in knowing how to enter the Place. With client-based semantics each Place may need a different client for its interpretation. Clearly, no client can implement the interpretation for every possible Place. If it does not, how can we ensure that any arbitrary Place may be linked to any other arbitrary Place?

Our solution to this seeming conundrum is to permit Places to indicate where an executable may be retrieved for interpreting the Place. We provide a mechanism whereby a browser may determine the type of a Place before entering. This serves the same purpose as a MIME type in HTTP. We also provide a mechanism whereby a Place browser could obtain a URL for a Java application that could enter and interpret the Place. Similarly, a Place could indicate where to find an OCX for interpreting it.

The mechanism for achieving this trick is a Place's Facade. The Facade permits certain information about a Place to be made publicly available so that a client may read the information without actually entering the Place. Thus, one can imagine a general-purpose Place browser that examines a Place's Facade, finds the URL for the Java code to interpret the Place, downloads the code, and then upon executing the code enters the Place. In this fashion, different applications can use Places in very different ways without foreclosing the ability of users to browse among the Places.

CONCLUDING COMMENTS

We have presented a brief description of the Notification Service Transfer Protocol (NSTP) and explained four basic design principles that have guided this work. We conclude by characterizing NSTP's niche in the world of groupware infrastructure and presenting some areas for future improvement.

One way of classifying alternative groupware infrastructure is in terms of the amount of service that they migrate into a session server. We can begin with systems that do not use a session server and then consider ones that use a session server and progressively introduce an awareness service, shared state, and then semantics. This yields the following classification with examples:

- No Server
  - ProShare
- Server: awareness only
  - Rendezvous
  - Touring Machine
  - IRC
  - GroupKit
- Server: awareness, and shared state, but no semantics
  - NSTP
- Server: awareness, shared state, and application semantics
  - LambdaMOO/Jupiter
  - VRML+

The most distinctive aspect of the NSTP design is that it provides server-based state without server-based semantics. Unlike GroupKit, NSTP does not shy away from centralizing state; unlike LambdaMOO, the semantics associated with this state remains the concern of the client, not the server. We are trying to strike a balance between centralizing some critical services, e.g., serialization, lock-keeping, and latecomer support, while ensuring that a Notification Server can provide low-latency notifications for many simultaneously active applications.

Three extensions of NSTP currently under consideration are:

- multicast messages,
- predefined Places, and
- server-launched agents.

Multicast messages allow clients to send messages to other clients within a Place without actually changing any state on the server. Initially, our design did not include this capability because we were concentrating on the shared state. We expect, however, that without some ability to send multicast messages, clients will use shared state for multicasting when there is no need to record the message, even ephemerally. This is inefficient. With multicast, we believe we will improve overall efficiency by eliminating an inefficient practice.

Predefined Places and server-launched agents are mechanisms for allowing the server to impose more structure on Places so that clients can trust them more. Predefined Places are templates for well known Places that a server is prepared to create. The client obtains the Place by specifying its type in the creation request. Then the server creates the Place and populates it with the Things that such a Place requires. For these types of Places, any client can be assured that certain Things will be available and initialized with the appropriate attributes; the clients need not trust that some other client has set everything up correctly. Since NSTP currently requires that Places be prepopulated with certain state, predefined Places are merely a generalization of this capability and should not unduly harm performance. (They could actually improve it.)

Server-launched agents are a mechanism for providing the appearance of server-based semantics without actually encumbering the Notification Server. For some predefined Places, it should be possible to indicate that the server should launch a special client, called an agent, when the Place is created. The server could ensure that the agent is always the first client to enter the Place and the last to leave. This client could then offer some seemingly server-based processing for the Place.

Although server-launched agents might seem to violate our approach of client-based semantics, a server need not launch agents on its own processor. It can communicate with a special agent server on another processor and thereby prevent the agents from disrupting notification performance. In addition, since agents are simply clients associated with predefined Places, we have not changed the protocol to make them possible. Notification Servers that support applications that do not need server-based semantics are not encumbered by the fact that other Notification Servers use the capability.

As we look at the future of NSTP, the challenge will remain the same. We must maintain a balance between providing more service and preserving the efficiency of Notification Servers. We think the best way to achieve the "lean, mean notification machine" is to remain committed to client-based semantics and avoid encumbering the server with functionality that is application-specific. Our strategy has been to add functionality to the server only when we can persuade ourselves that it is:

- a generally valuable capability,
- needed in the server, and
- not a burden on applications that choose not to use it.

We expect this strategy to continue.

WHERE TO GET MORE INFORMATION

More information about NSTP may be obtained at our web site, www.research.lotus.com, including a complete specification of the protocol and a reference implementation of the server (written in Java.)

REFERENCES

1. Anderson, D.B., Barrus, J.W., Howard, J.H., Rich, C., Shen, C., and Waters, R.C. "Building Multiuser Interactive Multimedia Environments at MERL" *IEEE Multimedia* 2, 4, 1995, pp. 77-82.

2. Arango, M., Bahler, L., Bates, P., Cochinwala, M., Cohrs, D., Fish, R., Gopal, G., Griffeth, N., Herman, G.E., Hickey, T., Lee, K.C., Leland, W.E., Lowery, C., Mak, V., Patterson, J., Ruston, L., Segal, M., Sekar, R.C., Vecchi, M.P., Weinrib, A., and Wuu, S-Y. "The Touring Machine." *Communications of the ACM* 36, 1, 1993, pp 68-77.

3. Baecker, R.M. (Ed) *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration* Morgan Kaufmann Publishers, San Mateo, CA 1993.

4. Curtis, P. *LambdaMOO Programmer's Manual*, Xerox, August 1993. The most recent version is available as ftp://ftp.parc.xerox.com/pub/MOO/ProgrammersManual.ps.

5. Curtis, P., Dixon, M., Frederick, R., and Nichols, D.A. "The Jupiter Audio/Video Architecture: Secure Multimedia in Network Places." *Proceedings of ACM Multimedia '95*, ACM Press, 1995.

6. DellaFera, C.A. and Eichin, M.W. "The *Zephyr* Notification Service." *Proceedings of the USENIX Winter Conference*, Dallas TX: USENIX Association, 1988.

7. Greif, I. (Ed) *Computer-Supported Cooperative Work: A Book of Readings* Morgan Kaufmann Publishers, San Mateo, CA, 1988.

8. Nichols, D.A., Curtis, P., Dixon, M., and Lamping, J. "High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System." *Proceedings of UIST '95*, Pittsburgh, PA: ACM Press, 1995, pp. 111-120.

9. Oikarinen, J., and Reed, D. *Internet Relay Chat Protocol* RFC 1459, May 1993, Available at ftp://ds.internic.net/rfc/rfc1459.txt.

10. Patterson, J.F., Hill, R.D., Rohall, S.L., and Meeks, W.S. "Rendezvous: An Architecture for Synchronous Multi-User Applications." *CSCW '90 Proceedings*, Los Angeles, CA: ACM, 1990.

11. Roseman, M. and Greenberg, S. "Building Real Time Groupware with GroupKit, A Groupware Toolkit." *ACM Transactions on Computer Human Interaction*, ACM Press, Scheduled for March 1996.

12. Steremberg, A., Close, J., Hayes, K., and Mitra *VRML+: Technical Documentation* March, 1996, The most recent version is available at http://www/worlds.net/vrml/technical/.