Understanding the Trade-Offs of Blending Collaboration Services in Support of Contextual Collaboration

Roberto S. Silva Filho[1], Werner Geyer[2], Beth Brownholtz[2], and David F. Redmiles[1]

[1] Department of Informatics
Donald Bren School of Information and
Computer Sciences, University of California
Irvine, CA, 92697 USA
[2] IBM T.J. Watson Research
One Rogers Street
Cambridge, MA 02142, USA
rsilvafi@ics.uci.edu, werner.geyer@us.ibm.com,
beth_brownholtz@us.ibm.com, 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 contextual collaboration infrastructure that needs to efficiently cope with the fast switching and integration of different modes of interaction. This paper experiments with 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 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.

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 [10, 14] 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 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 artifacts 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

Y.A. Dimitriadis et al. (Eds.): CRIWG 2006, LNCS 4154, pp. 270 – 285, 2006.

example, meeting servers to support synchronous collaboration, notification servers to support timely delivery of messages, or document repositories to allow sharing of content.

In this paper, we study a model for contextual collaboration that supports multiple modalities of media collaboration. 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) [6, 8]. 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. 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.

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 concept of using shared objects to support collaboration is similar to the Tuple Space work, proposed by Gelernter as part of the Linda coordination language [5]. Tuple Spaces are currently implemented in IBM's TSpaces system [18] and SUN's JavaSpaces [3]. 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 programming paradigm that helps developers with concurrency control and other issues, while we focus on offering a shared object service that can be used to build collaborative applications. As such, membership, notifications, and service-oriented communication are an integral part of our model.

Notification servers, as defined by Patterson et al. [12], 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 Tuple Spaces and our shared objects, state is usually not persistent.

Publish/subscribe systems are similar to our work since they offer general purpose event notification functionality based on the observer design pattern [4]. Notification servers such as Elvin [2] or YANCEES [16] 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. 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 [17] and [9], where new services around this model are proposed to address some of the deficiencies such as the lack of flexibility in the notification model, and support for end-user subscriptions.

The technical aspects of blending of synchronous and asynchronous collaboration have been also addressed in [13] and [8]. Preguiça et al. [13] provide a very good description of the general problem space. Compared to our work, they mainly address consistency control issues.

3 Activity Explorer

Activity Explorer (AE) is a contextual collaboration application based on the paradigm of activity-centric collaboration [7]. 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 create 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.

Fig. 1 shows the main AE user interface. My Activities (A) is a multi column “inbox-like” activity list that supports sorting and filtering of activities and shared objects. 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. The outcome of the activity is shown in Fig. 1.

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 perceives Dan is looking at the message due to the turning of the object icon to green (2b). Bob then 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).*

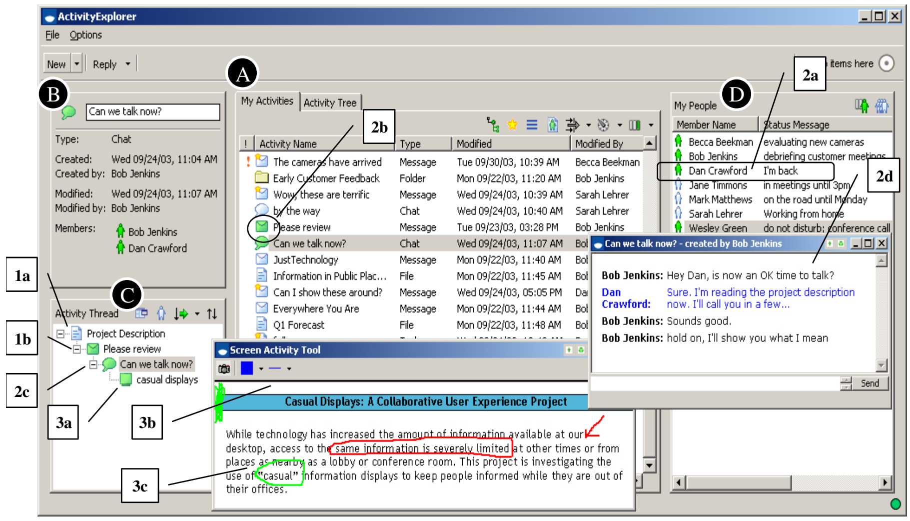


Fig. 1. Activity Explorer User Interface

4 Contextual Collaboration Model

The contextual collaboration model behind AE is based on the concept of *Generic Shared Objects (GSO)* [8]. GSOs are persistent collaboration objects 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 in the integrated implementation described later on in the paper. Note that the GSO model can be also implemented in different architectural styles (e.g. see [8]).

