Journal of Universal Computer Science, vol. 14, no. 1 (2008), 4-26
submitted: 6/11/06, accepted: 27/7/07, appeared: 1/1/08 © J.UCS

The Trade-Offs of Blending Synchronous and Asynchronous Communication Services to Support Contextual Collaboration

Werner Geyer
(IBM T.J. Watson Research, USA
werner.geyer@us.ibm.com)

Roberto S. Silva Filho
(University of California Irvine, USA
rsilvafi@ics.uci.edu)

Beth Brownholtz
(IBM T.J. Watson Research, USA
beth_brownholtz@us.ibm.com)

David F. Redmiles
(University of California Irvine, USA
redmiles@ics.uci.edu)

Abstract: Contextual collaboration seamlessly integrates existing groupware technologies into a uniform user experience that combines synchronous and asynchronous interactions. This user experience is usually supported by a collaboration infrastructure that needs to efficiently cope with the fast switching and integration of different modes of interaction. In this paper, we study a model for contextual collaboration that supports multiple modalities of collaboration. Our model is based on generic shared objects that provide building blocks for supporting contextual collaboration applications. We describe a native implementation of this model and evaluate its behavior under different media traffic conditions. We compare the native implementation with an alternative implementation that integrates existing notification and meeting servers to deliver the same model behavior. We discuss trade-offs and limitations of those two implementations.

Keywords: Synchronous collaboration, asynchronous collaboration, CSCW, groupware, simulation, architecture, notification
Categories: I.6.0, H.3.4, H.4.3, H.5.3

1 Introduction

Contextual collaboration promises new levels of productivity by seamlessly integrating content sharing, communication channels, and collaboration tools into a unified user experience. One form of contextual collaboration embeds collaborative features, such as presence awareness, instant messaging, real-time conferencing, file exchange, and virtual workspaces into other business applications (see [Mahowald, 06], [SearchDomino.com, 06]). 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, they can start a chat from

within their document editors, share a document on their desktops by dragging it on their buddy lists, or start a remote presentation by right-clicking on a presentation file on their desktop. Contextual collaboration lowers the end user's barrier to engage in collaboration by transparently integrating existing groupware technologies. By doing so, it reduces the end users' cognitive cost of switching between collaboration tools and applications, providing contextual points of access to a set of inter-related applications and the artefacts they produce. A highly contextualized user experience entails frequent changes in work mode and modalities. From an infrastructural perspective, this requires the use of different services, for example, meeting servers to support synchronous collaboration, notification servers to support timely delivery of messages, or document repositories to allow sharing of content.

While many commercial products today provide those kinds of backend services, integrating these services is challenging because they have been designed to support a single modality only. Developers of contextual collaboration applications face the decision whether or not to reuse and integrate existing single modality backend systems or to develop a new infrastructure from the scratch. There are many trade-offs that need to be considered. For example, reusing existing systems can lower the development costs but might lead to increased integration complexity and low robustness. Integrating existing systems might increase scalability because of the ability of a distributed deployment but might lead to low overall responsiveness of the system. In this paper, we focus mainly on performance trade-offs in a client-server architecture and on integration complexity.

We introduce a model for contextual collaboration that supports multiple modalities of media collaboration. Designing multiple modalities into a single collaboration model addresses the integration complexity for an application developer and also allows us to compare different implementations in order to better understand the integration trade-offs. Our model is based on generic shared objects that provide building blocks for supporting contextual collaboration applications. We present a native implementation of this interaction model and study its behavior under different interaction patterns, representing different kinds of media collaborations. We compare our native service implementation with an alternative integrated implementation where existing services such as meeting and notification servers are used. Our goal is to characterize and understand the trade-offs and limitations that exist in different implementations of services supporting contextual collaboration with respect to the responsiveness of the infrastructure and its ability to support the traffic requirements of different collaboration tools.

This work was motivated by previous research on Activity Explorer (AE) (e.g. [Geyer, 06] or [Geyer, 03]). AE provides a highly contextualized user experience integrating synchronous and asynchronous types of collaboration. AE is built on top of our collaboration model using generic shared objects (GSO). Previous works, however, did not analyze the limitations of the model in terms of scalability, support for different media interaction, and the trade-offs involved in building such an infrastructure using existing technologies. Hence, with this work, we expect to understand the applicability of the model to different traffic conditions, and to assess the use of existing services in supporting this blended collaborative model. The lessons learned can be applied to the development or improvement of contextual collaboration infrastructures.

The remainder of this paper is structured as follows: Section 2 of this paper discusses related work. In Section 3 we describe the contextual user experience in AE in more detail. Section 4 introduces the contextual collaboration model used as the basis for our study. Section 5 describes the two implementations of this model. In Section 6 we describe our simulation environment, the experiments performed, and the experimental results comparing both implementations. Section 7 discusses general trade-offs and lessons learned.

2 Related Work

The GSO model described in this paper supports contextual collaboration by blending synchronous and asynchronous interaction modes in a single service that supports different media interaction modalities. As such, it shares characteristics present in existing collaboration infrastructures.

Notification servers, as defined by Patterson et al. [Patterson, 96], provide a simple common service for sharing state in synchronous multi-user applications. They address the problem of maintaining consistency in real-time applications and supporting awareness. Compared to our shared objects model, state is usually not persistent, and the support for application-specific synchronous interaction modes is not provided.

Similar to persistent notification servers, publish/subscribe systems offer general purpose event notification functionality based on the observer design pattern [Gamma, 95]. Notification servers such as Elvin [Fitzpatrick, 99] or YANCEES [Silva, 05b] 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 usually not supported, and notification delays are common. Moreover, those systems are not usually designed to support synchronous real-time interaction.

The insufficiency of the publish/subscribe model in supporting different groupware applications is also discussed in [Souza, 02] and [Kantor, 01], where new services around this model are proposed to address some of the deficiencies such as the lack of flexibility in the notification model, or the support for end-user subscriptions.

