RC23756 (W0510-174) October 21, 2005
Computer Science

Architectural Trade-Offs for Collaboration Services Supporting Contextual Collaboration

Roberto S. Silva Filho[1], Werner Geyer[2], Beth Brownholtz[2], Ido Guy[3], David F. Redmiles[1], David R. Millen[2]

[1]Department of Informatics
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697

[2]IBM Research Division
One Rogers Street
Cambridge, MA 02142

[3]IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Architectural Trade-Offs for Collaboration Services Supporting Contextual Collaboration

Roberto S. Silva Filho[1], Werner Geyer[2], Beth Brownholtz[2],
Ido Guy[3], David F. Redmiles[1], David R Millen[2]

[1] Department of Informatics
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA, 92697 USA
{rsilvafi, redmiles}@ics.uci.edu

[2] IBM T.J. Watson Research Center
One Rogers Street
Cambridge, MA 02142, USA
{werner.geyer, beth_brownholtz, david_r_millen}@us.ibm.com

[3] IBM Research Haifa Labs
University Campus
Carmel Mountains
Haifa 31905, Israel
ido@il.ibm.com

ABSTRACT
Contextual collaboration services seamlessly integrate existing groupware technologies into a single consistent interaction model. This model is usually implemented by a contextual collaboration infrastructure that needs to efficiently cope with the fast switching between different modes of interaction. This paper introduces a new model for contextual collaboration based on the notion of generic shared objects. We describe a native implementation of this model and evaluate its behavior under different traffic conditions. We also explore an alternative implementation of the collaboration model integrating existing notification and meeting servers to deliver the same model behavior. We discuss trade-offs and limitations of those two implementations.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *Client/server*, C.4 [**Performance of Systems**]: *Design studies*

General Terms
Performance, Design, Experimentation, Human Factors.

Keywords
Collaboration, architecture, activity-centric, synchronous / asynchronous collaboration, performance, scalability, responsiveness.

1. INTRODUCTION
Collaboration is moving away from dedicated tools or specialized solutions, to highly integrated, contextualized user experiences. Contextual collaboration promises new levels of productivity by the seamless integration of collaboration channels, shared artifacts, communication, and collaboration tools under a common interaction paradigm. For example, through the integration of communication channels and office tools, users can easily switch between individual and collaborative work through a single click of a button. In other words, they can start a chat from within their document editors, share a document on their desktops by dragging it on their buddy lists, start a remote presentation by right-clicking on a presentation file on their desktop, or send the content of a spreadsheet by email from within the document itself. Contextual collaboration lowers the barrier to engage in collaboration by transparently integrating existing groupware technologies in a single consistent interaction model. By doing so, it reduces end user's cognitive cost of switching between applications, by providing a single point of access to a set of inter-related applications and the artifacts they produce.

In our research, we are particularly interested in the infrastructural implications of contextual collaboration. A highly contextualized user experience entails frequent changes in work mode and modalities. For example, imagine the following scenario: Alice starts work on a presentation by creating a few slides and an outline. She then shares her original slides with Bob in order to get some feedback. From within the document Alice adds Bob as a member to her document and adds the document to a shared workspace. When Bob becomes available online, he is notified about the new presentation Alice shared with him. When Bob opens the presentation and starts reviewing it, a status indicator in the document alerts Alice that Bob is currently active on the document. Seizing the opportunity, Alice starts a chat from within the open document by clicking on Bob's name in her buddy list. After a few turns in the chat, they open an audio channel and a real-time screen sharing session from within the chat to discuss the document in more detail.

In this scenario, Alice moves seamlessly from an individual work mode, to an asynchronous sharing mode, and then to real-time collaboration all from within the context of the document. In order to implement this scenario, several backend collaboration services are usually required: A document repository to share the document, an event notification service to push notifications to users, and a real-time conferencing/meeting service for chat, audio, and screen sharing support. Ideally, when delivering this user experience, the use of these infrastructural components should be transparent to the end user. Hence, they need to be integrated in smart ways to deliver their services in conjunction with each other. Integrating existing collaboration services agnostic of the user experience poses many challenges including but not limited to data consistency, scalability, or service responsiveness. An alternative solution, however, is to create a new collaboration service that merges all those interaction modes into a common model that provides such contextualized experience.

In this paper, we present a new model for contextual collaboration based on generic shared objects that work as building blocks for supporting contextual collaboration applications. We describe our experiences with a native implementation of this interaction model and study its behavior with different interaction patterns. We also describe an integrated implementation (where existing services such as meeting and notification servers are used), delivering the same contextual user experience as in the native implementation. Our work is in the context of a special instance of a contextual collaboration called activity-centric collaboration [1]. In our previous works we have discussed architecture [2, 3], usability and usefulness [1], as well as patterns of media use [4] involving an application called Activity Explorer, build on top of our model. Activity Explorer introduced a new collaboration paradigm by integrating collaboration technologies horizontally through the concept of a work activity (a logical unit of work that incorporates all the tools and resources to get the job done). In this paper, our goal is to study the contextual collaboration model behind Activity Explorer, understanding its scalability, responsiveness and implementation trade-offs.

This paper is organized as follows. Section 2 describes the contextual user experience in Activity Explorer in more detail. In Section 3, we describe the contextual collaboration model behind Activity Explorer. We use this model as the basis for our study. Section 4 describes the native implementation of this model. Section 5 describes our simulation environment, the experiments performed, as well as the simulation results for the native implementation. In Section 6 we describe an alternative integrated implementation combining existing collaboration technologies and we discuss simulation results and trade-offs between the two solutions. We close with related work and a summary of this work.