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 Fig. 2 (b).

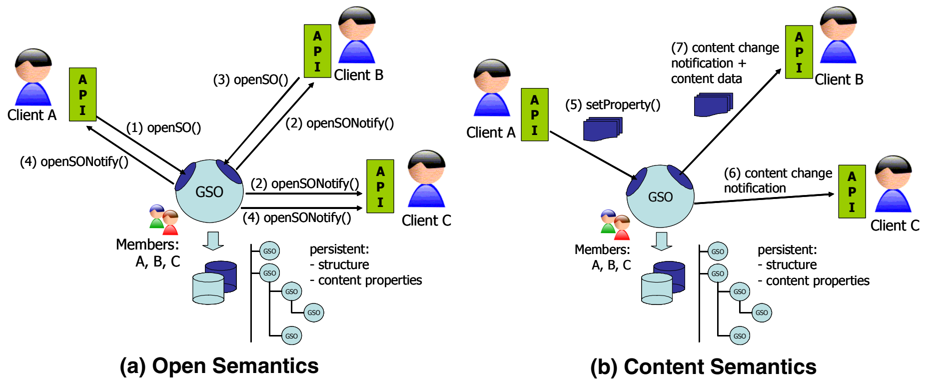


Fig. 2. Generic Shared Object behavior

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 in AE (see Fig. 1 C).

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 [1] and C&BSeen [11] 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. For example, the persistent chat object in AE, stores each chat message as an arbitrary long 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.

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 (or a delta change) are sent to all online members whose open status for that object is true. This semantic is important to prevent members that are not interested in certain objects from receiving unnecessary information each time a change is made in the object.

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 Fig. 2).

In the example of Fig. 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 Fig. 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 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. In the backend, however, the implementation complexity increased significantly. A more detailed description of the service integration and the data flow can be found in [15].

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 Fig. 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 assynchronous 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 used was 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.

The integrated solution was also completely implemented in Java. We used YANCEES [16] 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 [2]. We used a simple Java-based meeting server from the TeamSpace project [7]. 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.

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 an 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 and have strict control of the number of clients connected, 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. Table 1 describes the different patterns with their data characteristics and probabilities.

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 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 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.

Table 1. Media pattern programming used in our experiments

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

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 [8] and allow 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.

Fig. 3 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 Fig. 3, 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.

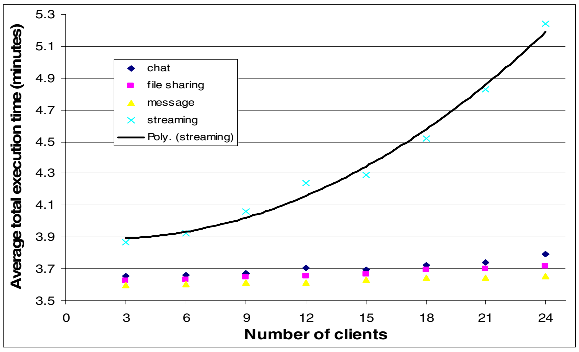


Fig. 3. Average total simulation execution times of the native implementation under different activity patterns for a typical workday

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. Fig. 4 shows the average method execution times for setting the content property of a GSO performed by the *setContent()* API call.

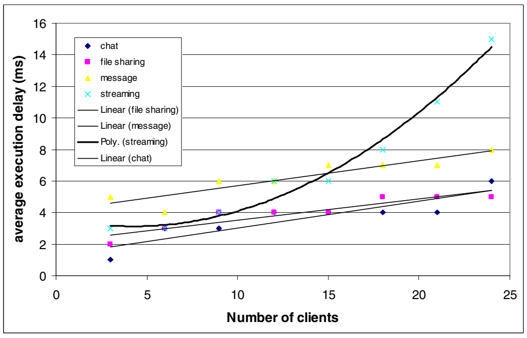


Fig. 4. Average execution time of the *setContent()* call in the native implementation with different media patterns

Fig. 4, 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.