Tuple Spaces, proposed by Gelernter as part of the Linda coordination language [Gelernter, 85] are currently implemented in IBM's TSpaces system [Wyckoff, 98] and SUN's JavaSpaces [Freeman, 99]. They provide a persistent shared memory accessed through an API that allows distributed processes to read, write, and remove information represented as tuples. Compared to our shared objects, Tuple Spaces are rather a generic programming paradigm that helps developers with concurrency control and other issues, while in our approach we focus on offering a shared object service that can be used to build collaborative applications. As an application-specific implementation, our approach provides a more convenient model natively supporting membership, notifications, hierarchical data structures, and application-specific synchronous communication. While the construction of a GSO model on top of tuple spaces would be feasible, the need for the management of membership, the

hierarchical composition of objects and the support for application-specific interaction modes would require deeper changes in the tuple-space model.

As we moved our Activity Explorer research to a commercialized product [Geyer, 06], we needed to understand the implications of using different backend technologies in terms of scalability and performance. While much research has been done on specialized collaboration services, such as notification or meeting servers, the technical aspects of blending different services to provide contextual collaboration experiences have not been addressed thoroughly. Preguiça et al. [Preguiça, 05] provide a very good description of the general problem space but focus mainly on consistency control issues. Along the same lines, Geyer at al. [Geyer, 03] also focus on consistency control aspects. Munkvold and Zigurs [Munkvold, 05] describe challenges and opportunities of integrating support for multiple modalities in collaborative applications. Since Activity Explorer provides a highly contextualized user experience integrating synchronous and asynchronous types of collaboration, we decided to use it as the framework for our research.

3 Activity Explorer

Activity Explorer (AE) is a contextual collaboration application based on the paradigm of activity-centric collaboration [Geyer, 06]. AE runs as a stand-alone desktop application that connects to a contextual collaboration server implementing our collaboration model. In AE an activity is a set of related, shared objects representing a task or project. The set of related objects is structured as a hierarchical thread called activity thread, representing the context of the task at hand. Users start new activity threads by creating root objects from any type of content or communication. Users add items to an activity thread by posting either a response or a resource addition to its parent object. Activity threads combine different types of objects, membership, and alerts. The context (membership and content of the activity thread) is made persistent thought the use of shared objects. AE supports sharing of six types of objects: message, chat transcript, file, folder, annotated screen snapshot, and to-do item.

Figure 1 shows the main AE user interface. The Activity List tab (A) is a multi column “inbox-like” activity list that supports sorting and filtering of activities and shared objects. New activities always bubble up on top of this list per default. The Activity Tree view (B) shows an overall tree structure of all your activities and can be used in a “Windows Explorer” like fashion. Selecting a shared object in the list or tree view populates a read-only info pane (C). The Activity Thread pane (D), maps a shared object as a node in a tree representing an entire activity thread. Activity Thread and Activity List are synchronized by object selection. Users interact with objects or members, as displayed in these views, through right-click context menus. Representative icons are highlighted in green or surrounded by green boxes to cue users of shared object access and member presence (e.g. (3), (4), (6) and (9) in Figure 1).

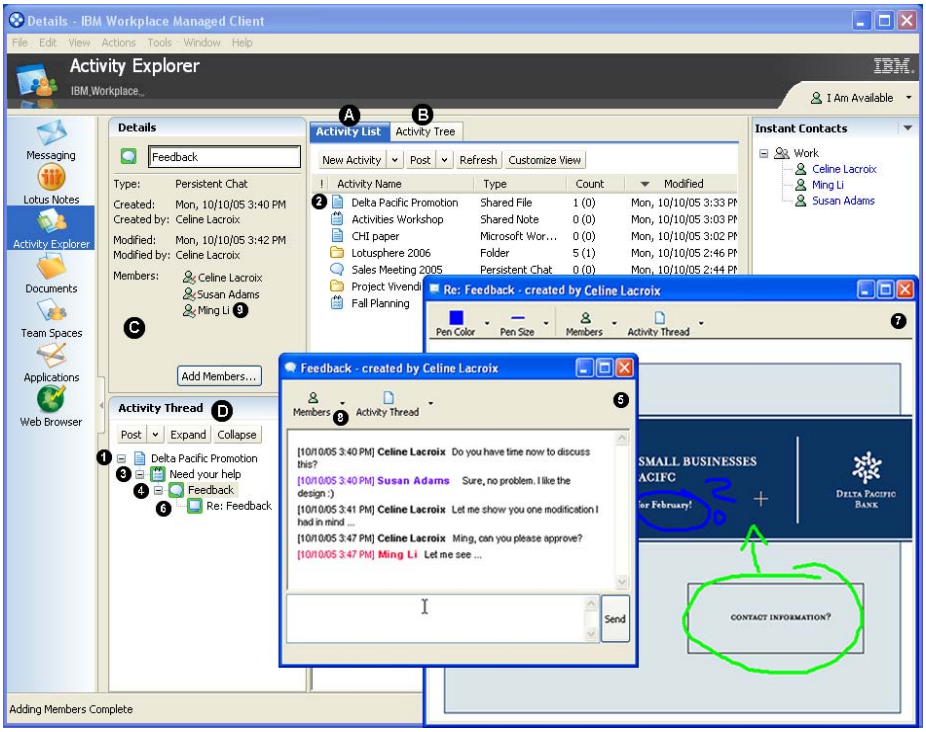


Figure 1: Activity Explorer User Interface

The following scenario illustrates the contextual user experience in which shared objects are used as building blocks for an activity that starts from a document. This scenario highlights only core features; for a more complete description of Activity Explorer's capabilities see [O'Neil, 05]. Figure 1 is a snapshot of an activity in progress, shown from the perspective of one of the actors (Celine). The activity thread is built dynamically as the actors collaborate.

