RC-22877 (W0308-061) August 14, 2003
Computer Science

IBM Research Report

Supporting Activity-centric Collaboration Through Peer-to-Peer Shared Objects

Werner Geyer*, Juergen Vogel, Li-Te Cheng*, Michael Muller***

* IBM Research Division
TJ Watson Research Center
One Rogers Street
Cambridge MA, 02142, USA

** University of Mannheim
Lehrstuhl Praktische Informatik IV
L 15, 16
68131 Mannheim, Germany

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. , payment of royalties). Copies may be requested from IBM T. J. Watson Research Center ,
P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Supporting Activity-centric Collaboration through Peer-to-Peer Shared Objects

Werner Geyer*, Jürgen Vogel**, Li-Te Cheng*, Michael Muller*

*IBM Thomas J. Watson Research Center
One Rogers Street
Cambridge, MA 02142, USA
+1 617 693 4791
{werner.geyer, li-te_cheng, michael_muller}@us.ibm.com

**University of Mannheim
Lehrstuhl Praktische Informatik IV
L 15, 16
68131 Mannheim, Germany
+49 621 181 2615
vogel@informatik.uni-mannheim.de

ABSTRACT
We describe a new collaborative technology that is mid-way between the informality of email and the formality of shared workspaces. Email and other ad hoc collaboration systems are typically lightweight and flexible, but build up an unmanageable clutter of copied objects. At the other extreme, shared workspaces provide formal, structured collaboration, but are too heavyweight for users to set up. To bridge this gap between the ad hoc and formal, this paper introduces the notion of "object-centric sharing", where users collaborate in a lightweight manner but aggregate and organize different types of shared artifacts into semi-structured activities with dynamic membership, hierarchical object relationships, as well as real-time and asynchronous collaboration. We present a working prototype implemented with a replicated peer-to-peer architecture, which we describe in detail, and demonstrate its performance in synchronous and asynchronous modes.

Keywords
Object-centric sharing, replication, synchronization, peer-to-peer, activity-centric collaboration, emerging collaboration.

1. INTRODUCTION
Collaborative processes very often emerge from unstructured ad hoc communication activities to more structured types of formal collaboration [3]. Groupware has focused on the two extremes of this continuum but neglected many of the possible stages in-between. Email at one extreme of this continuum can be considered as today's ultimate ad hoc communication support system. Recent studies indicate that email is the place where collaboration emerges (e.g. [7], [26]). A variety of email uses are reported in the literature such as information management, document management and sharing, task management, and meeting management. Whittaker et al. [26] coined the term "email overload" as the phenomenon of email being used for additional functions other than communicating.

While email is extremely flexible, it also requires the user to do a lot of work, such as manually keeping track of the organizational process; users are mostly left alone with the contextual sense-making of all the information contained in their cluttered inboxes. Despite these drawbacks, people keep on using email instead of managing their collaborative processes with special purpose groupware systems such as shared team workspaces, decision-support systems, or meeting management systems. While these systems provide more structure and richer support for collaboration, people often shy away from using them because email is readily available, always on, often the focus of attention, ad hoc, and does not require tedious set-up procedures.

Little work has been done to offer richer collaboration in email and to help support the progression of collaboration from ad hoc communication to more structured types of collaboration that are already supported in many special purpose groupware systems (see Sections 2.1, 2.2, and 8). We are currently investigating technologies for activity-centric collaboration that can help bridge this gap (see Figure 1).

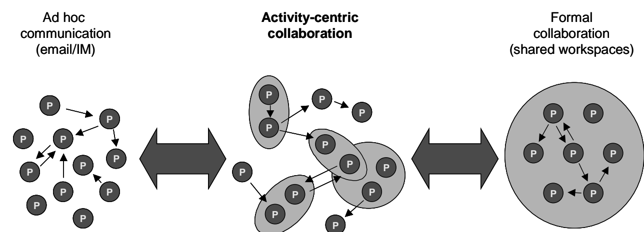


Figure1: From ad hoc communication to formal collaboration

We have designed and built a peer-to-peer prototype system that supports lightweight and ad hoc forms of sharing information, which we believe are key in bridging the gap because they do not overload the user with the overhead of manually creating shared workspaces or setting-up conferences.

This paper introduces the design concept behind our prototype and then focuses on the implementation of this system. In Section 2, we introduce the notion of "object-centric" sharing, which is fundamental to our design. Object-centric sharing allows individuals to aggregate and organize shared artifacts into larger collaborative activities, providing an emerging context that evolves and engages a dynamic group of participants. Section 3 presents the prototype system from a user interface perspective. We illustrate how the prototype can be used within email to engage in lightweight activities. In Sections 4 and 5, we focus on the architecture and implementation of this system. We not only decided to make this system feel peer-to-peer from a user perspective, but also implemented it based on a replicated peer-to-peer architecture. This decision poses various technical challenges. Keeping replicated data consistent in an architecture that supports real-time and asynchronous collaboration at the same time is not trivial, and relatively little research has been done in addressing this problem (e.g., [10]). Our approach enhances a popular consistency algorithm, which had been

originally designed for real-time collaboration. In Sections 6 and 7 we discuss preliminary results as well as trade-offs between centralized and replicated architectures for blended synchronous and asynchronous collaborative systems. Sections 8 and 9 conclude with related work and a summary of this contribution.

2. DESIGN PHILOSOPHY

In email, a collaborative work activity typically begins with a single message that might or might not grow into a collection of related messages including attachments [6]. While email is very good in supporting the ad hoc nature of collaboration and dynamic membership, it is not very good in preserving the context and structure during a conversation; related messages and attached documents are typically scattered or buried in the inbox and they are hard to find. Moreover, email does not support real sharing of content, let alone real-time collaboration. In order to support those aspects of a work activity, people have to "leave their inbox" and use other tools (shared workspaces, conferencing applications etc.) that produce new artifacts that are related to the original work activity. When they do this, they are totally disconnected because those artifacts reside somewhere else on the desktop, in the file system, or on a remote server. The design of our system was mainly driven by the desire to combine the lightweight and ad hoc characteristics of email and the rich support for sharing and structure in shared workspace systems[1].

2.1 Object-centric Sharing

Traditional shared workspaces typically entail a lot of management overhead and are far from being lightweight or ad hoc. They are "place-based", i.e. users first have to create a place, assign access rights, and then put content into that place in order to be able to share it. They are based on the assumption that "the team" already exists and that the purpose of collaboration is well known. However, when collaboration starts off, this is often not the case, and setting up a place can seem to be artificial if not obstructive at this stage. In our research, people often prefer to think in terms of whom to share with and what to share. Also collaboration in these early stages might be very short-term and instantaneous and involve only little amounts of data to be shared, e.g., exchanging one or more files, setting up a meeting agenda with people, or jointly annotating a document. These activities might or might not become part of a larger collaborative work process. However, people usually do not create heavyweight shared workspaces to perform these tasks.

