Dynamic Content and Offline Collaboration in Synchronous Groupware

Mihail Ionescu, Allan Meng Krebs, and Ivan Marsic
Center for Advanced Information Processing (CAIP)
Rutgers — The State University of New Jersey
Piscataway, NJ 08854-8088 USA
+1 732 445 0549
{mihaii, krebs, marsic}@caip.rutgers.edu

Abstract

With the proliferation of mobile computing devices we witness an increasing demand for applications supporting collaboration for wireless mobile devices, where the possibility of temporary losing the connection is considerable. We present here a novel framework that allows users to work offline and then to merge the local data with the global application state when the connection is reestablished. The algorithm is optimal with respect to the amount of data sent through the network and is completely transparent to the user. We implemented two example applications, as Java applications on workstations and as Java spotlets on Palm Pilots, and used them to evaluate the system performance. The results show that that the system is scalable and has good performance.

Keywords

Design of collaborative systems, Mobile and Wireless Collaborative Systems, Intelligent agents in Collaborative Systems.

1 INTRODUCTION

A key component for collaboration is sharing and manipulation of information. Most collaborative applications provide synchronization support, the so-called WYSIWIS (What You See Is What I See) technique. However, with recent anytime-anywhere proliferation of computing technology, the WYSIWIS paradigm cannot meet the requirements of future groupware applications. The users might need efficient collaboration to see the same data in different views or even see different subsets of the same data, depending on their computing capabilities, roles, expertise and current context. Moreover, the subsets can change dynamically at run-time without any prior knowledge. It is also important to support offline working and to merge the offline work with the current state of the application when the client is back online.

In this paper we present a new collaboration paradigm and a novel system, called DISCIPLE. Unlike WYSIWIS, which is screen- or view-centric, our new paradigm is *data-centric*. The participants collaborate on data and exchange data. The data is represented in a generic way and may be visualized differently for different users. Each user can get only subset of shared data by specifying at run-time the *data distribution conditions*. The conditions reflect the user's interests or computing and communication capabilities. The DISCIPLE system is suitable for environments where clients have:

- Slow and/or unreliable connections, e.g., wireless connection to the Internet
- Different display and processing capabilities, e.g., PDAs vs. desktop workstations
- Different needs and/or interests for the data to be shown on the access device.

The above are characteristics of field collaborators working with mobile access terminals, such as military or civilian rescue or disaster relief applications for distributed operations planning. Another example is collaboration of physicians working in rural areas with experts in hospitals.

The paper is organized as follows. We first review related work in this area. Next, we overview the architecture of the DISCIPLE system. We then present the algorithm used for enabling offline working and merging of the states when the client rejoins. System performance is evaluated on two example applications. Finally, we discuss further work and conclude the paper.

2 RELATED WORK

The propagation of data and data modifications in our framework resembles notification servers [3,5]. A notification server informs the clients about the modifications made to the shared data and the clients thus update the local views. However, unlike our system, the clients cannot specify the objects of interest and the system does not support offline collaboration.

Supporting network partitions and merging when the partitions are reconnected is a very active research area in distributed database systems [2]. However, these approaches cannot be used in synchronous groupware due to its specific issues. The concept of transactions and rollback are of no use because of the real-time aspects of synchronous groupware and also because rollback may result in user's confusion.

Many of the mechanisms described in this paper are present in some distributed file systems, such as CODA [6]. CODA supports disconnected users by caching the most used files on which the user can work while disconnected and updates the modifications at reconnection. However, file systems deal with different level of granularity and have different semantics. File systems are simply considered un-typed byte streams, with no meta-information about the structure of files, so the application cannot optimize the reconnection or conflict resolution. In our case optimizations are possible, since the structure of the data is known to the application.

A system for data management support is presented in [4]. The main focus is on asynchronous groupware and integrating awareness support into the system, but it is related with our work as it uses a replicated object store and local caching to support mobile clients. It differs mainly in having an application-dependent distributed object (called Coobject), which manages the data associated with a specific application. Our approach has a more clear separation between data and application, which simplifies development of different applications sharing the same data objects, and also, having more lightweight data objects makes implementation on small devices, like PDA's, easier. [4] does not address the different issues specific for synchronous groupware or heterogeneity of clients.

3 DISCIPLE ARCHITECTURE

The DISCIPLE system is based on a client/server architecture and comprises three main layers: the repository, the basic communication infrastructure and the task-specific application.