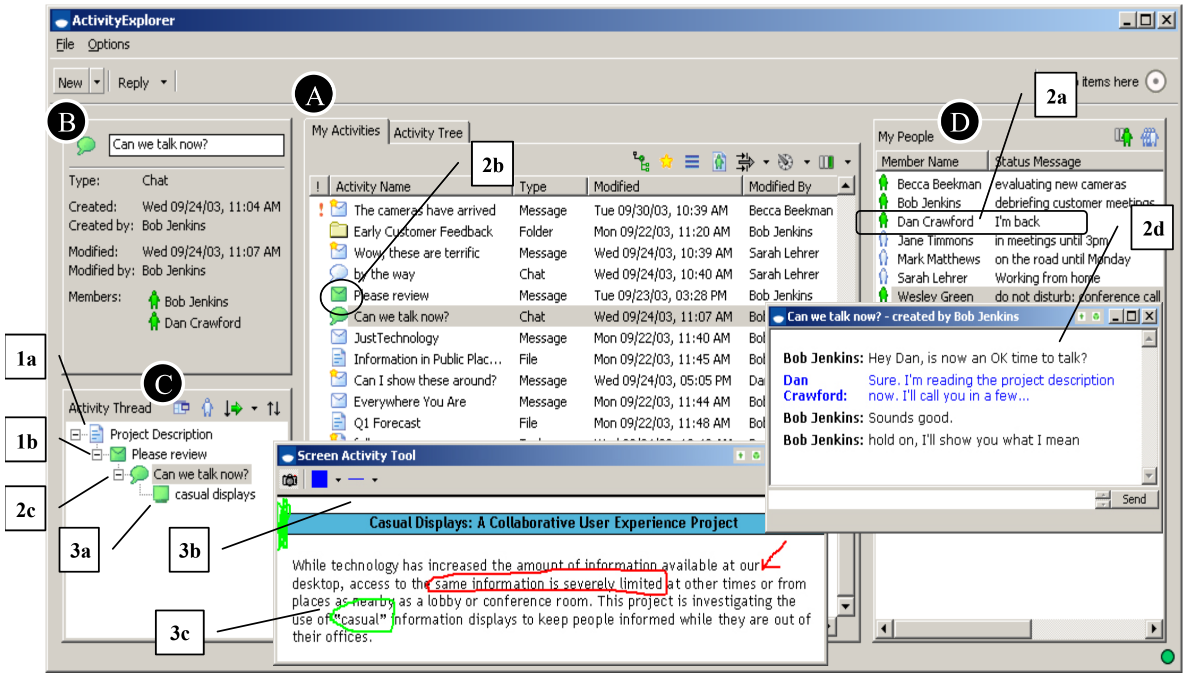
2. ACTIVITY EXPLORER

Activity Explorer (AE) is a contextual collaboration application based in the concept of activities. It is build upon the concept of generic shared objects supported in our contextual collaboration model. It is presented here as an application that illustrates the of our model in supporting contextual collaboration.

In AE, an activity represents an interaction involving one or more users, which context (membership and persistent content) is made persistent thought the use of shared objects. An activity thread is a set of related, shared objects that can be hierarchically composed, representing a complex task or project. Hence, activity threads combine different types of persistent objects, membership, and alerts. Users start new activity threads by creating root objects from any type of content or communication. The AE research prototype supports sharing of six types of persistent objects: message, chat transcript, file, folder, annotated screen shot, and to-do item. Users add items to an activity thread by posting either a response or a resource addition to its parent object.

As presented in Figure 1, in AE, the activity structure and membership are managed by several UI components: My Activities (A) is a multi column "inbox-like" activity list, supporting sorting and filtering. Selecting a shared object in this list populates a read-only info pane (B). The Activity Thread pane (C), maps a shared object as a node in a tree representing an entire "activity thread." Activity Thread and My Activities are synchronized by object selection. My People (D) is a buddy list showing all members the current user shares activities with. Users interact with objects or members, as displayed in these views, through right-click context menus. Representative icons are highlighted green to cue users of shared object access and member presence (2a, 2b).

The following scenario illustrates a contextual user experience in which shared objects are used in a collaborative context, as part of an activity. The activity starts from a document:



Can we talk now?

Type: Chat
Created: Wed 09/24/03, 11:04 AM
Created by: Bob Jenkins
Modified: Wed 09/24/03, 11:07 AM
Modified by: Bob Jenkins
Members: Bob Jenkins, Dan Crawford

!	Activity Name	Type	Modified	Modified By
!	The cameras have arrived	Message	Tue 09/30/03, 10:39 AM	Becca Beekman
	Early Customer Feedback	Folder	Mon 09/22/03, 11:20 AM	Bob Jenkins
	Wow, these are terrific	Message	Wed 09/24/03, 10:39 AM	Sarah Lehrer
	by the way	Chat	Wed 09/24/03, 10:40 AM	Sarah Lehrer
	Please review	Message	Tue 09/23/03, 03:28 PM	Bob Jenkins
	Can we talk now?	Chat	Wed 09/24/03, 11:07 AM	Bol
	JustTechnology	Message	Mon 09/22/03, 11:40 AM	Bol
	Information in Public Plac…	File	Mon 09/22/03, 11:45 AM	Bol
	Can I show these around?	Message	Wed 09/24/03, 05:05 PM	Da
	Everywhere You Are	Message	Mon 09/22/03, 11:44 AM	Bol
	Q1 Forecast	File	Mon 09/22/03, 11:48 AM	Bol

Member Name	Status Message
Becca Beekman	evaluating new cameras
Bob Jenkins	debriefing customer meetings
Dan Crawford	I'm back
Jane Timmons	in meetings until 3pm
Mark Matthews	on the road until Monday
Sarah Lehrer	Working from home
Wesley Green	do not disturb: conference call

Project Description
- Please review
  - Can we talk now?
    - casual displays

Can we talk now? - created by Bob Jenkins

Bob Jenkins: Hey Dan, is now an OK time to talk?
Dan Crawford: Sure. I'm reading the project description now. I'll call you in a few…
Bob Jenkins: Sounds good.
Bob Jenkins: hold on, I'll show you what I mean

Casual Displays: A Collaborative User Experience Project

While technology has increased the amount of information available at our desktop, access to the same information is severely limited at other times or from places as nearby as a lobby or conference room. This project is investigating the use of "casual" information displays to keep people informed while they are out of their offices.

Figure 1. Activity Explorer User Interface

Bob and Dan are working on a project (a file) using Activity Explorer. Bob right clicks on the file object in his list to add a message asking Dan for his comments (1b). A few hours later, Dan returns to his desktop (2a). In the system tray, Dan is alerted to *the new activity. Clicking on the alert, he is taken to the activity thread. He opens the message and while he is reading it, Bob can see Dan is looking at the message because the shared object is lit green (2b). Bob seizes the opportunity to expedite their progress; he right clicks on the initial message and adds a chat to this activity (2c). A chat window pops up on Dan's desktop and they start a chat session (2d). Bob refers to a detail in the project description; for clarity he wants to show Dan what he would like changed. By right clicking on the chat object, Bob creates a shared screen object (3a). A transparent window allows Bob to select and "screen scrape" any region on his desktop. He freezes the transparent window over the project text. The screen shot pops up on Dan's desktop (3b). Bob and Dan begin annotating the web content in real-time like a shared whiteboard (3c).*

3. COLLABORATION MODEL

The contextual collaboration server for Activity Explorer implements a simple and generic collaboration model based on the concept of *Generic Shared Objects (GSO)* introduced at [3]. GSOs are persistent collaboration objects that can be used as building blocks to create new collaborative applications. They support the implementation of a seamless user experience with blended synchronous and asynchronous collaboration. GSOs combine different communication styles, data representations, notifications, and membership control policies into objects that can be hierarchically composed. This section describes in detail the main aspects of the GSO model with which our experiments were performed.