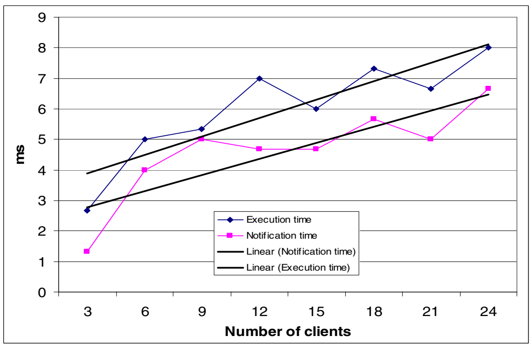


Fig. 5. Average execution vs. notification time for creating new GSOs in the native implementation

Fig. 5 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 response times of the native implementation are relatively high (i.e. below 10ms) and the notification delays are extremely low.

As a general conclusion, our experiments show that 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 (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.

6.3 Results: Integrated Implementation

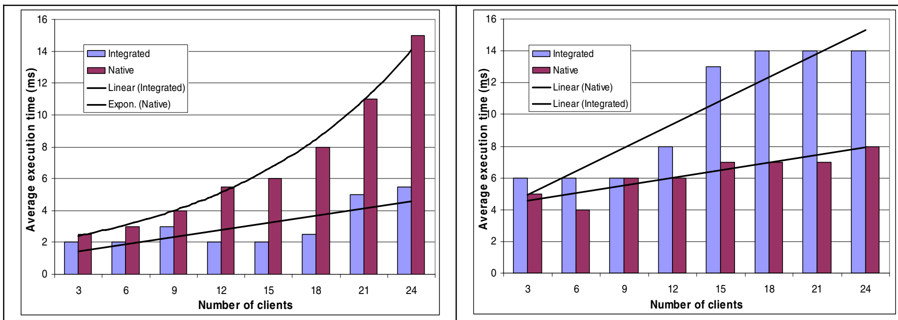
Existing real-time collaboration servers are optimized for online meetings with a smaller number of participants but relatively high data volume, e.g. audio, video. Given the results in the previous section, it seems reasonable to apply real-time meeting servers to support frequent and high volume property 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.

Fig. 6 (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. Fig. 6 (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

Fig. 6. Comparison of the average execution times for *setContent()* calls

Since the use of a meeting server introduces additional complexity (see Section 0), 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 Fig. 7 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.

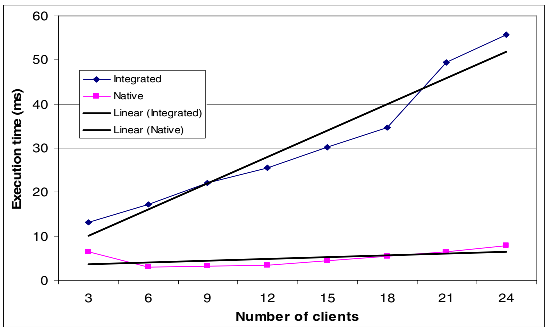


Fig. 7. Comparison of the average execution times for *openSO()* calls

When 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 through the member list.

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. Fig. 8 compares execution times for creating GSOs against the notification time. The integrated implementation has low response times but does not

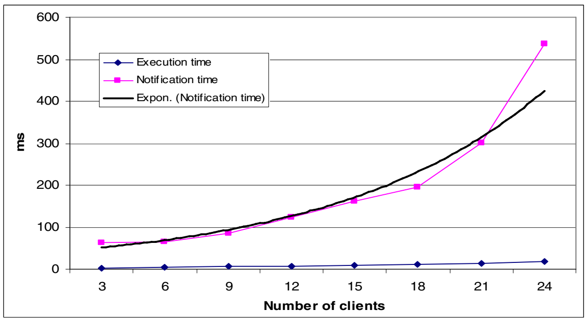


Fig. 8. Average response vs. notification times for creating new GSOs in the integrated implementation

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.