3.1 Distributed Repositories

The main role of the repositories is to store shared information. We designed an abstract data type to model the shared objects, called UForm. The UForm consists essentially of a unique identifier and a keyed list of properties. The UForms could be represented in a markup language, such as XML or WML. The important choice we make is that we impose the schema or meta-information on all the information exchanged between the publishers and subscribers and they are allowed to use only this schema. This choice simplifies the design of the conditions and agents while maintaining sufficient expressiveness for the applications of interest.

The global repository is situated at the server site and stores all the shared data objects. In the future we plan to distribute the global repository across different servers. The local repositories store the objects of interest to the local user and allow offline work when the connection is interrupted. They are synchronized with the global repository when the connection between the client and the server becomes alive.

3.2 DISCIPLE Communication Infrastructure

The communication infrastructure, called *collaboration bus*, is based on an intelligent reflection server, which reflects messages (commands) between the clients. The built-in intelligence on the server assures that the commands are only reflected to the clients that are actually interested in the object(s) the command acts upon, thereby minimizing the network traffic.

3.3 Collaborative Session and Application State

We define a collaborative session as a set of participant terminals connected by a communication network. A participant terminal is called a *site* and the condition imposed is that there is one site per user. Any site can and should communicate with any other site. A *configuration* $\Gamma = <S_{\alpha 1}, \dots S_{\alpha n}>$ is defined as a structure that holds the sites which are sharing data, including the server.

Each site hosts an application that collaborates with the applications from remote sites. Each application supports an arbitrary number of objects: $u_1, u_2, \dots u_n$, each object being completely defined by its properties (the objects can have different properties). The relationships between the objects, e.g., "part-of", are also defined as object properties. We define the *application state* as a structure $S = <id, O>$, where *id* is a unique identifier associated with the site (e.g., its IP address) and O is the set of instances of objects $u_1, u_2, \dots u_n$ along with the associated properties.

4 OFFLINE COLLABORATION AND MERGING STATES

A major problem with handheld computers and generally with the devices connected via wireless networks, is that the connections can frequently be temporarily interrupted or broken and resumed later. For collaborative applications that involve field work, it is very important that the users are able to continue working even when their devices are disconnected from the network and then, at reconnecting, their work should be merged with the global state. Ideally, the user should not notice a short-time loss of the connection.

In order to solve this we need a mechanism to detect if the connection was lost and also to detect when the connection is reestablished. We implemented a heartbeat protocol where clients send periodically heartbeat messages to the server. If the server does not reply for a constant number of time intervals, the client assumes that the connection is down. The user can work offline as normal, but we mark all the modified objects (including the newly created ones) as *dirty*. On the server side, if it does not receive the heartbeat message for a constant period of time it can also assume that the connection is down and does not send notification messages to that particular client until it receives a login request. The client then starts sending ping-like packets to detect when the connection to the server is restored. Once the server responds, meaning that the connection is reestablished, the client sends to the server its actual state, as recorded in the local repository. The server resumes the connection, receives the state and executes the algorithm for merging the states. A sketch of the merging algorithm is shown in Listing 1.

Listing 1: The data merging algorithm pseudo-code.

```
UFormArray MergeStates(
   UFormArray clientState, UFormArray serverState
) {
   UForm clientForm, serverForm;
   UFormArray resultState;
   Condition clientCondition;

   resultState.empty();
   clientCondition=clientConnection.getConditionNotification();
   for (int i=0; i<clientState.size(); i++) {
      clientForm= clientState.getAt(i);
      if (clientForm.isDeleted) {
         if (clientForm in serverState) {
            serverForm=serverState.get(clientForm.getId());
            if (!serverForm.isDeleted()) {
               serverForm.isDeleted=true;
            } else {
             sendNotificationMessages(clientForm);
            }
         } else {
            serverState.add(clientForm);
            sendNotificationMessages(clientForm);
         }
         resultState.add(clientForm);
         continue;
      }
      // if it arrives here, it means that clientForm.isDeleted is false
      if (clientForm in serverState) {
         serverForm=serverState.get(clientForm.getId());
         if (clientForm.isDirty()) {
            if (clientForm.lastType < serverForm.lastType) {
               resultState.add(serverForm)
            } else {
               resultState.add(clientForm);
               serverState.updateForm(
                  clientForm().getId(), clientForm);
               sendNotificationMessages(clientForm);
            }
         } else {
            resultState.add(clientForm);
            serverState.add(clientForm);
            sendNotificationMessages(clientForm);
         }
      }
   }
   for (i=0;i<serverState.size();i++) {
      serverForm=serverState.getAt(i);
      if (serverForm.satisfiesCondition(clientCondition)) {
         resultState.add(serverForm);
      }
   }
   return resultState;
}
```

The method `sendNotificationMessages()` sends the specified UForm to all the alive connections that are interested in this object, by evaluating the corresponding data distribution agents.