3.1 Communication

In this paper, we assume a client/server architecture in which many clients interact with each other through a collaboration server (or service) implementing the concept of GSOs. Note that the model can be also implemented in a peer-to-peer fashion as described in [3].

The communication protocol is based on three basic primitives: *Request*, *Response*, and *Notification*: A client asks for some service by issuing a Request to the server. The server replies with a Response to inform the requesting client about the result of its request.[1] Depending on the type of request, the server also sends out Notifications to other connected clients (see Figure 2).

3.2 Data Model

The data model allows sharing of any type of content based on the notion of GSOs. Our contextual collaboration service manages a collection of GSOs and their relationships, i.e. by containment and/or reference. This facilitates the aggregation of GSOs into hierarchical structures, thus modeling complex collaborations such as the previously mentioned activity threads. Each GSO holds one or more pieces of persistent information and defines a list of members (actual users) permitted to access its information. Each GSO provides a simple content model that defines a set of general properties and a set of variable properties for content. General properties of GSOs include unique id, name, author, creation time, modifier, modification time, reader, last access time, member list, and member status pertinent to the GSO. The variable properties of a GSO describe its actual content. A GSO does not provide any means for semantically describing the content. Content is associated with a GSO by adding arbitrary numbers of <name, value> pairs (or tuples). The interpretation and use of those <name, value> pairs is left to client applications. The value field can be of various types, e.g. String, Integer, Double, Boolean. In particular, a GSO supports binary content, allowing the storage of arbitrary content.

[1] Note that, depending on the implementation, request and response might be represented through a single operation, e.g. when using remote procedure calls.

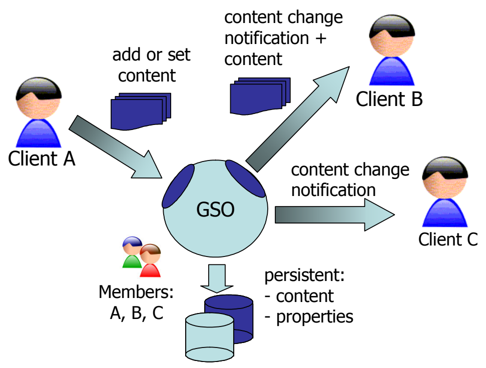


Figure 2. Generic Shared Object behavior

3.3 Membership

Each GSO also manages a list of members (e.g. A, B, and C in Figure 2). The GSO member list controls the access to its content and represents a distribution list for broadcasting notifications about the creation and modifications of a GSO. The member list is dynamic and allows adding new members or removing existing members at runtime. Since the member list is also a property of the GSO, any modification to the member list is also broadcast to all online GSO members. As noted earlier, our collaboration service supports more complex collaboration through the aggregation of GSOs into hierarchical structures. Each GSO within such a structure can have different membership supporting fine-grained access control policies within those hierarchies. The contextual collaboration service may provide convenience functions to help manage the membership within these structures; e.g. when adding to or removing members from a single GSO, the server might provide options to propagate membership to other related GSOs.

3.4 Notifications

Each GSO can be considered a persistent conferencing session between its members who are currently online. Any modification to the set of fixed properties or the set of variable properties (content) of a GSO is not only stored in the underlying data store but also automatically broadcast to all the other members of that GSO by means of notifications. The default behavior is that every modification to a GSO is broadcast to all its members. Using publish/subscribe vernacular, whenever a member is added to an object, it is implicitly subscribed to all change events of that object.

Notifications on content (property) come in two different flavors: They can only indicate that content was changed, or they can transport the actual new/modified content. This is controlled by the use of *open* and *close* semantics. Change notifications (without the actual content) are sent to all online members of the object whose open status of this object is false; whereas notifications with the actual content are sent to all online members whose open status of this object is true. Setting open to true basically subscribes a member to receive the content together with the content change notification. These semantics are important because they make sure that members of an object, who are not interested in collaborating on the object at the same time (real-time), are not flooded with unnecessary data. They only receive low volume content change notifications.

In the example in Figure 2, clients A, B, and C are all members of the same GSO. Hence, whenever client A changes a content property in the GSO (through a set property Request), clients A and C, which are online, receive notifications of that change. Client C's open status on the object is false, i.e. it receives only notification of changes. Client B previously set open to true and receives the entire new / modified content.

The collaboration service not only sends notifications about modifications to GSOs but also about the creation of new GSOs and the deletion of existing GSOs. Those notifications are sent to all online members of these objects. Since the state of a GSO is persistent, GSO properties are still available when clients disconnect and later reconnect to the service. This allows clients to interact asynchronously. In summary, the described behavior of GSOs inherently merges real-time conferencing with content management and asynchronous collaboration modes.

3.5 Example: Activity Explorer

The Activity Explorer (AE) research prototype, described in Section 2, is a stand-alone desktop application that connects to a contextual collaboration server implementing the model described in the previous sections. AE makes use of this collaboration model in the following way: each Shared Object in AE is represented by a GSO; the activity thread in AE is represented through a hierarchical structure of GSOs; the different shared object types use the property fields of a GSO to manage their content. The persistent chat object in AE, for example, works as follows: Each chat message is stored as a property. In order to send a chat message, the client (representing a member) adds a variable property to the GSO, representing another line in the chat, by submitting a request to the collaboration service. The collaboration service adds and stores the property and then sends the new property to all members of this GSO. The members of the chat receive this property change and add the chat message at the end of their transcript. Note only members whose open statuses are true will receive the actual chat message; all others will receive only notification that content was changed. This example illustrates that our collaboration model assumes a different paradigm for real-time collaboration. It is based on persistent state and state change notifications. Other objects in AE also map their content in a similar way: For example, the *shared screen objects* store background images and strokes as properties of the GSOs, the *shared message objects* store the content of the message as single properties, and so forth.