***Celine** is a designer and she works with **Susan** on a print promotion flyer for Delta Pacific bank. **Ming** is their project manager. The first review meeting with Delta Pacific is approaching. Celine has crafted a draft of the flyer and would like Susan's feedback. Celine glances at her Instant Contacts for Susan's name, and sees that Susan is currently offline. From her desktop, Celine drags the draft image file on to Susan's name, starting a new activity thread named "Delta Pacific Promotion" (1). The file is now shared and shows up as a new activity in Celine's activity list (2). She right clicks on the file object to add a message asking Susan for her comments (3).*

A few hours later, Susan returns to her desktop. In the system tray, Susan is alerted to the new activity by an alert message (whenever an object is changed – including the addition of a child object – all people who have access privileges on that object receive an alert message about the change). Clicking on the alert, she is taken to the activity thread. She opens the message and while she is reading it, Celine

notices Susan is looking at the message because the shared object is lit green (3).[1] *Celine seizes the opportunity to expedite their progress; she right clicks on the initial message and adds a chat to this activity (4). A chat window pops up on Susan's desktop and they chat (5). Celine refers to a detail in the image file; for clarity she wants to show Susan what she would like changed. By right clicking on the chat object, Celine creates a shared snapshot object (6). A transparent window allows Celine to select and "screen scrape" any region on her desktop. She freezes the transparent window over the draft image. The snapshot pops up on Susan's desktop. Celine and Susan discuss a few changes by annotating the image in real-time like a shared whiteboard (7).*

Aware of the upcoming deadline, Celine wants Ming informed about the status. Within the chat, she selects 'Invite' to add Ming as a member (8). On his client, Ming receives a pop-up invitation to join the chat and he accepts. Note that Ming is now a member of the chat and the shared snapshot only and not of the other objects in the activity (9). Ming approves the changes and Celine begins to work on the changes.

This scenario demonstrates how Activity Explorer helps people move seamlessly and effortlessly back and forth from private to public information and from asynchronous to synchronous real-time collaboration, without manually creating a shared workspace or setting up a meeting. Collaboration starts off with a single shared object and evolves into a multi-object activity, which is structured by a dynamic group of participants as they create and add new shared objects. An activity thread provides a persistent activity context aggregating a mix of different object types. Alerts provide up-to-the-minute awareness of person-relevant changes, even if Activity Explorer is not the top-most application on the user's computer.

4 Contextual Collaboration Model

The contextual collaboration model behind AE is based on the concept of Generic Shared Objects (GSO) [Geyer, 03]. GSOs are persistent collaboration objects, in programming language terms, that can be used as building blocks for new collaborative applications that require a seamless, contextual user experience with blended synchronous and asynchronous collaboration. This generic model provides both simplicity and uniformity, allowing the extension of the service to new media types, and the uniform composition of artifacts into hierarchies such as activity threads. GSOs combine various collaborative functions such as group communication, content management, notifications, and membership-based access control policies into objects that can be hierarchically composed.

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. This architectural style was selected for being currently supported in the AE prototype, as well as in existing technologies such as notification servers and meeting servers used in our experiments. Note that the GSO model can be also implemented in different architectural styles (e.g. see [Geyer, 03]).

---

[1] A fully accessible version of this concept would include a text alternative to the green highlighting.

The GSO communication protocol is based on three basic primitives: Request, Response, and Notification: A client interacts with a GSO by issuing a Request to that object (for example, reading an attribute, adding a new member, reorganizing the object hierarchy and so on). The object then replies with a Response to the requesting client. Depending on the type of request, the object can also send out Notifications to currently online clients as illustrated in Figure 2 (b).

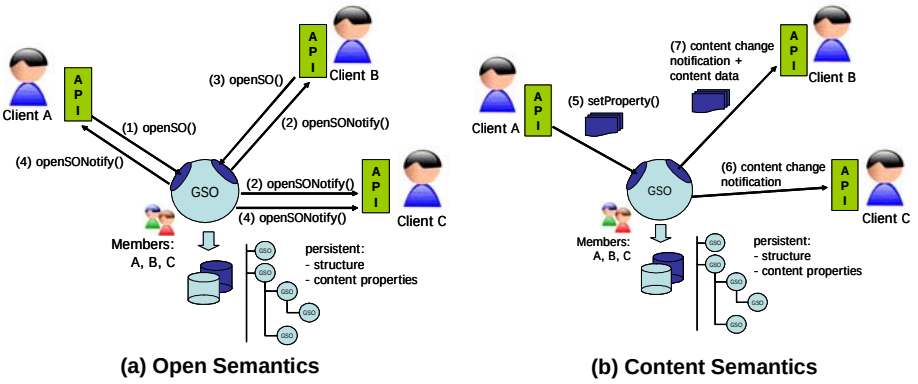


Figure 2: Generic Shared Object behaviour

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 modelling complex collaborations such as the previously mentioned activity threads in AE (see Figure 1 D).

Each GSO provides a simple content model based on a set of properties. The content model describes what kind of data an object shares and stores, for example, chat transcripts, e-mails, file contents, streaming media and so on; e.g. each Shared Object in AE is represented by a GSO. Jazz [Cheng, 03] and C&BSeen [Moody, 06] are other examples of applications that use GSOs in a less direct way. Note that 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. The interpretation and use of the <name, value> pairs is left to client applications, which provides flexibility to the model. The value field can be of various types, e.g. String, Integer, Double, Boolean. For example, the persistent chat object in AE, stores each chat message as a String property.

Every GSO represents a "persistent conferencing session" between its members. The distribution of content (synchronous or asynchronous) is performed through the use of notifications. Any modification to the set of properties of a GSO is not only stored in the underlying data store, but also automatically sent as notifications to all the other members of that GSO. Hence, our model provides a different paradigm for real-time collaboration based on persistent state and state change notifications. In AE for example, for each new chat message (stored as a new property), the system sends out a notification informing clients about the new property.

Each GSO also manages a list of members (e.g. A, B, and C in Fig. 2). The GSO member list controls the access to its content and represents a distribution list for sending notifications about the creation and modifications of a GSO. The member list is dynamic, allowing the addition and removal of existing members at runtime. Since the member list is also a property of the GSO, any modification to this list, triggers notifications that are sent to all online GSO members.

Notifications of content change come in two different modalities controlled by the use of open and close requests. Change notifications (without the actual content) are sent to all online members of the object whose open status for that object is false. Notifications with the actual content are sent to all online members whose open status for that object is true. Setting open to true basically subscribes a member to receive the content together with the content change notification. This semantic is important to prevent members that are not interested in certain objects from receiving unnecessary information each time a change is made to an object. In the AE user interface, the open status is hinted by a green icon if at least one member of the object sets the status to true.