So unlike providing one persistent place for sharing multiple pieces of information, our paradigm is rather "object-centric" or "content-centric," which is very similar to Dourish's [5] notion of "placeless" documents. In this approach, sharing becomes a property of the content itself, i.e. content is collaboration-aware. In this paper, we use the term "shared object" for one shared piece of persistent information. Shared objects support membership, provide object-level awareness, and enable group communication. In other words, they define a set of people who are allowed to access the information, they indicate who is currently looking at the content, and they allow sending or broadcasting of data to members of the object.

[1] Our solution uses shared objects as its reference point. For a contrasting solution that uses email as a reference point, see our discussion of related work in Sections 2.2 and 8.

2.2 Conversational Structure

In our approach, shared objects are building blocks of collaboration. We allow users to combine and aggregate them into hierarchical structures as their collaboration evolves. We call a set of related shared objects an *activity thread*, representing the context of an evolving collaborative activity. Aggregating objects allows people to add structure to their collaboration. We see this structure being defined by their ongoing conversation, i.e. each object added to an existing object can be considered as a "reply" to the previous one. While this approach is similar to threads in email or discussion databases, or thrasks [1], it is much richer because (1) activity threads may contain different types of objects, not only messages, (2) all objects are equal, unlike in email where attachments are subordinates contained in the message, (3) membership is dynamic and may differ within an activity thread from object to object, (4) objects support real-time collaboration and provide rich awareness information.

Unlike shared workspaces, we intentionally do not provide an explicit object as a structural container for a collaborative activity. Each individual shared object can be a container and thus could be considered as a "seed" for collaboration that either decays or grows to more structured forms with the structure being defined as people collaborate.

Our design also deliberately does not make a distinction between asynchronous and synchronous types of collaboration. If other people are present at the time of accessing an object, they can work synchronously, if not, work is asynchronous. From a more technical perspective, objects can be considered as an "infinite", persistent (conferencing) session bounded only by the lifetime of the object. Modifications to the object are broadcast to the members of that object if they are online.

3. USER EXPERIENCE

The user interface to our prototype system is integrated into an email client. The client supports sharing of five types of objects: message, chat transcript, file, annotated screen shot, and to-do item. These objects are managed through a simple tree-like user interface that is contained in the right pane (A) in Figure 2. Each "branch" in that tree represents an activity thread.

Users interact with shared objects by right-clicking on the nodes of the tree which pops up a context menu. Users can create new objects, delete objects, add and remove members etc. Our client supports POP3 email messaging: The upper left pane is the inbox (B) and the lower left pane a message viewer (C). In the following, we use a scenario to illustrate how shared objects as building blocks can be used to collaborate in an activity that starts from an email message. Please note that the activity thread displayed in Figure 2 is just a snapshot at the end of an activity from the perspective of one of the actors (Bob); the thread is built dynamically as the actors collaborate.

Bob is a project lead and he works with Dan on a project on "Casual Displays". Catherine is a web designer in their company who is responsible for the external web site. Bob receives an email from Catherine containing a draft for a project description that she would like to put on their external web site (1). She wants some feedback from Bob. Before getting back to her, Bob wants to discuss the design of that web page with Dan. Instead of forwarding the message to Dan via email, Bob decides to start a new activity by creating a shared object based on this message. He right-clicks on the original message in his inbox, selects "share", enters Dan's email address, and hits "Share". A

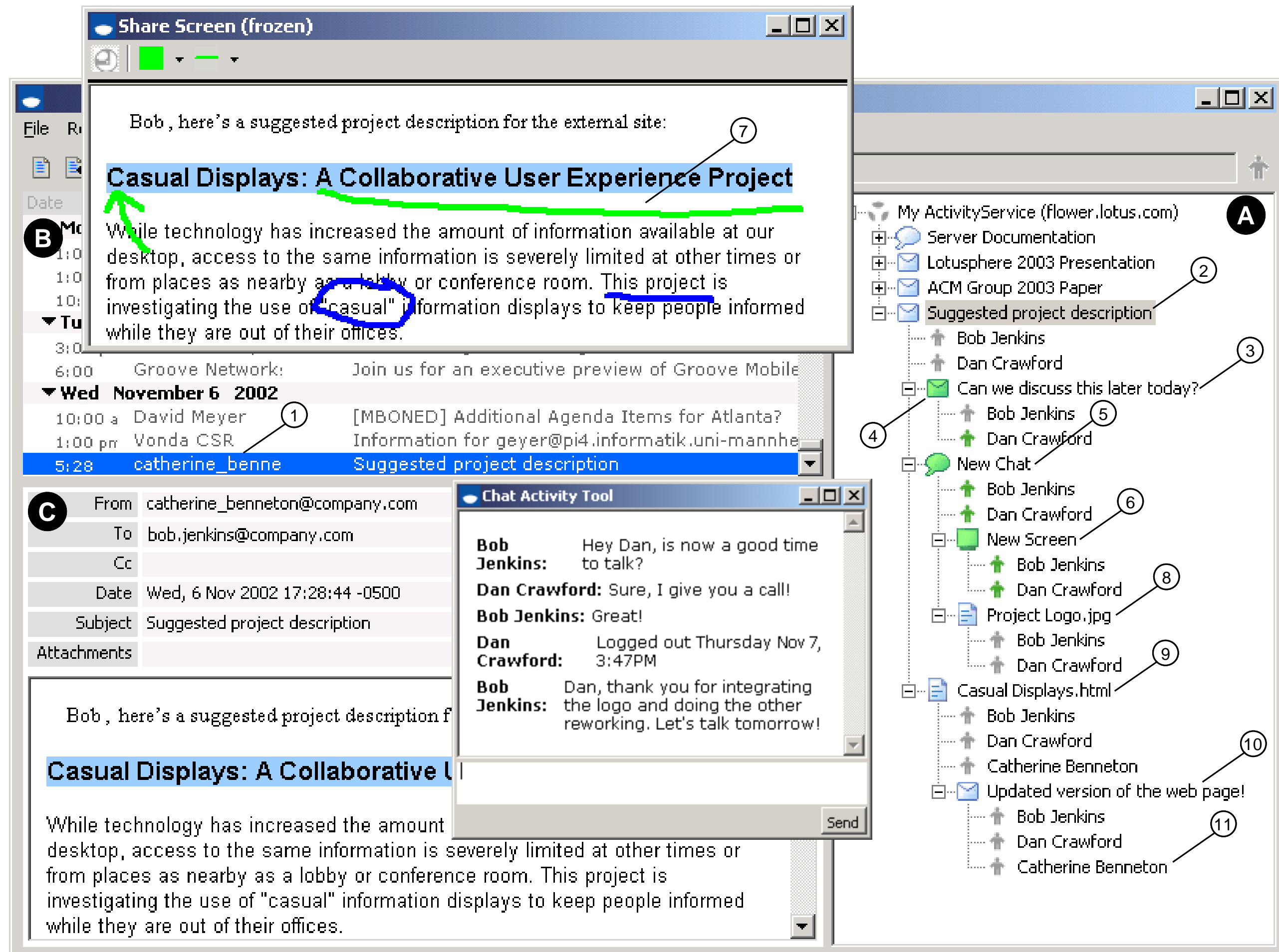


Figure 2: Prototype user interface, (A) Activity Thread Pane, (B) Email Inbox Pane, (C) Email Viewer Pane