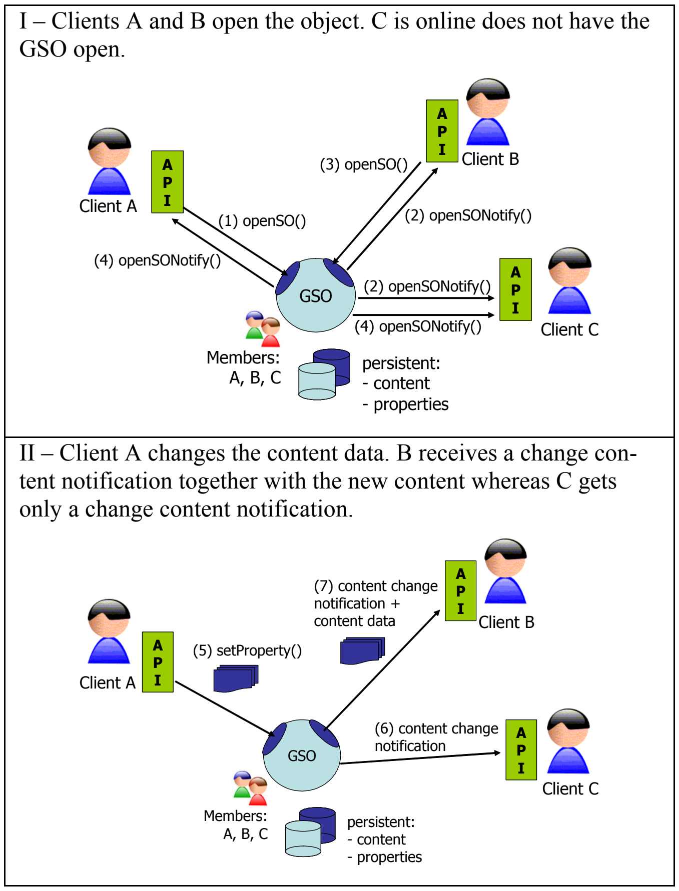
Note that while AE exposes an almost direct mapping to this model, the collaboration service does not assume any particular use of its GSOs. Hence, GSOs can be used in various ways as building blocks for other collaborative application. Jazz [5] and C&BSeen [6] are two example applications that implement contextual collaboration by using GSOs in a different way than AE.

4. NATIVE IMPLEMENTATION

In order to study and better understand the implications of combining various interaction modes of collaboration into a single model, we have built a reference (or native) implementation of a collaboration server implementing the new collaboration model from scratch, i.e. without using/integrating existing collaboration services.

In our native implementation, the GSO concept is mapped to persistent objects (using the OO programming paradigm). The implementation of the GSO manages every aspect of the model, i.e. property management, membership management, access control, notifications, and data persistency. The GSO service manages a collection of GSOs and their aggregation into hierarchical structures (trees).

Clients access the GSO service through a client side GSO API See Figure 3). The GSO objects in the server manage the notification process and data transfer with the clients.


I – Clients A and B open the object. C is online does not have the GSO open.

II – Client A changes the content data. B receives a change content notification together with the new content whereas C gets only a change content notification.

Figure 3. Native implementation of the GSO model

In the example of Figure 3, clients A, B, and C are all members of a GSO object. In (I) client A and B open the object for real-time interaction by submitting an *openSO()* requests to the server (1, 3). The server GSO then sends open notifications to all its members, by iterating over the member list and invoking the registered callback interface methods (2, 4). The open state of the GSO is now changed to true for clients A and B. Sending notifications to every member of the GSO keeps all connected clients in a consistent state (i.e. with the latest view of the GSOs they are members of). Client C, for example, knows that A and B are working on the GSO content in real-time. Based on this information, client C can decide to open the GSO object and start receiving the actual new content as it gets changed. In (II), client A changes the content of the GSO (add or set) by submitting a *setProperty()* request (5); client B receives a content change notification including the content data (7). Client C is online but receives only a content change notification without the data because its open state is false (6). However, knowing that the content has changed, Client C could now read the updated content of the object by submitting a *getContent()* request to the server.

The server is implemented in Java and communicates via Remote Method Invocation (RMI) with its clients. Notifications are sent to clients through RMI too. Upon logon, each client registers an RMI callback interface with the server. Since we assume storage to be a constant throughout this paper, we did not implement a particular storage mechanism in our native implementation.

5. SIMULATION

The model described in Section 3 unifies characteristics of publish/subscribe systems, real-time collaboration servers, and content management. As such, it facilitates the development of collaborative applications that have contextual collaboration characteristics that typically require the blending of synchronous and asynchronous interaction models.

We assume that this blending of synchronous and asynchronous communication in a single model requires the compromising of different requirements from these two interaction modalities. For example, traditional synchronous communication infrastructures, such as meeting servers, are usually designed to support the collaboration of a small group of members, under more strict timing and bandwidth conditions such as audio or video. Notification servers, on the other hand, generally are employed in applications with less strict timing and synchronous constraints, such as applications for awareness and messaging, where the number of clients is potentially large and the data traffic is relatively small. Combining various interaction modes in one collaboration model raises interesting questions regarding scalability, delays, responsiveness, robustness, and implementation complexity.

5.1 Experimental Setup

We used our native implementation of the model to evaluate its response to different traffic conditions. In order to understand the behavior of the model under regular use conditions, we developed a “user simulator” client program. The user simulator interacts with the server API performing regular actions such as create new object, set properties, open, close, add member and so forth. Actions are executed according to predefined media patterns. We defined four different patterns approximating the traffic conditions of chat, file sharing, message exchange, and streaming media. The streaming media pattern was especially defined to analyze the server behavior under heavier load, testing its scalability limits. Note that these patterns are only approximations of actual real world patterns. Table 1 describes the programming of parameters and probabilities of each media pattern. Parameters are randomly varied.

Table 1: Media pattern programming

Media Pattern	n° Mem-bers	Data			Content change probabilities		
		Size (chars)	n° msg	interval	Set	Add	Del
Streaming	5	64K	100	50 ms	0.5	0.5	0.0
Chat	2	40	10	15 sec	0.0	1.0	0.0
File Sharing	4	100K	10	5 min	0.7	0.1	0.2
Message Exchange	8	1K	1	1 sec	1.0	0.0	0.0

The main differences between the four media traffic patterns are the size of the data messages, the number of messages exchanged by each member, and the frequency (defined by the interval between messages). For example, a typical chat session in our simulator corresponds to an interaction with a GSO with two members on average exchanging an average of 10 messages each member. Each message has an average length of 40 characters. Each chat GSO also has an average of seven properties that are modified with 16 characters on average. Chat message are send with a frequency of 15 seconds on average. During this interaction, periods of inactivity may occur with an average of 15 seconds.

In our GSO implementation, a property can be set (overwritten or created), added (appended to the end of the current content), or deleted. Table 1 shows the probabilities for these content change actions. In the chat pattern, for example, all chat content changes are of type “Add” because chat transcripts are typically not randomly modified, but grow over time as new messages are exchanged.

For each pattern, we reproduce the actions of a typical work day with 8 hours. For easier handling, we simulated each workday in 4 minutes. During one simulated workday, the following actions take place on average: A total of 15 shared objects are created with five objects being root objects (representing a new activity thread). Each client listens to an average number of 10 objects. 15 open and 15 closed objects on average are modified that day. The simulated patterns also differ in respect to the time span that each client is working either online or offline.