Since all GSO content changes persist, GSO properties are still available when clients disconnect and later reconnect to the service. This allows members of an object to interact asynchronously. In summary, the described behavior of GSOs inherently merges real-time conferencing with content management and asynchronous collaboration modes.

5 Implementation

In order to study and better understand the implications and trade-offs of combining various interaction modes of collaboration in a common model, we have built two implementations: (1) a server that implements the GSO collaboration model natively; and (2) a server that uses existing collaboration technologies to deliver the same functionality offered by our model.

5.1 Native Implementation

In our native implementation, the GSO concept is directly mapped to persistent objects (using the OO programming paradigm). The implementation of the GSO manages every aspect of the model, i.e. content management, membership, access control, notifications, data transfer and 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 API (see Figure 2).

In the example of Figure 2 (a), clients A, B, and C are all members of a GSO object. 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 currently working on the GSO content. 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 Figure 2 (b), client A changes the content of the GSO 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 native server is implemented in Java and communicates via Remote Method Invocation (RMI) with its clients. Notifications are sent to clients through RMI also. 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 prototypes.

5.2 Integrated Implementation

In our alternative integrated implementation, the initial native implementation was modified to perform synchronous interaction through meeting servers and to deliver events using a notification service. The integration of the two new backend technologies was completely transparent to the end users. Clients interact through the same GSO service API (see Figure 3). In the backend, however, the implementation complexity increased significantly.

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 synchronous meeting and the persistent aspects of the model. A SSC is a special client in a meeting session. A meeting is a session created between two or more participants/clients that provides a non-persistent shared space where messages are sent to all the meeting members. The SSC is responsible for storing session data in a persistent repository by updating the respective GSO when content is changed. For example, when a chat message is posted to a meeting session, the SSC for that session stores the message in the GSO, which itself triggers a notification. This approach provides a generic mechanism that can be used to transparently integrate any meeting server.

Note that using meeting servers to support real-time collaboration entails setting up a meeting session with the meeting server every time a client opens a shared object for real-time interaction (see Figure 2 (a)). Likewise the meeting session needs to be disposed every time the client closes the GSO. For each session, a SSC also needs to be created in the beginning and disposed in the end.

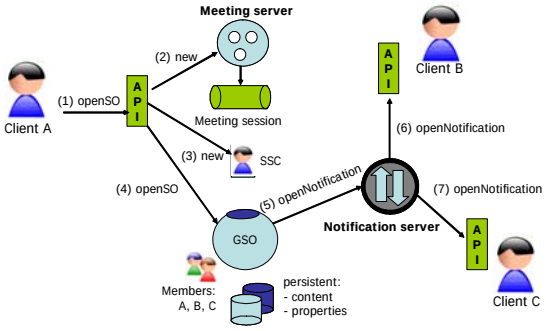
We integrated a notification server into the service to support asynchronous change notifications. Whenever a GSO's property or content is changed, a single notification is produced. Differently from the previous native implementation, that produced one notification per GSO member, a single message is now relayed to a notification server that is responsible for distributing the notification to all the members of the object. The subscription style is topic-based: each client subscribes/un-subscribes to a global GSO notification topic when logging on and off the service. 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.

Figure 3 illustrates the integration complexity. 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 (4). Consequently, an open notification is sent to all the members of the object (5, 6, 7) through the notification server. The SSC joins the meeting (8) and listens to messages on 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).

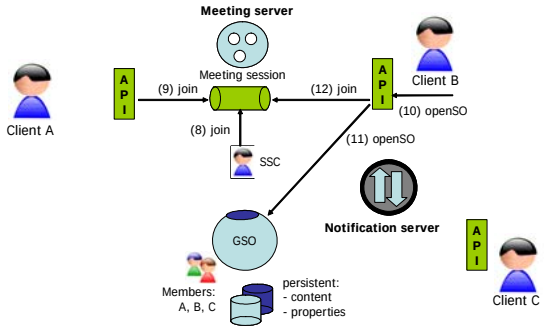
The open call is propagated to the GSO object (11), which returns the existing meeting id. Client B also automatically joins the meeting (12). As content messages are exchanged between members A and B and the SSC (14, 15, 16), the SSC makes the content persistent by invoking *setContent()* on the GSO (17). The GSO then 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).

The integrated solution was also completely implemented in Java. We used YANCEES [Silva, 05b] as the notification server because of its ability to be configured with a simple topic-based core, and for having a simple API, similar to Elvin [Fitzpatrick, 99]. Additionally, in our preliminary tests with both servers, YANCEES has shown to outperform Elvin in its throughput and ability to handle a large number of subscriptions. Since we were running our experiments in a dedicated LAN, we decided to run the notification server with client-side filtering of events. This approach simplified subscription management and resulted in better performance than server-side event filtering (discussed in Section 6.3). We used a simple Java-based meeting server from the TeamSpace project [Geyer, 01]. We sought to keep the two implementations as similar as possible in order to get meaningful results for a comparison, e.g. both implementations share the same common GSO model and externalize the same GSO API. However, given the number of different existing publish/subscribe and real-time collaboration systems, our simulation results may vary depending on the backend technologies used.

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 through the meeting. The SSC updates the data model and notifications are sent to the clients

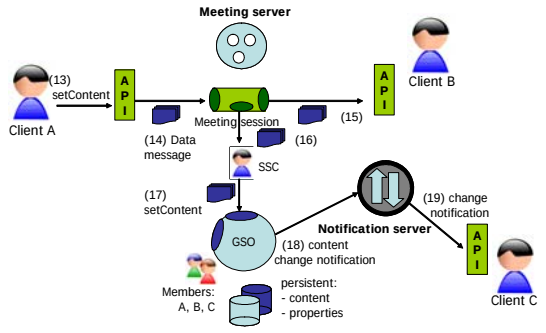


Figure 3: Integrated implementation of the GSO model

6 Experimental Results

The model described in Section 4 unifies characteristics of publish/subscribe systems, synchronous collaboration servers, and content management in a uniform and flexible way. As such, it facilitates the development of collaborative applications that have contextual collaboration characteristics. This blending of synchronous and asynchronous collaboration, however, 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 small groups, 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 real-time constraints, focusing on awareness and messaging, where the number of clients is potentially large and the data traffic is relatively small. When those two different interaction modes are combined in a single collaboration model, different trade-offs involving scalability, responsiveness, robustness, and implementation complexity have to be considered. We conducted a series of experiments to understand these trade-offs and answer the following questions: How well do the two different implementations of the model handle the blending of synchronous and asynchronous collaboration? What is the impact of different data rates and data sizes depending on the type of media interaction? How is the response of the infrastructure to different combinations of those factors?