new shared message object (with Bob and Dan as members) shows up in Bob's activity tree in the right window pane (2). Bob right-clicks on the shared object and adds a new shared message to the initial one, because he wants to let Dan know that he would like to discuss this with him. Bob's message shows up as a reply to the initial message similarly to a newsgroup thread (3).

A few hours later, Dan returns to his desktop, which is running the client, and notices Bob's newly created shared messages. He opens one message and while he is reading it, Bob sees that Dan is looking at the messages because the shared object is lit green along with Dan's name underneath the object (4). Bob takes this as an opportunity to begin a discussion with Dan within the context of the shared object. He right-clicks on the initial message and adds a chat object to this activity (5). A chat window pops up on Dan's desktop and they chat. In their chat conversation, Bob and Dan continue talking about the web page over the phone. At some point during the discussion, Bob wants to show directly how to change the web page. He right-clicks on the chat object in his activity tree and adds a shared screen object (6). A transparent window allows Bob to select and "screen scrape" any region on his desktop. He freezes the transparent window over Catherine's draft web page. The screen shot pops up on Dan's desktop. Bob and Dan begin annotating the web page in real-time like a shared whiteboard (7). As they discuss a few changes, Bob is asking Dan to integrate a project logo into the web page. Dan agrees but is pressured now to run to another meeting. He says good-bye to Bob and tells him that he will check with him next day. Dan closes all his windows and as he leaves, his name turns gray throughout all of his shared objects displayed on Bob's client.

Now alone, Bob continues annotating the web page. He also types in a few lines for Dan in the chat window before closing it. He then right clicks on the chat object and creates a new shared file object. He picks the logo file from his local file system and the file object becomes part of Bob's and Dan's activity thread (8). Bob closes all windows and leaves. Next morning when Dan returns to his office, he finds Bob's additional annotations, his chat message, and the project logo file. He starts working on the web page and few hours later, he puts the reworked page into the activity thread as a shared file object (9) and adds a message with some comments (10). He also shares these two objects with Catherine (11) so that she can download and deploy the newly revised web page and logo.

This scenario demonstrates how our prototype moves 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 the conversational context and awareness for an emerging collaboration; it allows aggregating a mix of different object types.

4. SYSTEM ARCHITECTURE

The design philosophy and the envisioned use of the system contributed to our decision to implement our prototype as a peer-to-peer system. In particular, the high administrative cost of centralized shared workspace systems was a major factor in this decision. We wanted users to be able to create shared objects on the fly in order to facilitate instantaneous and effortless collaboration. Giving full control to the user implies that the system should function without any additional infrastructure.

Another major requirement is that users be able to collaborate both synchronously and asynchronously. Asynchronous work may take place while people are online but also offline when they are disconnected from the network. To provide offline access at any time, shared objects need not only be persistent but to reside on the user's local machine (desktop, laptop, or PDA). In order to synchronize local replicas, the system must provide appropriate communication and consistency control mechanisms. Since the membership of shared objects can be highly dynamic, the system also has to support late-joining users [25]. Besides ubiquitous access to data, replication helps to achieve good responsiveness.

Finally, object-centric sharing as described in Section 2 implies that membership management occurs individually for each shared object. This fine-grained access to and control of shared objects might entail a scalability penalty in a server-based system. A replicated system scales better with respect to the number of shared objects and the number of users.

Out of these considerations, we opted against a client-server solution and decided to build a peer-to-peer system where each user runs an equal instance of the application locally. Figure 3 shows three applications instances (A, B, and C).

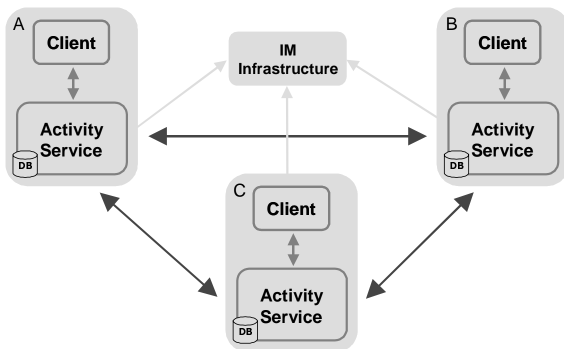


Figure 3: System architecture

Each local instance consists of a client component that provides the user interface to the system (see description in Section 3) and a service component, called ActivityService, that maintains a copy of all shared objects that are relevant to the local user. Peer discovery is accomplished by leveraging an existing instant messaging infrastructure (as discussed in section 4.2). We use a local database in each application instance to store and access shared objects. Changes to the set of shared objects (e.g., by creating new objects or modifying existing shared objects) must be communicated to all peers that participate in an activity. In the following sections of this paper, we refer to the set of shared objects and their properties as the *state* of the application, and we denote states and state changes that are distributed to the set of users as *operations*.

From the user's perspective, this peer-to-peer architecture means that, apart from acquiring and running the application, no extra steps are necessary before new shared objects can be created or existing shared objects can be accessed. New peers can be invited and integrated easily, for example, by an email referencing a web link that automatically installs the application.

4.1 Communication Protocols

The application-level protocol description for the distribution of states and state changes among the peers is based on XML, mainly to allow rapid development and easy debugging. Preliminary tests with our prototype have shown that the resulting performance is sufficient, but should the need arise we plan to switch to a binary protocol.

Since objects can be shared by an arbitrary number of users, application data usually needs to be delivered to more than one destination. Thus, the system has to employ a group communication protocol. We opted against IP multicast due to its insufficient deployment [4] and because it would require the use of UDP as a transport protocol, which is blocked by firewalls in many organizations. Instead, a sender delivers application data directly via TCP to each receiver, forming a star-shaped distribution topology. Since we expect groups for most activities to be small (i.e., up to 10 participants), the network overhead imposed by this approach seems to be acceptable. An alternative would be to use an application-level multicast protocol such as Narada [4] which remains an issue for future work.