Unless otherwise noted, all our simulations are carried out on three client machines (IBM T30, 1.6GHz, 512MB) and one server machine (IBM MPro, 3 GHz, 1.5 GB). Client machines are equally loaded with simulator clients in steps of one, i.e. the first simulations starts with 3 clients (one in each client machine), then 6 clients (two per client machine) and so forth. Please note the number of simulator clients running on a single client machine impacts the overall simulation results. Based on tests, we decided to limit the number of simulator clients to eight per client machine in order to minimize this effect.

5.2 Simulation Results

In order to understand the response of the system to the different media patterns described in Table 1, we plotted the total average execution times of the four patterns against the number of clients as shown in Figure 4.

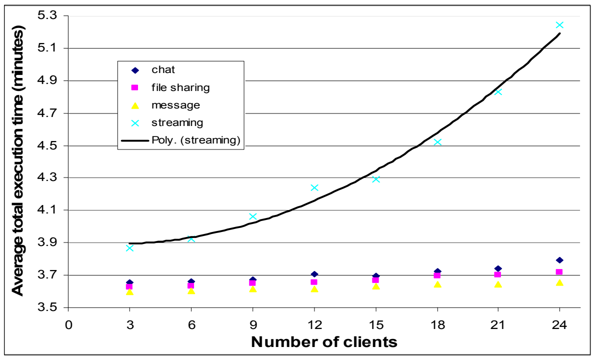


Figure 4. Average total simulation execution times of the native implementation under different activity patterns

The system scales well for low-frequency traffic such as chat, message exchange, and file sharing. The size of the data, as in the case of file sharing does not impact performance as much as the frequency of the messages. Streaming media has a high frequency of relatively large data messages. As expected, our reference implementation does not scale very well for this pattern. One reason is related to the behavior of the model. Since each GSO in our model comes with data persistency, we currently send out content change notification (without or without the content) to every member of the GSO. Given the high data frequency of streaming media, (each data message triggers a series of content change notifications, typically one for each member of the objects involved), the server becomes pretty busy with sending out notifications.

The responsiveness of a collaborative system describes how fast the system reacts to user input, i.e. how fast actions are reflected in the user interface of the client executing the action. Responsiveness in our model is determined by the execution time of the client API calls. Figure 5 shows the average method execution times for setting the content property of a GSO.

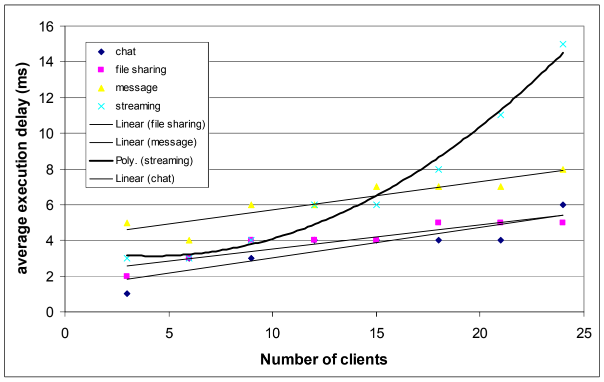


Figure 5. Average execution time of the *setContent()* call in the native implementation with different media patterns

The execution times are low and grow linearly except for the streaming media pattern. For a low number of clients (and consequently a lower number of method calls on the server), the streaming media pattern is comparable to the other patterns but, as the number of method calls increases with the number of clients, this pattern grows quadratically. Interestingly, the message pattern initially has higher execution times than streaming media. The reason is the higher number of eight members on average.

The notification time is the time from calling a method in the client API to the delivery of its notification to the other members of a GSO. The notification time basically describes how fast a collaborative system updates remote clients. In a collaborative setting, it is desirable to keep this number as low as possible because it expresses how much clients are in sync with their collaborators. Figure 6 shows the average execution time versus notification delays for creating new GSOs.

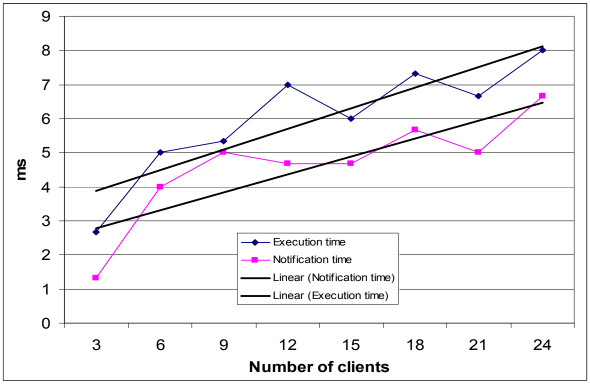


Figure 6. Average execution vs. notification time for creating new GSOs in the native implementation

Notification times are slightly lower than execution times. At an almost constant difference of about 1 ms (in the trend lines) each local user interaction is visible in remote clients at about the same time. Except for streaming media *setContent()* calls, the responsiveness of the native implementation is relatively high (below 10ms) and the notification delays are extremely low.

As a general conclusion, the performance of the model is a function of the data frequency of the interaction pattern (number of data messages/second), and the number of members of a GSO. For general traffic the system scales very well having good responsiveness and optimal notification delays. However, for streaming media traffic, with a relatively medium number of members, the system delays increase quadratically.

6. INTEGRATED IMPLEMENTATION

Pursuant on better understanding design implications, implementation complexity, and performance trade-offs of using traditional collaboration services to implement a contextual user experience as described by our model, versus the native implementation of the same model, we have implemented an alternative collaboration server. Our alternative implementation integrates existing technologies in the backend but provides exactly the same GSO behavior at the GSO API level to its clients.

The results in the previous section indicate that our native implementation is not very well suited to handle high bandwidth, high frequency data. Existing real-time collaboration servers, however, are optimized for online meetings with a smaller number of participants but relatively high data volume, e.g. audio, video. Hence, real-time collaboration servers would very well support frequent and high volume property changes in a GSO. It seemed that supporting the real-time aspects of our model with a meeting server would increase the overall system performance. Consequently, we decided to use a meeting server for handling real-time content changes of a GSO, i.e. whenever members decide to work on an object at the same time.

Notifications are another aspect of our model that we believed to be well understood today. Publish/subscribe systems have been around for a while. They are typically optimized for a very large number of subscribers and small to medium data volumes for each subscriber. GSO events such as create, delete, add member, remove member, or infrequent property changes (e.g. changing the presence status of a member on an object) would be very well supported by a publish/subscribe system. Publish/subscribe system are general purpose distributed implementations of the observer design pattern [7], usually implemented in the form of notification servers. Those servers receive anonymous notifications and route them to interested parties. This routing is orchestrated by subscriptions: queries on the content or order of notifications published to the server.