6.1 Experimental Setup

Since we wanted to understand the behavior of the model under regular use conditions, we decided to model typical user behaviour and use event-based simulation to test the different implementations (e.g. [Munkvold, 05], [Preece, 02], or [Vogel, 03]). We developed an automated client simulator that interacts with our service implementation using different patterns. Those patterns simulate the use of different collaborative tools with their traffic conditions, number of users and data size. The simulator client exercises the server APIs performing regular actions such as: create new object, set properties, open, close, add member and so forth. For the purpose of our tests, we defined four different patterns approximating the traffic conditions of chat, file sharing, message exchange, and streaming media. The streaming media pattern was defined to analyze the server behavior under heavier load, testing its scalability limits. Note that these patterns are only approximations of actual interaction patterns and some values, such as the number of members involved in a pattern represent an estimation of actual usage parameters observed in our previous field studies with AE [Millen, 05]. Table 1 describes the different patterns with their data characteristics and probabilities used in our experiments.

The main differences between the four media traffic patterns are in the size of the data, 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 client corresponds to an interaction with a GSO with two members on average. In this interaction, each member exchanges 10 messages on average. Each message has an average length of 40 characters. Chat messages are exchanged at

every 15 seconds on average. During this interaction, periods of inactivity may also occur with an average duration of 15 seconds.

Media Pattern	nº Members	Data			Content change probabilities		
		Size (chars)	nº 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

Table 1: Media pattern programming used in our experiments

In our GSO model, a property can be set (overwritten or created), added (appended to the end of the current content), or deleted. Table 1 also 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 they grow over time as new messages are exchanged.

For each pattern, we reproduce the actions of a typical work day of 8 hours. We programmed our automatic client to perform those actions in a simulation time of 4 minutes. This setup is similar to [Geyer, 03] and allows us to stress test the infrastructures using a reduced number of clients. During one simulated workday, the following actions are performed by the client: A total of 15 shared objects are created on average 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 interaction patterns also differ with respect to the time span that each client is working either online or offline.

All experiments were carried out on three client machines (IBM T30, 1.6GHz, 512MB) and one server machine (IBM MPro, 3 GHz, 1.5 GB). The client machines and the server were connected on an isolated 100Mbps Ethernet local network to eliminate interference with other network traffic. Client machines were equally loaded with a set of client simulators in steps of one, i.e. the first test 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 client processes running on a single client machine impacts the overall simulation results. Based on tests, we decided to limit the number of automated clients to eight per client machine in order to minimize this effect.

6.2 Results: Native Implementation

In order to understand the overall service behavior to the different media patterns, we plotted the total average execution times for each one of the four patterns against the number of clients interacting with the system. In this experiment, each client process executes a typical work day, using a single interaction pattern which includes open and closing objects, logging in and out, offline times and content changes.

Figure 4 shows that the system has a linear response to the increase in the number of clients, for low-frequency traffic patterns such as chat, message exchange, and file sharing. The graph also shows that the size of the data, as in the case of file sharing,

does not impact performance as much as the frequency of the messages. The main characteristic of streaming media is its high frequency of relatively large data messages. As can be seen in Figure 4, our reference implementation does not scale as well for this pattern (it grows in a non-linear fashion). This can be explained by the fact that we send out content change notifications (with or without the actual content) to every member of the GSO. Given the high data frequency of streaming media, the server load increases quickly, since each data message triggers a series of content change notifications, typically one for each member of the objects involved.

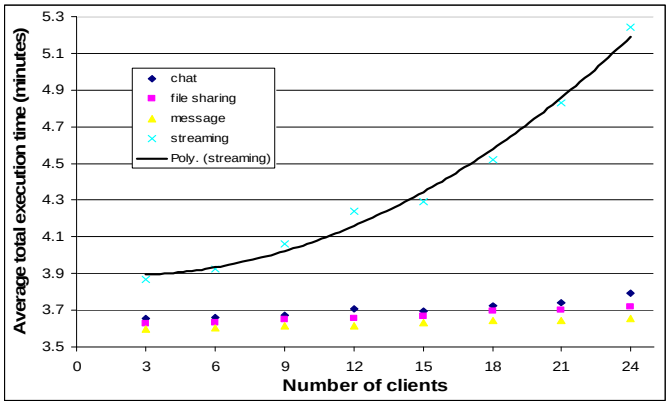


Figure 4: Average total execution times of the native implementation under different activity patterns for a typical workday

In another experiment, under the same experimental conditions, we sought to understand the responsiveness of our implementation. The responsiveness of a collaborative system is defined by its response and notification times. The response time describes how fast the system reacts to user input, i.e. how fast actions are reflected in the user interface of the clients executing the action and receiving responses. The notification time describes how fast a collaborative system updates remote clients. In a collaborative setting, it is desirable to keep this number as low as possible in order to keep all clients in sync with each other minimizing lag. Response time 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 performed by the *setContent()* API call.

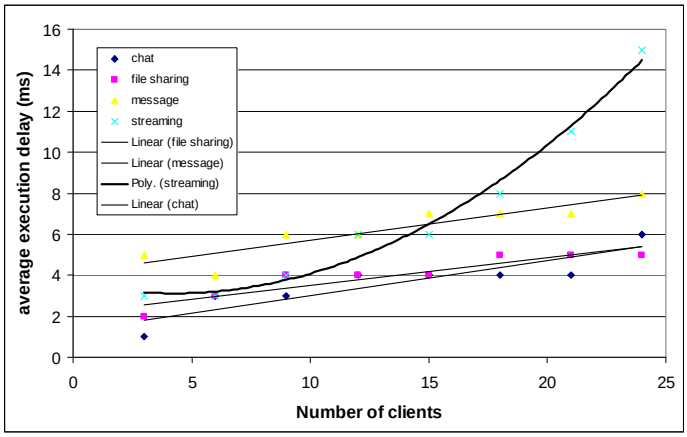


Figure 5: Average execution time of the setContent() call in the native implementation with different media patterns

Figure 5 shows that the execution times for the *setContent()* API call are relatively low (in the order of milliseconds). They grow linearly with the number of clients for all interaction patterns, except for the streaming media pattern. For a small number of clients, and consequently a small 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, the response time of the system to this pattern grows quadratically. Note that the message pattern initially has a relatively high execution time compared to streaming media. The reason is the higher number of members in that pattern (eight on average). This demonstrates the low impact of notifications (without data) relative to the frequency of interaction with the system.