4.2 Peer Discovery

Building the application's communication on top of point-to-point unicast connections means that a sender has to contact each receiver individually. Therefore, a peer needs to know the IP addresses of all the peers it is collaborating with. Since our prototype is integrated into an email client, the natural contact information of a peer is its user's email address. This has to be mapped to the IP address of the corresponding user's computer. The address resolution is a dynamic process because an IP address might change when users connect via dial-up, work on different computers, or use dynamic IP.

To allow a peer to connect to the system for the first time, we use the existing instant messaging infrastructure in a company to resolve email addresses (see Figure 3). Each peer connects to its user's corresponding instant messaging server. Each peer contacting the server also leaves its own address information — i.e. the IM infrastructure serves as a means for peer discovery and address resolution. All addresses resolved by a peer are saved in a local address table together with the time the information was obtained. This address table is persistent and used on the first attempt to establish a connection to another peer. Should this fail, the instant messaging server is contacted to inquire whether the peer's IP address has changed. Once a peer makes contact to other peers, they can exchange addresses to complement their local address tables with new entries and to exchange outdated addresses. This way communication with the address server can be limited and the peer-to-peer system will be able to function for some time in case the address server is not reachable.

We are not focusing on peer discovery protocols in this paper. The above mechanisms could be also easily replaced with existing peer discovery protocols such as Gnutella [16] and JXTA [21].

Another alternative would be to make use of the email integration and exchange IP addresses with special emails that are filtered by the email client. The advantage of this approach would be that no additional infrastructure for address resolution is required. Its realization is an issue for future work.

5. CONSISTENCY CONTROL

The replication of the application's state as described in Section 4 requires explicit mechanisms to keep all state copies consistent. Much research has been done in keeping the state of synchronous multi-user applications such as whiteboards, shared editors etc. consistent. Likewise our prototype requires consistency mechanisms when people are working on shared objects at the same time. However, by design our system also supports offline use when people are disconnected from the network and people who are online are able to share objects with others who are currently offline. Little work has been done on algorithms that support consistency in blended synchronous and asynchronous collaborative applications. We have chosen and modified an existing consistency mechanism for synchronous collaboration so that it also supports asynchronous collaboration. In the following, we first describe consistency control in the "ideal" case when everyone is online before we cover the asynchronous case.

5.1 Synchronous Collaboration

Consider the scenario described in Section 3. Let us assume that Bob decides to change the name of the shared screen object that he and Dan were annotating to "project homepage" (see (7) in Figure 2). In order to execute this local state change in Dan's application, Bob's application needs to propagate the pertinent information. But since this transmission is subject to network delay, Dan could also change the object's name to "first draft" in the brief time span before Bob's update is received. In this case, Bob's and Dan's changes are *concurrent*, and they *conflict* since they target the same aspect of the state. Without further actions, the name of Bob's activity would be "first draft" and that of Dan "project homepage", meaning that the object's state is *inconsistent*. To prevent this from happening, the application needs to employ an appropriate *concurrency control* mechanism.

To be more specific, there are two consistency criteria that the application should observe: *causality* and *convergence* [9]. Causality means that an operation O_b that is issued at site i after another operation O_a was executed at i needs to be executed after O_a at all sites, so that the cause is always visible before the effect. For example, the shared screen object has to be created before its name can be changed. Convergence demands that the state of all peers is identical after the same set of operations $\{O_i\}$ was executed. This means in our example that Bob and Dan should see the same name of the screen activity.

We decided to use serialization [23] for establishing causality and convergence. The basic idea is to execute a set of operations that targets a certain object in the same order at all sites. As a prerequisite, an appropriate ordering relation has to be defined. Possible ordering relations are timestamps [19] or state vectors [15]. When using timestamps, operations are ordered by their assigned execution time. Timestamp ordering can be applied to all types of applications, including continuous. However, it requires that the clocks at all sites are synchronized, which increases the administrative overhead and creates dependencies on a time synchronization infrastructure. Hence, we decided to use state vectors to order operations in our prototype. Moreover, this seemed sufficient since shared objects in our system currently do not support time-based state changes.

A state vector SV is a set of tuples (i, SN_i), $i = 1,..,n$, where i denotes a certain peer, n is the number of peers, and SN_i is the sequence number of peer i. Whenever i issues an operation O, SN_i is incremented by 1 and the new SV is assigned to O as well as to the state that results after executing O. We define $SV[i] := SN_i$. With the help of state vectors, causality can be achieved as follows [23]: Let SV_O be the state vector of an operation O issued at site a and SV_b the state vector at site b at the time O is received. Then O can be executed at site b when (1) $SV_O[a] = SV_b[a] + 1$, and (2) $SV_O[i] \leq SV_b[i]$ for all peers $i \neq a$. This means that prior to the execution of O all other operations that causally precede O have been received and executed. If this is the case, we call O *causally ready*. But if O violates this rule it needs to be buffered until it is causally ready, i.e., until all necessary preceding operations have arrived and have been executed.

Convergence can be achieved by applying the following ordering relation to all operations that are causally ready [23]: Let O_a and O_b be two operations generated at sites a and b, SV_a the state vector of O_a and SV_b the state vector of O_b, and $sum(SV) := \sum SV[i]$. Then $O_a < O_b$, if (1) $sum(SV_a) < sum(SV_b)$, or (2) $sum(SV_a) = sum(SV_b)$ and $a < b$.

In the following, we denote the set of operations that was received by a peer and executed in the order defined above as *operations history*. Due to the propagation delay of the network, it might happen that an operation O_a that should have been executed before an operation O_b according to the ordering relation is received only after O_b has been executed, i.e., O_a would not be the last operation when sorted into the history. This means that applying O_a to the current state would cause an inconsistency. Thus, a repair mechanism is required that restores the correct order of operations. *Timewarp* [19] is such an algorithm and works as follows: Each peer saves for each shared object the history of all local and remote operations. Moreover, snapshots of the current state are added periodically to the history. Let us assume that an operation O_a is received out of order. First, it is inserted into the history of the target object in the correct order. Then the application's state is set back to the last state saved before O_a should have been executed, and all states that are newer are removed from the history. After that, all operations following O_a are executed in a fast-forward mode until the end of the history is reached. To avoid confusion, only the repaired final state of the application should be visible for the user.

The advantages of the timewarp algorithm are that it functions exclusively on local information and does not require additional communication among peers, it is robust and scalable, it is applicable to all peer-to-peer applications, and the operations history can be reused for other purposes such as versioning and local recording. One major drawback is the memory usage of the operations history which is determined to a large part by the frequency of state snapshots. While a low frequency saves memory, it increases the average processing time for the execution of a timewarp. From our experience gained with the prototype implementation, we opted to save a state snapshot every 10-15 operations, depending on the shared object.