The algorithm takes as input the client state, $S = <id, O>$, as received by the server after the connection was reestablished and the actual state of the whole configuration as maintained on the server in the global repository. It first inspects all the UForms in the client state. If a UForm has the `deleted` flag set, it checks if the UForm with the same ID is also in the global state. If yes, it tests if the UForm was deleted while the connection was down, and, if so, the delete command is propagated to all the interested clients and the global repository. If the UForm's `deleted` flag is not set, it is checked if that UForm existed or not in the global state before connection reestablishment. If not, it means that it was created while the user worked offline, so the create command is sent to all the interested clients.

If the UForm existed before in the global state, interesting phenomena can arise. Suppose that there are two users, *A* and *B* working on a shared whiteboard. Initially, both users are online and the common state contains a rectangle, *R*. Suppose now that user *A* goes offline and, while offline, it moves the rectangle *R* to position p_1. Meanwhile, user *B* moved the rectangle *R* to position p_2. What should happen when the user *A* goes back online? The algorithm has to decide what state of the object *R* to keep. This problem is similar to conflict resolution in concurrency control algorithms and cannot have a general solution since it depends on the context. One possible solution adopted here is to assign to each user a priority (the lastType field of each UForm keeps the priority of the last user that accessed the object). Then, if the priority of the user *A* is greater than the priority of user *B*, the object will be moved to position p_1 or, if opposite is the case, to position p_2. In the environments where the assignment of priorities is not possible (as in large systems), we assume that each user has the default priority (0), so the algorithm always chooses the server state of the UForm. When the conflict is detected, the users are also informed about it. If the automatic solution provided by our system is not satisfactory, the users can choose another variant. In order to make the user aware of other users actions, we developed in DISCIPLE several types of group-awareness widgets [1]. Telepointers are widgets that allow a given user to track remote users' cursors. In addition, the users can exchange messages, post small notes, and annotate regions of the window. It is also important to note that this paper focuses on support of intermittent links with short-duration loss of connections (on the order of minutes or hours). For long-duration disconnections, there should be devised other mechanisms. Consider, for example, the case when the client with the highest priority goes offline, makes a lot of far-reaching changes and then shuts down for a few days, while all other clients continue working, perhaps on data already deleted on that mobile client. A

possible solution to this would be to progressively decrease with time the priority of that client, or to allow the collaborators to manually decide what version of the objects to keep by presenting all the possible choices.

4.1 Minimizing Network Traffic at Reconnection

When a client rejoins a session it first sends its state to the server. The server computes the merged state and sends it back to the client. However, a more natural solution would be for the client *A* to send only the UForms modified while working offline, and the server to send back to *A* only the UForms modified by other clients while *A* was offline or the UForms modified by the merging algorithm.

On the client site, sending only the UForms modified after the client is notified that the connection is down will not work. For example, suppose that the connection is broken at time t_α but the application detects this only at time $t_{\alpha+\delta}$, where δ is a random value. At rejoining, the client cannot determine what the real value of δ was , so it is unable to decide what UForms to send to the server. To solve this, we developed an acknowledgement-based algorithm. The client assigns a sequence number to each message sent to the server and marks the UForm modified by this message as dirty. Each UForm contains an additional field, `lastMessageID`, which holds the sequence number of the last message that modified this UForm. The server acknowledges the receipt of the messages in the heartbeat that is periodically exchanged between the client and the server. When the client receives the acknowledgement, the dirty flag of the corresponding UForm is cleared if the acknowledgement contains the `lastMessageID`. At rejoining, only the UForms with the dirty flag set are sent to the server. Although it may happen that the server receives UForms that it already received (e.g., if the acknowledgement message has been lost), it is easy to prove that the server will receive all the messages that it has not received while the client was disconnected.