In the same experiment, we also measured notification times: the period of time from calling a method in the client API to the delivery of its notification to the other members of a GSO.

Figure 6 shows the average execution vs. notification times for creating new GSOs. In this experiment, the 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 made visible to remote clients at about the same time. Except for streaming media *setContent()* calls, the execution times of the native implementation are relatively low (i.e. below 10ms) and the notification delays are extremely low.

Our experiments indicate that the performance of the system 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 (low frequency and low bandwidth) the model scales very well having good responsiveness. However, for streaming media traffic, with a relatively medium number of members, and an average volume of information, the system delays increase quadratically.

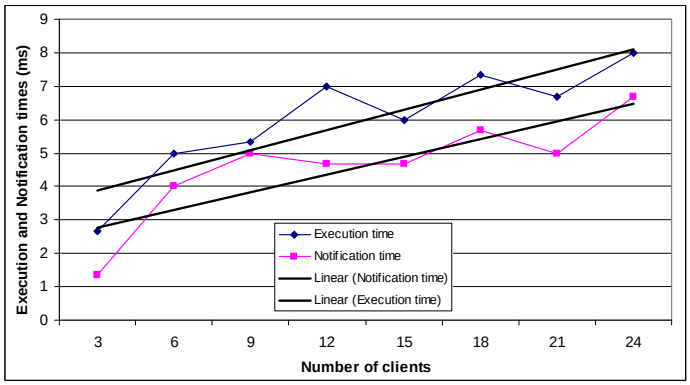


Figure 6: Average execution vs. notification times for creating new GSOs in the native implementation

6.3 Results: Integrated Implementation

Existing real-time collaboration servers are optimized for online meetings with a smaller number of participants but relatively high data volume, e.g. audio, video. Given the results in the previous section, it seems reasonable to apply real-time meeting servers to support frequent and high volume property changes in a GSO. We hypothesized that the implementation of the synchronous aspects of our model with a meeting server would increase the overall system performance.

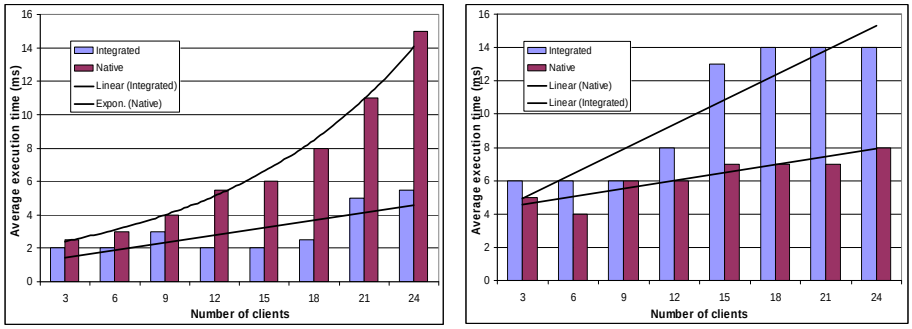
Notifications are another aspect of our model that we believed to be well understood today. Publish/subscribe systems provide general-purpose event notification services. Notification servers receive anonymous notifications and route them to interested parties. This routing is orchestrated by subscriptions. These systems are typically optimized for a very large number of subscribers and small to medium data volumes for each subscriber. We hypothesized that GSO events such as create / delete GSO, add/remove member, or infrequent property changes (e.g. changing the presence status of a member on an object) would be well supported by a publish/subscribe system.

Hence, we expected that our integrated implementation of the model using meeting and notification servers, would result in better scalability of both the notification process (asynchronous mode in our model), and the synchronous collaboration through content exchange (the synchronous mode of our model). An expected price to be paid, however, would be the extra cost of integration and the increased complexity of the architecture. In order to verify this hypothesis, we repeated the same set of tests with the integrated service implementation.

Figure 7 (a) compares the cost of the set/add content calls in both implementations for the streaming media pattern. As expected, the integrated implementation scales better, in a more linear fashion, than our original native implementation. In other words, 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. The message exchange pattern, however, yielded some interesting results. Figure 7 (b) shows that the use of our meeting server was more costly, in terms of performance, 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 (eight on average for the message pattern). While our meeting server seems to handle high bandwidth, high frequency traffic well, performance seems to degrade with an increased number of meeting participants.



(a) Streaming media pattern (b) message exchange pattern

Figure 7: Comparison of the average execution times for setContent() calls

Since the use of a meeting server introduces additional complexity (see Section 5.2), we expected that the price for better scalability during the synchronous interaction phase of a GSO would come with additional delays in the start up of the shared meeting that handles it. The data in Figure 8 compares the cost for opening GSOs in both implementations. The data confirms that the open call, where a new meeting session is started, 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.

Table 2 shows a summary of the average execution times of GSO API calls. Comparing the average execution times of other GSO API calls for both implementations, we noticed that the *registerMember()* and *loginMember()* calls also impose high delays in the integrated implementation. The reason for these delays 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 in that model, since becoming a member of the object automatically subscribes members to receive change notifications. 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).