The size of the operations history can be limited by analyzing the information included in the state vector SV_O of an operation O issued by peer j and received by i: $SV_O[k]$ gives the sequence number of the last operation that j received from k, i.e., j implicitly acknowledges all older operations of k. Under the

condition that O is causally ready, there can be no timewarp triggered by operations from j that are concurrent to operations from k with $SN_k \leq SV_O[k]$. This means, that from j's perspective, i can remove all operations of k with $SN_k \leq SV_O[k]$ from its history. Once i has received similar acknowledgements from all peers, old operations can be cleared. This process can be sped up by periodically exchanging status messages with the current state vectors of a peer.

Under the unlikely event that a new peer late-joins an ongoing collaboration on a shared object, obtains the current state, and issues a state change concurrently to the operation of another peer, the aforementioned improvement might fail since peers clearing their history have no knowledge about the late-joining peer. This can be prevented by keeping outdated operations for some extra time span or by using an additional repair mechanism such as a state request [24].

Another drawback of the timewarp algorithm is the processing time to recalculate the application's state. But since each shared object has its own operations history and since states are inserted frequently into the history, the time required for a timewarp is usually below 40 ms on an AMD 2.0 GHz PC, which is not noticeable for the user. Also, as found in [24], the number of timewarps can be reduced dramatically by analyzing concurrent actions.

The actual effect of a timewarp might be disturbing for a user: It is possible that the recalculated state differs to a high degree from the object's state that was visible before, e.g., when objects are deleted. In this case, an application should provide the user with information about the cause of the visual artifact, and highlight the objects and the users involved [24].

5.2 Asynchronous Collaboration

A key aspect of our prototype is that shared objects can be accessed by users anytime and independently of others. They are persistent even when all peers are offline, and they facilitate synchronous as well as asynchronous collaboration. The latter means for the scenario described in Section 3 that Bob can access and manipulate data of the shared screen object (see (7) in Figure 2) even when Dan is offline and his ActivityService is not running[2]. This raises the question how Dan's application can acquire the information that was missed once it is active again so that it possesses the current state and Dan is able to continue the collaboration. For easier discussion, we denote the process of reconnecting after operations have been missed as *late-join*, the peer that missed operations as *late-join client*, and any peer that provides information to the late-join client as *late-join server* [25].

One design option is whether the late-join client should be provided with all operations missed or, alternatively, with a copy of the current state. The latter option has the advantage that a state transmission is more efficient in terms of data volume and processing time whenever the number of missed state changes is rather high (depending on the nature of the object's state and the state changes). But at the same time, this approach requires additional measures to ensure the consistency of the late-join client's state [24]: The late-join server providing the object's state cannot guarantee its consistency since concurrent operations that are not included yet could be under way. Thus, we chose the first option and provide all missed operations. This is more robust since the late-join client will possess a complete operations history under the condition that all missing operations will be delivered eventually. The late-join client is then able to ensure the consistency of the object's state as described in Section 5.1.

Other design options determine if the data transmission is initiated by the late-join client or the server, and when the transmission takes place. Initiating and controlling the transmission by the late-join client is the obvious choice since the client has to ensure the consistency of its own state. When starting the application instance, the client therefore contacts all peers with whom it collaborates one by one and hands over the current state vectors of its shared objects. Comparing those state vectors to its own, a peer can decide which operations from its history need to be sent to the late-joining peer (if any). Thus, the late-join client can obtain missed state changes from any peer that participates in a shared object, not only from the original source of an operation. This is especially useful when that source is offline at the time the late-join client connects. Additionally, it increases the robustness of the system and spreads the burden of initializing clients to all peers. To further increase the probability that all missed operations are received, information about the current state of objects is also exchanged whenever a user joins a shared object.

However, this approach succeeds only if a peer that was already participating in a shared object missed some state changes. But another possibility is that a new shared object is created or that a new member is invited to join an existing object (as in step (1) of the scenario in Section 3). Notice that either the peer creating a new shared object or the peer receiving a new shared object could be offline. So when reconnecting, the late-join client does not know about the new shared object and can therefore not ask for the transmission of missed operations. Even worse, it might be that the late-join client does not know the other peers that are members of this shared object if they did not collaborate before. Thus, in case a peer misses the creation of an object or the invitation to an existing object, the responsibility for the retransmission must be with the originator of that action. Should both peers connect for the transmission of any other data, the missed operations can be delivered to the late-join client. The late-join server also tries to contact the client periodically (depending on the current application and network load).

As in the previous case, the probability for quickly updating the late-join client can be increased by placing the responsibility for updating the late-join client not solely on the original source but on all members of a shared object. The more members a shared object has, the higher the likelihood is for a quick update. For this purpose, the peer that created an object or invited new members to an object informs all available peer members that the relevant operations could not be delivered to one or more peers. All peers are assigned the task to try to transmit the operations to the late-join client either by making regular contact or by probing periodically. They stop as soon as they receive a state vector from the client that indicates that it possesses the required information. In case the late-join client receives the same operations more than once, it simply ignores them.

In summary, our approach to achieve a consistent application state (also in case of asynchronous/offline use) is based on the retransmission of missed operations, accomplished by the aforementioned mechanisms in our prototype. Retransmissions

[2] In case Dan is not currently working on a shared object but his application is running, no additional actions are required since Dan's ActivityService still receives all state updates.

can be initiated by the late-joining peer, by the peer who was the source for an operation, or by other peer members of a shared object. All operations received by the late-joining peer are then executed in the order defined in Section 5.1. They might trigger a timewarp if necessary so that consistency can be achieved.

6. SIMULATION RESULTS

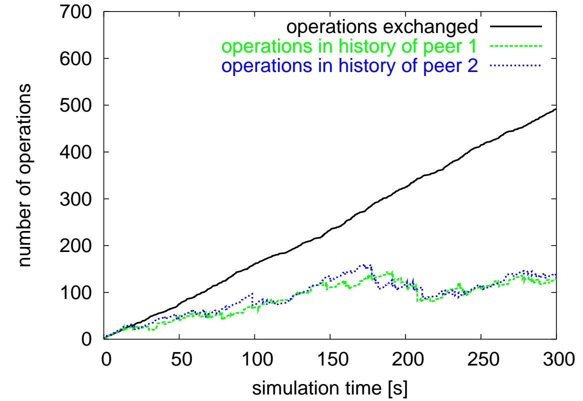
In order to evaluate the correctness and the performance of the algorithms described above, we simulated different scenarios for the synchronous and asynchronous collaboration of two and three peers respectively. In each scenario we randomly reproduce the activities of a typical work week with five days. During one simulated workday, the following actions take place on average: A total of three shared objects are created, each peer views the data of an object fifteen times (i.e., opens and closes the view window of an object fifteen times), and each peer modifies the state of an object ten times. The simulated scenarios differ in respect to the time span that each peer is working either online or offline. For easier handling, we simulated each workday in 60 seconds. Please note that this increases the probability for a timewarp when peers work synchronously. Because of the limited number of operations in our scenarios, we insert a state snapshot every 5 operations into the history.