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 Fig. 8. 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 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). 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 Fig. 8.

7 Lessons Learned

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 patters, 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.

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 prone to errors and additional setup delays, such as startup times, as observed in our experiments, during member log-in and opening objects.

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: client-side filtering moves part of the processing 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. However, 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.

Impact of distribution. An advantage of using separate components such as a meeting server and a notification server is the ability to distribute processing throughout different hosts in a network. In additional tests, we distributed the notification and meeting servers across different machines in the network. We found that, with more than 30 clients, the distributed configuration begins to perform better than the centralized approach. This shows that with a significant number of clients, the distribution of main system components is a good approach for scalability.

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. 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.

Acknowledgements. We would like to thank John Patterson, the Pesto team in Haifa, and the Activity Explorer product and research teams for their inspiring discussions.

References

1. Cheng, L.-T., Hupfer, S., Ross, S. and Patterson, J., Jazzing up Eclipse with collaborative tools. in *OOPSLA'03 workshop on eclipse technology eXchange*, (Anaheim, CA, 2003), 45-49.
2. 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 *(ECSCW '99)*, (Copenhagen, Denmark, 1999), Kluwer, 431-451.
3. Freeman, E., Hupfer, S. and Arnold, K. *JavaSpaces Principles, Patterns, and Practice*. Book News, Inc, 1999.
4. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.
5. Gelernter, D. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS*, *7* (1).
6. Geyer, W. and Cheng, L., Facilitating Emerging Collaboration through Light-weight Information Sharing. in *conference supplement CSCW'02*, (New Orleans, LA, 2002).
7. 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 *ACM 2001 International Conference on Supporting Group Work*, (Boulder, CO, USA, 2001), ACM.
8. Geyer, W., Vogel, J., Cheng, L. and Muller, M., Supporting Activity-Centric Collaboration through Peer-to-Peer Shared Objects. in *ACM GROUP*, (Sanibel Island, FL, 2003), 115-124.
9. Kantor, M. and Redmiles, D., Creating an Infrastructure for Ubiquitous Awareness. in *Eighth IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001)*, (Tokyo, Japan, 2001), 431-438.
10. Mahowald, R. From ICE Age To Contextual Collaboration, IDC, retrieved at http://www.cio.com/analyst/062901_idc.html, June 29, 2006.
11. Moody, P. and Feinberg, J., C+B Seen Project. in *http://domino.research.ibm.com/cambridge/research.nsf/pages/projects.html*.
12. Patterson, J.F., Day, M. and Kucan, J., Notification servers for synchronous groupware. in *ACM conference on Computer supported cooperative work (CSCW'96)*, (Boston, Massachusetts, 1996), 122-129.
13. Preguiça, N., Martins, J.L., Domingos, H. and Duarte, S., Integrating Synchronous and Asynchronous Interactions in Groupware Applications. in *11th International Workshop, CRIWG 2005*, (Porto de Galinhas, Brazil, 2005).
14. SearchDomino.com. Contextual Collaboration, http://searchdomino.techtarget.com/sDefinition/0,,sid4_gci934929,00.html, June 26, 2006.
15. Silva Filho, R.S., Geyer, W., Brownholtz, B., Guy, I., Redmiles, D.F. and Millen, D.R. Architectural Trade-Offs for Collaboration Services Supporting Contextual Collaboration - RC23756, IBM T. J. Watson - Cambridge, Cambridge, MA, 2005.
16. Silva Filho, R.S. and Redmiles, D., Striving for Versatility in Publish/Subscribe Infrastructures. in *5th International Workshop on Software Engineering and Middleware (SEM'2005)*, (Lisbon, Portugal., 2005), ACM Press, 17 - 24.
17. Souza, C.R.B.d., Basaveswara, S.D. and Redmiles, D.F., Using Event Notification Servers to Support Application Awareness. in *IASTED International Conference on Software Engineering and Applications*, (Cambridge, MA, 2002), 691-697.
18. Wyckoff, P. TSpaces. *IBM Systems Journal*, *37* (3).