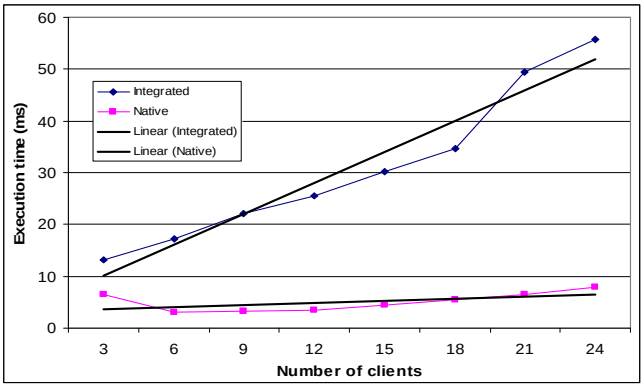


Figure 8: Comparison of the average execution times for openSO() calls

GSO API CALL	NATIVE (ms)	INTEGRATED (ms)
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**

Table 2: Average execution times of GSO API calls in milliseconds (24 clients)

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 9 compares execution times for creating GSOs against the notification time. The integrated implementation has low response 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.

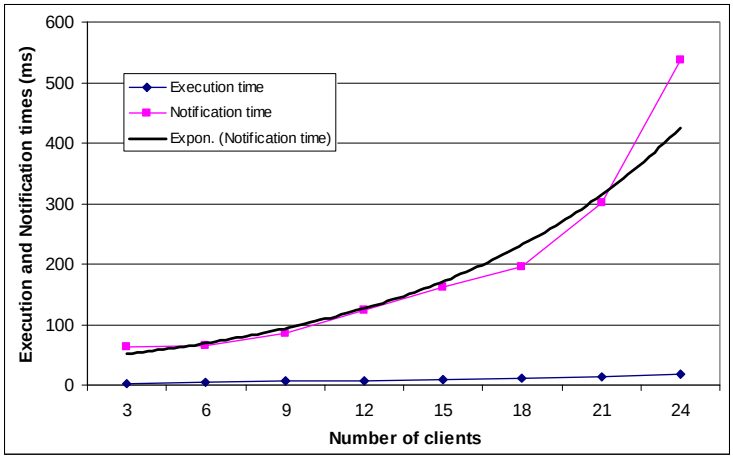


Figure 9: Average execution times vs. notification times for creating new GSOs in the integrated implementation

One could argue that the use of the notification server as a shared bus (event channel) is one of the reasons for the notification server behavior observed in Figure 9. In another alternative implementation, we tested 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 to the client. This approach, however, required constant update of the subscriptions to reflect the current object state (each client manages one or more subscriptions filtering out events that do not belong to the objects they are members of). In the server-side subscription approach, subscriptions need to be updated when new objects are created, members log on/off, or members are removed/added to objects. Given the high subscription costs impacting the *registerMember()* and *login()* operations, this solution did not scale well. These membership and object life-cycle dynamics resulted in similar or worse delays than the ones observed in Figure 9, which led us to choose the client-side filtering option.

7 Lessons Learned

7.1 Interference of conflicting requirements

The support of synchronous and asynchronous interaction in a common and simple model is not a trivial task. While the native implementation of the GSO model supported well the majority of traffic patterns, it did not scale well for high frequency, high-bandwidth data as in our stream media pattern. The use of meeting servers can improve the performance of synchronous message exchange under those circumstances. However, the notification server in our integrated implementation became a bottleneck, impacting the scalability of the entire model. This demonstrates how a combination of different services can interfere with one another, limiting the performance of the overall infrastructure.

7.2 Integration complexity

Our initial hypothesis, that the integration of existing services to support contextual collaboration, would combine the strengths of both services, showed not to be completely true. It had shortcomings in the form of extra complexity. Even though an integrated solution, that uses specialized services, can perform better than a more simple implementation, the integration of those off-the-shelf components usually demands special attention to matters such as timing, synchronization, and adequacy to the model. It also makes the implementation of the system more complex, requiring the combined operation of the notification and meeting servers, an activity prone to errors and additional setup delays, such as startup times, as observed in our experiments, during member log-in and opening objects. While our work in this paper mainly focused on performance and integration complexity, there are also trade-offs between developing a collaboration infrastructure from scratch versus using existing standardized components, paying the extra cost for integration but saving on the development effort.

7.3 Mismatch of programming models

Another issue elucidated in our experiments was a mismatch of the programming models of the different components used. For example, the extension of the meeting server to support persistency was not trivial; our solution was to use a server-side client acting as meeting recorder. Another example was the inadequacy of the notification server in handling frequent subscription changes. In our experiments, we tested the integrated GSO 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, however, had their own trade-offs and limitations, for example, the balance between processing and network bandwidth: Client-side filtering moves part of the processing load to the client side, but requires the delivery of extra notifications through the network. Server-side filtering limits the amount of traffic to the clients and relieves them from discarding unnecessary notifications. Moreover, the latter approach results in an extra burden to the notification server, that needs to deal with constantly changing subscriptions in order to accommodate changes in the GSO membership. As shown in our experiments, the subscription process is usually costly for notification servers, resulting in extra delays in the infrastructure.

8 Conclusion

In this paper we studied two implementations of a new collaboration model that seamlessly integrates different collaboration modalities into a single interaction model. Our model facilitates the development of contextual collaboration applications such as Activity Explorer. Our experiments show the trade-offs of developing contextual collaboration systems based on existing collaboration services such as meeting and notification server versus the implementation of the model from scratch.

The simultaneous support for synchronous and asynchronous interaction in a single model tends to work well in a native implementation for the average case,

where neither the synchronous nor the asynchronous aspects of the model are put to exceeding stress. The low complexity of a native implementation together with high responsiveness might satisfy the requirements of the majority of contextual collaboration applications today. The integration of meeting servers restricted to only media traffic can significantly improve the scalability of the implementation. The use of generic notification servers to support the model, however, was problematic because mapping GSO behavior onto publish/subscribe semantics caused additional overhead.

While the data presented in this paper provides a good understanding of the trade-offs involved in building contextual collaboration applications, our work has focused on Activity Explorer as an example. More work needs to be done to understand the requirements of other contextual collaboration applications. Our results are also limited by our choice of notification and meeting servers used to conduct our experiments. Additional experiments will be necessary to better understand the impact of different backend technologies.

We started to analyze the impact of distribution of our integrated implementation by running notification and meeting servers on different machines, an advantage that comes from the use of existing components. Preliminary tests indicate scalability advantages with an increased number of clients. A more detailed analysis will be required to shed more light on the trade-offs of distributing or clustering various services.