6.1 Results for the Timewarp Algorithm

When all peers are collaborating synchronously over the entire simulation time, a timewarp can happen only under the rare condition that two or more operations targeting the same shared object are issued concurrently, i.e., within the time span that is necessary to propagate these operations. For a scenario with two peers, a total number of five timewarps was triggered with an average duration of 26 ms and an average number of 6.4 operations that needed to be executed in order to regain a consistent state[3]. When both peers are working online for only 80% of the simulated time, a timewarp becomes much more likely since concurrent actions can now happen over the whole time span where at least one peer is offline. Consequently, a total number of 27 timewarps occurred that took an average execution time of 28 ms and an average sequence of 6.7 operations to rebuild the application's state. When the time that peers spend online is reduced further to 50%, 118 timewarps happened with an average time of 32 ms and 13.2 operations to be executed. In the last scenario, the two peers worked synchronously only for 20% of the simulated time. Since in this case many shared objects and subsequent actions are actually created when being offline, the total number of timewarps decreases to 44 with an average duration of 30 ms and 9.5 operations in a timewarp sequence. In all cases, the average time to execute a single timewarp is below 32 ms which is not noticeable for the user.

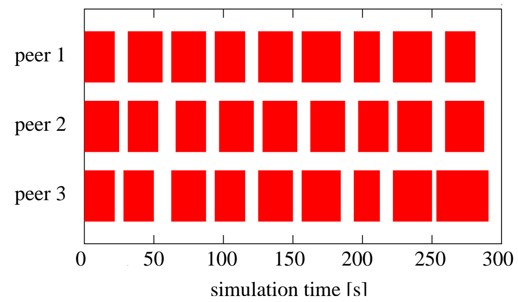
The effect of the algorithm to reduce the size of the operations history can be seen in Figure 4, which depicts the total number of operations exchanged in comparison with the actual size of the history of the two peers, assuming an online time of 80%. In this scenario, the algorithm is able to limit the size of the histories to approximately 25% of the total number of exchanged operations. For the other scenarios, this number lies between 20% and 40%.

[3] Please note that the sequence of operations to be executed in the case of a timewarp also includes operations that were not causally ready before.

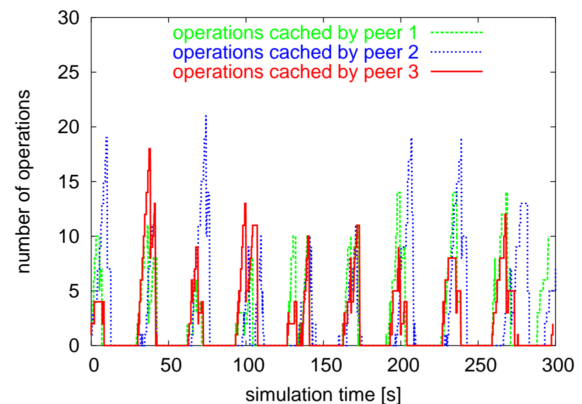


6.2 Results for the Delivery of Late-Join Data

While a peer is working offline, operations originating from or targeting that peer cannot be delivered immediately. Instead, the algorithms described in Section 5.2 transmit the missed operations when the peer reconnects. Figure 5 shows the times peers are collaborating synchronously in the scenario where the three peers spend 80% of the simulation time online. The number of



operations that actually need to be delivered belatedly because the receivers are unreachable is depicted in Figure 6. The late-join algorithm manages to update peers as soon as they are online again. The curves include those operations that are cached by all receivers that were online at the time they were issued (see Section 5.2). On average, peer 1 stores 5.3 operations (peer 2: 7.2, peer 3: 5.3) for the other peers, and it takes about 3.9 s (peer 2: 4.4 s, peer 3: 4.0 s) between generating and delivering a certain



operation[4]. These numbers increase considerably for the scenario where peers are online for 50% of the simulation time: Peer 1 caches on average 25.4 operations (peer 2: 15.3, peer 3: 22.5) and missed operations are transmitted 22.7 s (peer 2: 13.5 s, peer 3: 20.5 s) after they were issued. In the last scenario where peers are online for only 20% of the simulation time, the first peer meets the other peers only rarely. Consequently, peer 1 stores on average 107.4 operations for 60.6 s (peer 2: 80.4 operations for 22.8 s, peer 3: 66.4 for 27.6 s). These numbers show that the state of a shared object might diverge to a considerable degree when peers are working offline for a longer period of time. The algorithm for the distributed caching of missed operations most likely will alleviate this problem, if the number of members of a shared object increases. However, its performance also depends on the work patterns of the members of a shared object.

7. LESSONS LEARNED AND DISCUSSION

Our design philosophy and usage scenarios were a major factor in the decision of building a peer-to-peer system that would function without much administrative overhead. The most challenging aspect of this system is to make it work for asynchronous/offline collaboration as well as for real-time collaboration. Our approach builds upon an existing consistency mechanism for synchronous collaboration. In Section 5.2, we discussed several strategies that we implemented to make sure that this algorithm would also work in case of asynchronous/offline work. They are based on recovering the operations history after periods of offline/ asynchronous work. While our simulation results in Section 6 indicate that our system design works and performs adequately, this approach has also some shortcomings.

The time for reaching a consistent state after periods of offline use depends very much on the user behavior. In the worst case, local states of a shared objects might diverge for a very long period of time, e.g. if two users are alternating between being offline and online. Each user works on a state that is not the current state and then, after eventually merging the operations history, they might see an unexpected result. This reflects a major problem of this approach: While the state is now consistent for both users, the machine cannot guess what their actual intention was when they were independently modifying the shared objects. The algorithm only makes sure that both can see the same end result. However, for the user it is difficult to understand how the resulting state of the object came to be. We believe that it is crucial to provide some mechanisms that assist the user in this task. For example, the application could animate parts of the history to visualize the sequence of operations that led to the current state. Also, a user should possibly be able to restore parts of its operations history after consulting with the conflicting user.

There are also two more technical solutions that would alleviate the aforementioned problem: Adding additional servers, which cache operations and update late-joining peers, would help to decrease the time between resynchronizations of states. Whenever peers are connected to the system, their application can send operations to the caching server, if the collaborating peers are currently offline. When the other peers connect, they first contact the caching server to collect missed operations. The drawback of this approach is that it requires additional infrastructure that needs to be maintained. Another technical approach could be to put more semantics into the consistency algorithm itself. The ordering of operations in state vectors is based on sequence numbers, i.e. when people work offline for long periods of time, the system only looks at the sequence numbers to restore a consistent state. But the sequence numbers do not reflect when operations actually took place. If two users work offline at different times, ordering of operations could be prioritized by the recency of the state change, which would probably help users in better understanding the resulting state after merging the operation histories. This could be achieved either by using globally synchronized clocks as a means for ordering (which again requires infrastructure) or, more elegantly, by using a modified state vector scheme that incorporates local time into the numbering.