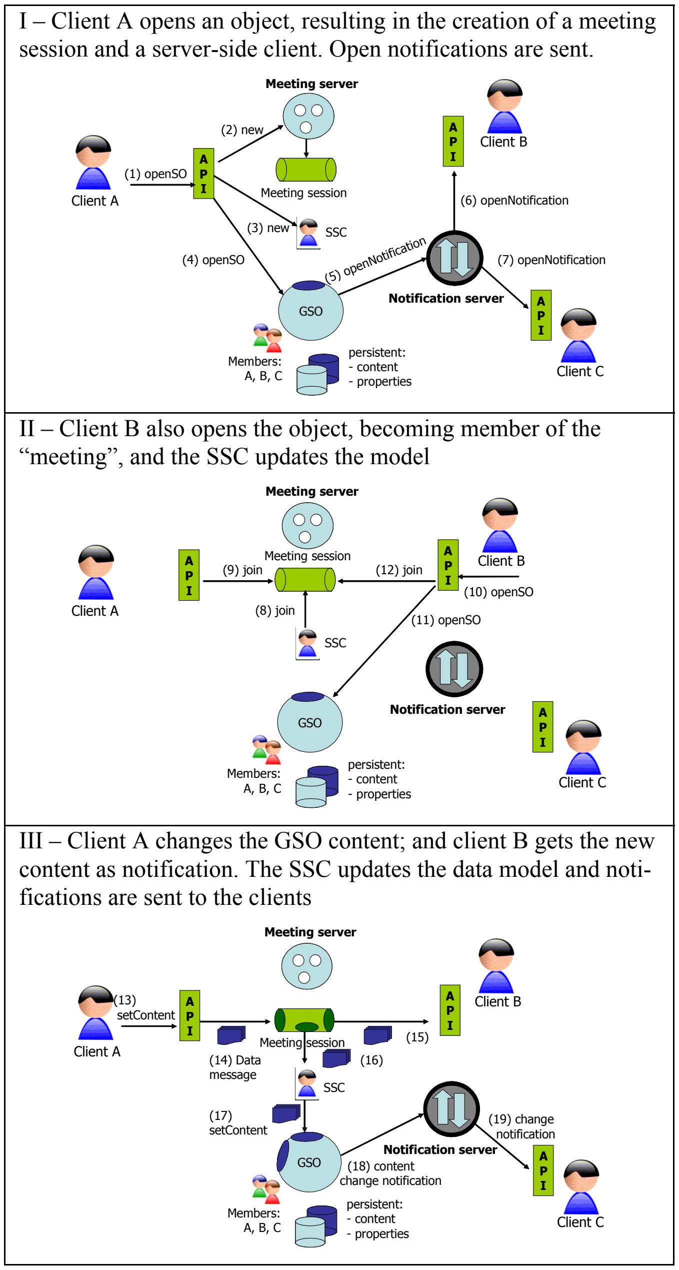
We expected that integrating these two technologies to our model, would result in better scalability of both the notification process (asynchronous mode in our model), and the real-time collaboration through content exchange (the synchronous mode of our model). However, in return we expected an extra cost in terms of complexity of the architecture and a higher toll on execution times for some API calls and notification times.

6.1 Implementation Details

The integration of the two new backend technologies is completely transparent to the clients (see Figure 7). They interact as usual through the client API. In the backend, however, the implementation complexity increased. For example, in order to integrate the meeting server with our model, we introduced the concept of a server-side client (SSC) that acts as a connector between the real-time meeting and the persistent aspects of the model. A SSC is a special client in the meeting (a meeting is a session created between two or more participants/clients). It provides a non-persistent shared space where messages are multicast to all the meeting members. The SSC is responsible for persisting session data and for updating the model when content has changed through the meeting, e.g., when a chat message is posted to a meeting session, the SSC for that session stores the message in the GSO. This approach provides a generic mechanism that can be used to integrate any meeting server implementation in a way agnostic to our GSO server.

Whenever a GSO's properties and content change it produces a single notification (and not one for each member as before) that is sent using a notification server. We decided to keep the implementation simple: Each client subscribes/un-subscribes to a global GSO notification topic when logging on and off. In this approach, the notification server acts as a broadcast channel; a bus connecting all online clients. Notifications are subsequently filtered in the client side API, i.e. the client API ignores notifications that are not addressed to that particular client.

As illustrated in Figure 7 (I) when client A first opens the GSO (1), a new meeting session (2) together with a hidden SSC (3) are created. The GSO object is also open. Consequently, an open notification is sent to all clients (5, 6, 7) through the notification server. The SSC joins the meeting (8) and listens to messages in that channel. The *openSO()* call returns the meeting id to client A. Upon receiving the meeting id, client A also joins the meeting and is ready to transmit data. Client B decides to open the GSO as well and submits an *openSO()* request to the GSO server (10) (for simplicity notifications are omitted in the picture). The open call is propagated to the GSO object (11), which returns the existing meeting id. Client B also automatically joins the meeting (12). As content messages are exchanged between members A and B and the SSC (14, 15, 16), the SSC makes the content persistent by invoking *setContent()* on the GSO (17). The GSO server contacts the notification server to deliver content change notifications (without content) to the other members of the object who are not in the open state (18, 19).


I – Client A opens an object, resulting in the creation of a meeting session and a server-side client. Open notifications are sent.

II – Client B also opens the object, becoming member of the "meeting", and the SSC updates the model

III – Client A changes the GSO content; and client B gets the new content as notification. The SSC updates the data model and notifications are sent to the clients

Figure 7. Integrated implementation of the GSO model

The integrated solution is also completely implemented in Java. We used YANCEES [8], an extensible and configurable event service, as the notification server. This server was chosen because of its ability to be configured with a simple topic based core, and for having a simple API, similar to Elvin [9]. YANCEES, written in Java, was previously developed by one of the authors of this paper. We also used a simple Java-based meeting server developed by one of the authors. The meeting server was previously used in the TeamSpace project [10].

For the integrated implementation we carefully modified our core native implementation where required. Both implementations share the same common GSO model and externalize the same GSO API to the end users (clients). However, where necessary, we adjusted the code to integrate the meeting server and the notification server. For example, we intercept some client API calls in order to send messages to the meeting server (instead of setting the properties directly in the GSO). The implementation of the notification interfaces send messages through the notification server, instead of notifying the clients directly.

We tried to keep the two implementations as similar as possible in order to get meaningful results for a comparison. However, given the number of different existing publish/subscribe and real-time collaboration systems, the results may vary depending on the backend technologies used.

6.2 Simulation Results

As described in previous sessions, our native implementation did not expose good scalability properties for streaming media traffic patterns. In this session, we want to study the response of the alternative integrated implementation to this kind of traffic. Figure 8 compares the cost of the set/add content calls in both implementations.