From a model perspective, 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. The GSO model was designed to support multiple collaboration modalities. However, it 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.

Acknowledgements

We would like to thank John Patterson, Juergen Vogel, the Pesto team in Haifa, and the Activity Explorer product and research teams for their inspiring discussions and support. Some aspects of this work were supported by the U.S. National Science Foundation under grant numbers 0534775, 0205724 and 0326105, and an IBM Eclipse Technology Exchange Grant.

References

[Cheng, 03] Cheng, L.-T., Hupfer, S., Ross, S. and Patterson, J., Jazzing up Eclipse with collaborative tools In Proc OOPSLA'03, Workshop on Eclipse Technology eXchange, Anaheim, CA, 2003, 45-49.

[Fitzpatrick, 99] Fitzpatrick, G., Mansfield, T., Arnold, D., Phelps, T., Segall, B. and Kaplan, S., Instrumenting and Augmenting the Workaday World with a Generic Notification Service called Elvin, In Proc. ECSCW '99, Copenhagen, Denmark, 1999, Kluwer, 431-451.

[Freeman, 99] Freeman, E., Hupfer, S. and Arnold, K., JavaSpaces Principles, Patterns, and Practice., Book News, Inc, 1999.

[Gamma, 95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1995.

[Gelernter, 85] Gelernter, D., Generative communication in Linda, ACM Transactions on Programming Languages and Systems (TOPLAS), 7 (1), January 1985, 80-112.

[Geyer, 01] Geyer, W., Richter, H., Fuchs, L., Frauenhofer, T., Daijavad, S. and Poltrock, S., A Team Collaboration Space Supporting Capture and Access of Virtual Meetings, In Proc ACM Group 2001, International Conference on Supporting Group Work, Boulder, CO, 2001.

[Geyer, 03] Geyer, W., Vogel, J., Cheng, L. and Muller, M., Supporting Activity-Centric Collaboration through Peer-to-Peer Shared Objects, In Proc ACM Group 2003, Sanibel Island, FL, 2003, 115-124.

[Geyer, 06] Geyer, W., Muller, M.J., Moore, M., Wilcox, E., Cheng, L., Brownholtz, B., Hill, C.R., Millen, D.R., ActivityExplorer: Activity-Centric Collaboration from Research to Product, IBM Systems Journal – Special Issue on Business Collaboration, Vol. 45, No. 4, 2006, 713-738.

[Kantor, 01] Kantor, M. and Redmiles, D., Creating an Infrastructure for Ubiquitous Awareness, In Proc. Eighth IFIP TC 13 Conference on Human-Computer Interaction, INTERACT 2001, Tokyo, Japan, 2001, 431-438.

[Law, 00] Law, A.M., Kelton, W.D., Simulation Modeling and Analysis, McGraw-Hill, 3rd edition, 2000.

[Mahowald , 06] Mahowald, R, From ICE Age To Contextual Collaboration, IDC, http://www.cio.com/analyst/062901_idc.html, retrieved June 29, 2006.

[Millen, 05] D. Millen, M. Muller, W. Geyer, E. Wilcox, and B. Brownholtz, Patterns of Media Use in an Activity-Centric Collaborative Environment, In Proc. ACM SIGCHI, Portland, Oregon, WA, 2005.

[Moody, 06] Moody, P. and Feinberg, J., C+B Seen Project, retrieved at http://domino.research.ibm.com/cambridge/research.nsf/pages/projects.html, 2006.

[Munkvold, 05] Munkvold, B.E., and Zigurs, I., Integration of E-Collaboration Technologies: Research Opportunities and Challenges, International Journal of e-Collaboration, 1(2), 2005, 1-24.

[O'Neil, 05] A. O'Neil, Discovering Activity Explorer in the IBM Workplace Managed Client, http://www-128.ibm.com/developerworks/lotus/library/ae/, retrieved July 26, 2005.

[Patterson, 96] Patterson, J.F., Day, M. and Kucan, J., Notification servers for synchronous groupware, In Proc. ACM Conference on Computer Supported Cooperative Work, CSCW'96, Boston, Massachusetts, 1996, 122-129.

[Preece, 02] Preece, J., Rogers, Y., Sharp, H., Interaction Design, John Wiley & Sons, Hoboken, NJ, USA, 2002

[Preguiça, 05] Preguiça, N., Martins, J.L., Domingos, H. and Duarte, S., Integrating Synchronous and Asynchronous Interactions in Groupware Applications, In Proc. 11th International Workshop on Groupware, CRIWG 2005, Porto de Galinhas, Brazil, 2005.

[SearchDomino.com, 06] SearchDomino.com. Contextual Collaboration, http://searchdomino.techtarget.com/sDefinition/0,,sid4_gci934929,00.html, retrieved June 26, 2006.

[Silva, 05a] Silva Filho, R.S., Geyer, W., Brownholtz, B, Redmiles, D.F., Understanding the Trade-offs of Blending Collaboration Services in Support of Contextual Collaboration, In Proc. 12th International Workshop on Groupware, CRIWG 2006, Medina del Campo, Valladolid, Spain, Sept. 17-21, 2006.

[Silva , 05b] Silva Filho, R.S. and Redmiles, D., Striving for Versatility in Publish/Subscribe Infrastructures, In Proc. 5th International Workshop on Software Engineering and Middleware, SEM'2005, Lisbon, Portugal, ACM Press, 2005, 17 - 24.

[Souza, 02] Souza, C.R.B.d., Basaveswara, S.D. and Redmiles, D.F., Using Event Notification Servers to Support Application Awareness, In Proc. IASTED International Conference on Software Engineering and Applications, Cambridge, MA, 2002, 691-697.

[Vogel, 03] Vogel, J., Mauve, M., Hilt, V., and Effelsberg, W, Late Join Algorithms for Distributed Interactive Applications, ACM/Springer Multimedia Systems, Vol. 9, No. 4, pages 327–336, 2003.

[Wyckoff , 98] P. Wyckoff , S. W. McLaughry , T. J. Lehman , D. A. Ford, T Spaces, IBM Systems Journal, 37 (3), 1998, 454-474.