It is also noteworthy that when we started this project, we were not anticipating the complexity of such a peer-to-peer system despite the fact that some of the authors of this paper had several years of experience in developing distributed systems. The implementation of only the consistency mechanisms took three full person months. And, before starting to work on the peer-to-peer aspects, we already had a server-based prototype available[5].

When building a peer-to-peer system, the advantages of this approach have to be carefully traded off against the benefits that a server-based solution offers. We believe that there is no definite answer to the question peer-to-peer or server-based. The design philosophy and the user experience of our system require offline use. Hence, we need replication of data to the local machine. However, the aspect of offline use would also greatly benefit from having a server that helps synchronize divergent copies of the distributed state as discussed above. We are currently investigating hybrid architectures that use servers for resynchronization, caching, address resolution, and other useful more lightweight services. While they hold a master copy of the state, local applications would still communicate directly peer-to-peer. In this topology, the server would be just another peer with a different role. The presence of a server facilitates synchronization. However, please note that even with a server, there is still a need for consistency algorithms like the one described in Section 5. When working offline, local replicas and operations need to be merged with the master copy on the server. With some modifications, the algorithms described earlier in this paper could be also used in a hybrid architecture. We are aware that a hybrid architecture again comes with the problem of an additional infrastructure that needs to be introduced and accepted within an organization, which may take a long time. In the meantime pure peer-to-peer systems help paving the road to get there.

We believe that ultimately the user experience has to be peer-to-peer. Email is actually a very good example for such a hybrid system that feels peer-to-peer from a user's perspective but is implemented through a network of mail servers (that talk peer-to-peer to one another). Most email clients today also support offline use by keeping local replicas of their inboxes. However, email does not support shared states and real-time collaboration and thus does not face the consistency challenges like in our system.

This paper has mainly focused on the technical aspects of a system that supports object-centric sharing. However, this new paradigm also deserves and needs to be studied from an HCI

[4] Peers are offline at the end of all simulations (see Figure 5) and operations that have not been delivered are not included in the analysis of the delivery time span.

[5] Our prototype is implemented in Java 1.4 using SWT/JFACE user interface widgets from the Eclipse framework [8].

perspective. A usage study is currently underway, and we do not want to anticipate results here. However, there is some preliminary feedback that is worth mentioning. The system was shown to more than 50 people through demos on three consecutive days. We also used the system informally in our own group for a period of six weeks for discussions and bug reporting specific to the system itself.

Many people reported that the notion of activity-centric work is a substantial part of their everyday work practices. They liked the capabilities in our prototype that support this notion, such as aggregating related items and seamlessly moving back and forth between different modes of collaboration. While the design philosophies of object-centric sharing and conversational structure seem to resonate very well, people had some concerns about our user interface design and other shortcomings that we had not addressed then. Our user interface, for example, shows tree-like structures the way they are stored in the system. We envision that there are better ways of presenting and navigating that information. More conversational interfaces that are structured by time could help understanding the history of an activity. People-centered views could help focus on who you want to collaborate with and could display shared objects that relate to a certain user or a group of users. A disadvantage of the tree-like approach is also that this tree-hierarchy is forced on all users, including newcomers who might not understand how to navigate this structure. Whether this is a serious hindrance or not will be determined by our on-going user evaluation.

Another major issue was the management of shared objects. People reported that they are engaged in numerous lightweight activities every day. They were concerned that the user interface can easily become cluttered and that it would be hard to find information in the tree. More work needs to be done to face this challenge. Activities might grow quickly and evolve to agile, more formal public spaces. We need to provide means so that people can transform these to a shared workspace system. Many activities though will never reach that state. So as a major desired feature of our system, people were asking for some kind of archiving mechanism that helps them getting completed stuff out of their way in the tree, with the option of still having access to the information for later perusal.

8. RELATED WORK

Our notion of activity-centric collaboration based on shared objects has been inspired by previous work in this area. This includes a variety of studies on email and activities, collaborative improvements to email, and "peer-to-peer like" systems featuring replicated or synchronous shared objects.

The use of email has been widely studied and research indicates that email is the place where collaboration emerges (e.g. [7], [6], [18], or [26]). Ducheneaut et al. [6] report on how people manage work activities within their email, confirming Bernstein's [3] description of emerging group processes. Their findings include: the outcome of an activity is often unpredictable, membership in activities is fluid, activities can evolve from the informal to the formal, and late-joiners of activities are poorly supported because they have no access to the history.

A number of recent collaboration systems illustrate concepts similar to activity threads. Rohall et al. [22] show how automatic analysis of subject headers and reply-relationships in different emails can group them into a coherent thread of messages. They also present how carefully designed thread visualizations can reduce inbox clutter. However, their threads are not shared, nor do they reflect any collaborative activity beyond the exchange of messages.

Bellotti et al. [1] introduce the notion of a "thrask" as a means for better organizing email activities. Thrasks are threaded task-oriented collections that contain different types of artifacts such as messages, hyperlinks, and attachments and treat such content at an equal level. Thrasks can be manually created, with contents assigned by users to meaningful activities. While all of these attributes are similar to our use of activity threads, thrasks are not shared: they are private collections of related artifacts, whose organization are specific only to the owner of the thrasks, and lack any awareness of how others are manipulating the artifacts.

Along the same lines, Kaptelinin [13] presents a system that helps users organize resources into higher-level activities ("project-related pools"). The system attempts to include not just email, but also any desktop application. It allows manual addition of resources of any type but also monitors user activities and automatically adds resources to the current activity. However, the system has been designed for personal information management only, not complete synchronous/asynchronous sharing and awareness. Rohall's and Kaptelinin's work demonstrate semi-automatic management of activities and their artifacts. This is something we should investigate in our future work to help make managing activity threads more lightweight.

Whittaker et al. [27] present five design criteria for lightweight, informal interactions, which influenced the design of our prototype: conversational tracking, rapid connection, ability to leave a message, context management, and shared real-time objects. Their work also emphasizes the importance of work-related artifacts to help manage the history and context of intermittent interactions: "Task-related documents can serve to hold the context of multiple ongoing conversations". The system they present mainly focuses on threaded IM-like conversations using "sticky notes" on the desktop.