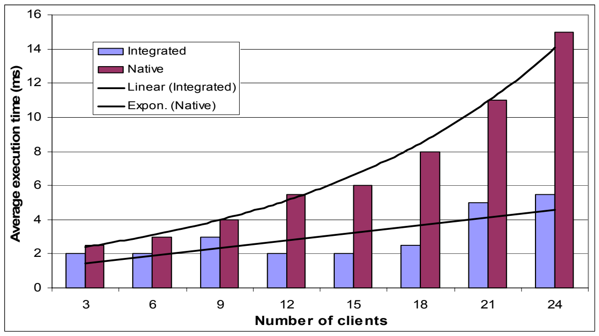


Figure 8. Comparison of the average execution times for *setContent()* calls for the stream media pattern

As expected, the integrated implementation scales well in a linear fashion compared to the native implementation. Using a dedicated meeting server seems to pay off for this type of traffic.

The chat and the file sharing media patterns did not expose any significant differences in the integrated implementation with regards to the cost of the *setContent()* call. However, the message exchange pattern yielded some interesting results. Figure 9 shows that the use of our meeting server is more costly than the native implementation for this pattern. Both implementations though seem to expose linear behavior as indicated by the trend lines. One of the major differences between the message exchange pattern and the other patterns is the number of members per GSO, which in the message pattern case are 8 on average. While our meeting server seems to handle high bandwidth, high frequency traffic well, performance seems to degrade with an increased number of meeting participants.

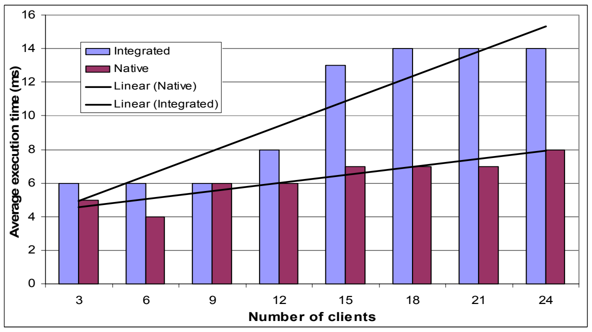


Figure 9. Comparison of the average execution times for *setContent()* calls for the message exchange pattern

Using a meeting server introduces additional complexity as described in Section 6.1. We expected that the price for better scalability during the real-time phase of a GSO would be additional delays in getting started. The data in Figure 10 compares the cost for opening GSOs in both implementations. The data confirms that the open call has become one of the most costly calls in the integrated implementation. However, it still scales in a linear fashion indicated by the trend line.

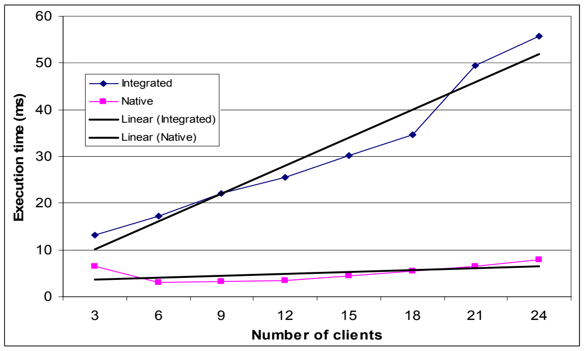


Figure 10. Comparison of the average execution times for *openSO()* calls

Table 2 shows a summary of the average execution times of GSO API calls. We can see that not only the open call stands out relative to other calls in the integrated implementation but also *registerMember()* and *loginMember()*. The reason for such relatively high numbers is our notification server. Creating subscriptions when registering members and when logging in comes at an additional expense. Note that subscriptions in our native implementation were implicit through the member list. Another interesting observation in Table 2 is that the execution times of most calls in the integrated implementation are generally higher. The code executed is the same for most API calls (except for open and close, which create SSC objects, and for set and add content, that route data through the meeting server). We can only explain this as a consequence of higher load on the server machine imposed by three server processes running at the same time, in the same host (GSO server, meeting server, notification server).

While we expected that subscription management would come at an extra cost, we were surprised to see that the notification server introduced high delays in delivering notifications. Figure 11 compares execution times for creating GSOs against the notification time.

Table 2. Average cost of GSO API calls (24 clients)

GSO API Call	Native	Integrated
getIds	9	24
addMember	4	17
getContent	6	13
open	6	**52**
create	8	19
setProperty	6	13
close	5	14
logout	9	20
registerMember	**178**	**1699**
setContent	7	10
getGSOs	43	68
addContent	6	6
login	**17**	**130**

The integrated implementation has a high responsiveness given the low and linear execution times but does not scale well with regards to notifications. On average, under a load of 24 clients, remote clients are updated only 0.5 second after the GSO was created locally. The notification times seem to grow exponentially according to the trend line.

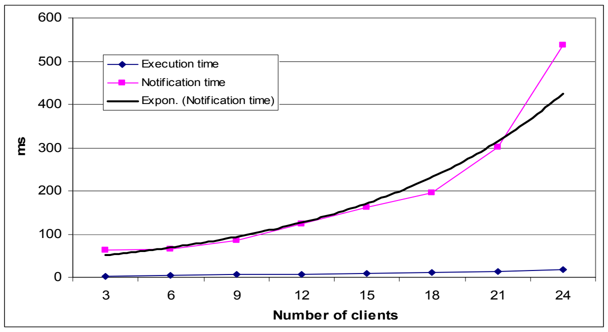


Figure 11: Average responsiveness vs. notification delay for creating new GSOs in the integrated implementation

One could argue that the use of the notification server as a shared bus is one of the reasons for the notification server behavior observed in Figure 11. In another alternative implementation, we opted to perform server-side filtering of events, i.e. the configuration of the notification server with more accurate subscriptions that filter out events that are not of interest of the client. This approach, however, required constant update of the subscriptions (each client manages one or more subscriptions filtering out events that do not belong to the objects they are members of). As new objects are created, members logged off or got removed/added to different objects, the subscription needs to be updated. Given the high subscription costs presented in Table 2 (part of *registerMember()* and *login()* operations), this solution did not scale. These membership and object life-cycle dynamics resulted in similar or worse delays than the ones observed in Figure 11.

7. LESSONS LEARNED AND DISCUSSION

Our data indicates that the use of a simpler native implementation scales well for the majority of traffic patters we propose except for high frequency, high-bandwidth data. The overall responsiveness of the model, including its notification delays and API execution times, for the number of clients we tested, looks very good.

The use of a specialized meeting server in the way proposed in our integrated implementation can improve the overall response of the system for frequent synchronous mode traffic patterns such as streaming media. However, when applied to more regular traffic such as messages or chat, it does not improve the overall performance of the system. Its use comes with extra integration complexity and startup costs such as for the creation of the session and the server-side clients.