A variety of collaborative systems implement replicated shared objects and "collaborative building blocks" similar to our prototype. One example is Groove [12], a peer-to-peer system which features a large suite of collaborative tools (chat, calendar, whiteboard, etc.) and email integration. However, Groove is workspace-centric (although creation of shared workspaces is quick), and collaboration is centered on the tools placed in the workspace (e.g., all shared files appear in the shared files tool), except for persistent chat which appears across all tools. In contrast, our system focuses collaboration around artifacts, which can be organized in a single connected hierarchy of different types. While Groove's design philosophy is different from ours, the architecture is very alike. Their system has to deal with problems similar to the ones described in Section 5. However, we have no technical information to compare the two approaches.

Another example is Eudora's Sharing Protocol [11], which offers a replicated shared file architecture completely based on email and leverages special MIME headers. Kubi Spaces [14] also offers replicated shared objects, with a richer suite of object types, including to-dos, contacts, and timelines. Microsoft Outlook [20] has a server-based "shared email folder" capability. Lotus Notes [17] offers a replicated object architecture, although typically email objects are copies, not shared amongst different users. A powerful benefit of these "shared email solutions" is that no additional infrastructure beyond email is needed. However, synchronization only occurs on a triggered refresh interval and

depends on a non-real-time message delivery between intermediary email servers. Thus, collaboration is entirely asynchronous (e.g., users cannot work simultaneously on a whiteboard in real-time) with no real-time presence awareness.

9. CONCLUSION

We described a new collaboration technology that targets lightweight collaborative activities sitting mid-way between ad hoc communication in email and more formal collaboration in shared workspace systems. As a major new design principle, we introduced the notion of object-centric sharing, which allows people to aggregate and organize shared objects into activity threads, providing an emerging context that evolves and engages a dynamic group of participants. Being "placeless," our approach imposes little overhead and allows for lightweight, ad hoc collaboration. We are currently conducting a larger user study to better understand the usefulness and usability of this new approach. As part of our future work, we are also looking into user interface improvements, better ways of managing activity threads, and new shared object types that introduce more structure and help evolving activity threads to shared workspaces.

This work focused on describing technical aspects and implementation challenges of a peer-to-peer prototype system supporting the notion of object-centric sharing. As an implication of our design philosophy we need to maintain consistency in a blended synchronous and asynchronous collaborative system. Our approach enhances a popular consistency algorithm, which had been originally designed for real-time collaboration. Our preliminary simulation results indicate that this approach works well. However, a major difficulty is that during long phases of asynchronous work the application state might diverge significantly. To alleviate this problem, we are currently looking into improved versions of our consistency algorithm and we are also investigating new hybrid architectures that use lightweight caching servers to make the system more robust.

10. REFERENCES

[1] Bellotti, V., Ducheneaut, N., Howard, M., Smith, I., "Taking Email to Task: The Design and Evaluation of a Task Management Centered Email Tool," in: *Proc. ACM CHI 2003*, Ft. Lauderdale, Florida, 2003, 345-352.

[2] Bellotti, V., Smith, I., "Informing the Design of an Information Management System with Iterative Fieldwork," in: *Proc. ACM DIS'2000*, Brooklyn, New York, 2000, 227-238.

[3] Bernstein, A., "How Can Cooperative Work Tools Support Dynamic Group Processes? Bridging the Specifity Frontier," in: *Proc. ACM CSCW'00*, 279-288.

[4] Chu, Y., Rao, S.G., Seshan, S., Zhang, H., "Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture", in: *Proc. ACM SIGCOMM 2001,* San Diego, CA, August 2001.

[5] Dourish, P., Edwards, W.K., Lamarca, A., Salisbury, M., "Presto: An Experimental Architecture for Fluid Interactive Document Spaces," in: *ACM Transactions on Computer-Human Interaction*, Vol. 6, No. 2, June 1999, 133-161.

[6] Ducheneaut, N., Bellotti, V., "A Study of Email Work Activities in Three Organizations," working paper, Parc. Inc.

[7] Ducheneaut, N., Bellotti, V., "E-mail as Habitat: An Exploration of Embedded Personal Information Management," in: *ACM interactions*, 9(5), 2001, 30-38.

[8] Eclipse Project, "The Eclipse Platform Subproject", http://www.eclipse.org/platform/index.html

[9] Ellis, C.A., Gibbs, S.J., "Concurrency Control in Groupware Systems", in: *Proc. ACM SIGMOD 1989*, Portland, OR, May 1989, 399- 407.

[10] Erickson, T., Smith, D.N., Kellogg, W.A., Laff, M., Richards, J.T., and Bradner, E., "Socially Translucent Systems: Social Proxies, Persistent Conversation, and the Design of Babble," in: *Proc. ACM CHI 1999*, 1999, 72-79.

[11] Eudora, "Eudora Sharing Protocol (ESP)", http://www.eudora.com/email/features/esp.html

[12] Groove Networks, http://www.groove.net

[13] Kaptelinin, V., "UMEA: Translating Interaction Histories into Project Contexts," in: *Proc. ACM CHI 2003*, Ft. Lauderdale, Florida, 2003, 353-360.

[14] Kubi Software, http://www.kubisoft.com

[15] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System", in *Communications of the ACM*, 21(7), July 1978.

[16] LimeWire and Gnutella, http://www.limewire.com

[17] Lotus Notes, http://www.lotus.com/notes

[18] Mackay, Wendy E., "More Than Just a Communication System: Diversity in the Use of Electronic Mail," in: *Proc. ACM CSCW'88*, Portland, Oregon, 1988, 344-353.

[19] Mauve, M., "Consistency in Continuous Distributed Interactive Media", in: *Proc. ACM CSCW 2000*, Philadelphia, PA, December 2000.

[20] Microsoft Outlook, http://www.microsoft.com/outlook/

[21] Project JXTA , http://www.jxta.org

[22] Rohall, S. L., Gruen, D., Moody, P., Kellerman, S., "Email Visualizations to Aid Communications," in: *Proc. IEEE InfoVis 2001*, San Diego, CA, October 22-23, 2001.

[23] Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D., "Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems", in: *ACM Transactions on Computer-Human Interaction*, Vol. 5, No., 1, March 1998, 63-108.

[24] Vogel, J., Mauve, M., "Consistency Control for Distributed Interactive Media", in: *Proc. ACM Multimedia 2001*, Ottawa, Canada, September 2001.

[25] Vogel, J., Mauve, M., Geyer, W., Hilt, V., Kuhmuench, C., "A Generic Late Join Service for Distributed Interactive Media", in: *Proc. ACM Multimedia 2000*, Los Angeles, CA, November 2000, 259 – 268.

[26] Whittaker, S., Sidner, C., "Email Overload: Exploring Personal Information Management of Email," in: *Proc. ACM CHI'96*, 1996, 276–283.

[27] Whittaker, S., Swanson, J., Kucan, J., Sidner, C., "TeleNotes: Managing Lightweight Interactions in the desktop," in: *ACM Transactions on Computer-Human Interaction*, Vol. 4, No. 2, June 1997, 137-168.