The use of the publish/subscribe model, provided by notification servers, did not meet our original expectations. During the tests, the notification server was the most CPU demanding component, more than the meeting server was, and the notification traffic did not scale linearly. We tested the implementation with two subscription models: server-side filtering and client-side filtering. Client-side filtering was the approach that better scaled in our implementation. Both approaches have trade-offs and limitations: client-side filtering requires the delivery of extra notifications through the network but makes the routing process easier (topic-based). Server-side filtering limits the amount of traffic to the clients and relief them from discarding uninterested notifications. This approach, however, loads the notification server that needs to handle with more complex subscriptions and constant re-subscriptions issued by the clients to reflect their new membership condition.

Hence, as a general conclusion, the use of notification servers (distributed publish/subscribe implementations) were not a good choice for this particular problem. The GSO model requires a simple observer model, as implemented in the native solution, and not a fully distributed publish/subscribe model, whose adaptation to our model introduces more overhead.

One advantage of using separate components such as a meeting server and a notification server, however, is the ability to distribute those servers throughout other hosts in a network. In our tests comparing a distributed server (notification server in one host and meeting server in another machine versus all together in the same host), did not show any significant improvements up to 27 clients. With more than 30 clients, however, the distributed configuration begins to perform better than a single server does, indicating that with a significant number of clients, this approach may be an option for scalability.

Overall, with respect to the implementation complexity, the native implementation is more simple and straightforward; it has less synchronization and integration problems and scales well. The integrated implementation, however, is more complex and demands special attention to matters such as timing and synchronization, it is more prone to implementation failures and incurs in higher startup times, including delays associated to member log-in and opening objects. In addition, as shown by our experience with the notification server, delays associated to one or another component can negatively impact the whole system performance.

8. RELATED WORK

The GSO characteristics of persistency, change notifications, and membership are also found in other collaboration infrastructures. Those systems alone, however, do not provide all the combined characteristics of the GSO model.

The Tuple Space concept, originally proposed by Gelernter as part of the Linda coordination language [11], and currently implemented by systems such as IBM's TSpaces [12] and SUN's JavaSpaces [13], provides a persistent and shared memory (or space), accessed through an API that allows distributed processes to read, write, and remove information represented as tuples (type, attribute, value pairs). Tuples can be concurrently read or removed from the space by different processes. In this programming paradigm, concurrency and interoperability mechanisms can be easily implemented, as well as more advanced communication and coordination mechanisms such as distributed queues and locks. Even tough very powerful, this model does not provide concepts such as group, membership or object hierarchies.

Notification servers, as defined by Patterson et al. [14], provide a simple common service for sharing state in synchronous multi-user applications. They address the problem of maintaining consistency in real-time collaborative applications and supporting awareness. They are similar to tuple spaces with regard to the addition of specialized services for managing the event space and for supporting different notification policies required to improve all sorts of activity awareness. Membership, hierarchy of information, and persistency of the data are not covered by that work.

Event notification servers such as Elvin [9] are usually employed as event routing infrastructure to support the development of awareness applications. Elvin provides a relatively simple but optimized set of functionalities, efficiently processing large quantities of events based on content-based routing of tuple-based events. In such systems, however, event persistency is not usually addressed moreover, as previously discussed, they usually are not designed to support synchronous meeting interaction. In fact, during the development of our system, Elvin was originally used as the notification server in the integration solution, and became a bottleneck for the scalability of the system. The use of a simpler routing strategy as the one programmed in YANCEES solved this problem.

9. CONCLUSIONS

This paper introduced a new collaboration model that seamlessly integrates existing collaboration modalities into a single consistent interaction model. This model facilitates the development of contextual collaboration applications such as Activity Explorer. Our simulation data indicted that our native implementation proves to scale sufficiently well except for high-bandwidth, high-frequency data traffic. Depending on the application scenario, real-time collaboration servers can improve the performance of the model. The use of notification servers to support the model was problematic. In the future, we would like to better understand the limits of the model by schematically varying the size of the data messages, frequency, and the number of members per object instead of using traffic patterns. Also, our model currently treats asynchronous and synchronous modifications of the content of a GSO in a very similar way. We are exploring alternative ways of improving the performance of the system by reducing the number of persistent GSO content updates and notifications (to members who do not have the object open) during phases of synchronous collaboration.

10. ACKNOWLEDGEMENTS

We would like to express our gratitude to John Patterson and the Activity Explorer product team for their inspiring discussions.

11. REFERENCES

[1] M. Muller, W. Geyer, B. Brownholtz, E. Wilcox, and D. Millen, "One Hundred Days in an Activity-Centric Collaboration Environment based on Shared Objects," presented at ACM SIGCHI, Vienna, Austria, 2004.

[2] W. Geyer and L. Cheng, "Facilitating Emerging Collaboration through Light-weight Information Sharing," presented at CSCW'02, New Orleans, LA, 2002.

[3] W. Geyer, J. Vogel, L. Cheng, and M. Muller, "Supporting Activity-Centric Collaboration through Peer-to-Peer Shared Objects," presented at ACM GROUP, Sanibel Island, FL, 2003.

[4] D. Millen, M. Muller, W. Geyer, E. Wilcox, and B. Brownholtz, "Patterns of Media Use in an Activity-Centric Collaborative Environment," presented at ACM SIGCHI, Portland, Oregon, WA, 2005.

[5] L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson, "Jazzing up Eclipse with collaborative tools," presented at OOPSLA'03 workshop on eclipse technology eXchange, Anaheim, CA, 2003.

[6] P. Moody and J. Feinberg, "C+B Seen Project," presented at http://domino.research.ibm.com/cambridge/research.nsf/pages/projects.html.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley Publishing Company, 1995.

[8] R. S. Silva Filho, C. R. B. de Souza, and D. F. Redmiles, "The Design of a Configurable, Extensible and Dynamic Notification Service," presented at DEBS'03, San Diego, CA, 2003.

[9] G. Fitzpatrick, T. Mansfield, D. Arnold, T. Phelps, B. Segall, and S. Kaplan, "Instrumenting and Augmenting the Workaday World with a Generic Notification Service called Elvin," at ECSCW '99, Copenhagen, Denmark, 1999.

[10] W. Geyer, H. Richter, L. Fuchs, T. Frauenhofer, S. Daijavad, and S. Poltrock, "A Team Collaboration Space Supporting Capture and Access of Virtual Meetings," presented at ACM 2001 International Conference on Supporting Group Work, Boulder, CO, USA, 2001.

[11] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems (TOPLAS*, vol. 7, 1985.

[12] P. Wyckoff, "TSpaces," *IBM Systems Journal*, vol. 37, 1998.

[13] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces Principles, Patterns, and Practice*: Book News, Inc, 1999.

[14] J. F. Patterson, M. Day, and J. Kucan, "Notification servers for synchronous groupware," presented at CSCW'96, Boston, Massachusetts, 1996.