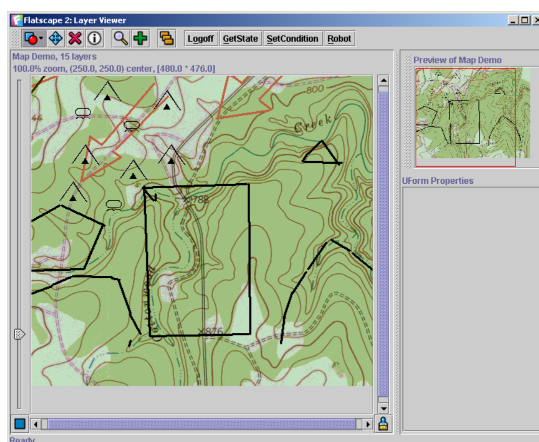
On the server side, the problem is more complex because it deals with many clients. We keep on the server a list of all the messages received by the server, time-stamped at the server site. (The same log is used to implement undo/redo mechanism.) Each client acknowledges in the heartbeat the receipt of messages sent by the server. The server maintains for each client the time when it received the last acknowledgement, t_α. When the server, upon a client rejoining, computes the merged state, only the UForms modified after t_α are considered in the merging process. When the client receives the merged state, it simply concatenates its state with the merged state.

In choosing the heartbeat period, denoted by *T*, one should take into consideration the following aspects. On one hand, *T* should be small in order for the acknowledgement sent by the server to reach the client as soon as possible.In that way, the dirty flag can be removed very quickly thus minimizing the network traffic at reconnection. On the other hand, *T* should not be very small because of the overhead introduced when the quality of the connection is bad. Also, if *T* is small, the server can decide that the connection with a client is down even if the messages are just delayed in the network. In our implementation, *T* is dependent on the quality of the connection and its actual value is negotiated at logon time between the client and the server.

5 IMPLEMENTATION AND PERFORMANCE

We evaluated the performance of the framework both in an environment using standard 10Mbps Ethernet Local Area Network (LAN) and using wireless connection to a wireless ISP provider via a 19Kbps cellular CDPD modem. Two versions of the collaborative map application were used:

Flatscape/Palmscape Map Application. Flatscape and Palmscape are two implementations of a graphical editor targeted for use in collaboration on a situation map. Flatscape is the full-featured implementation, which can run either as an applet or standalone. Palmscape is a limited version running on Palm Pilots Vx. A typical snapshot of collaboration is illustrated in Figure 1.



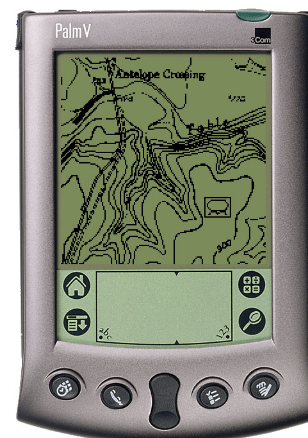


Figure 1: A snapshot of collaboration between a workstation and a Palm.

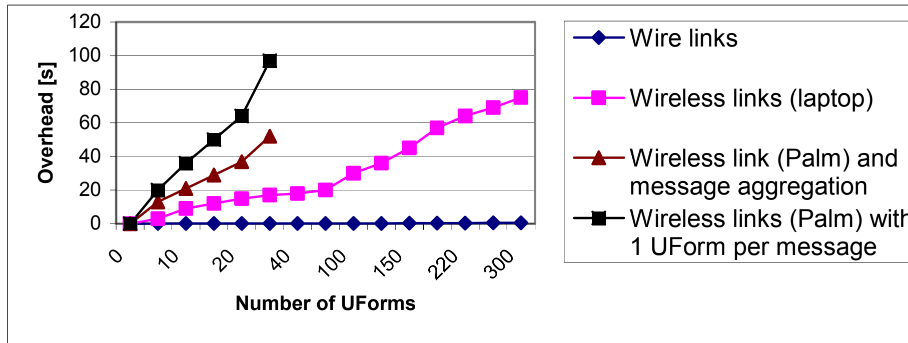


Figure 2: The overhead as a function of the number of UForms.

5.1 Performances

We first measured the latency of updates between the clients using four combinations of PC-to-Palm collaboration. The latency is defined as the time that elapses from the moment the user clicks on the display to create a UForm to the moment the graphical representation of that UForm is shown on a peer client. For each configuration the measurement was repeated 10 times and the mean time and standard deviation were calculated. Table 1 shows the results for the four cases.

	Mean time (sec)	Standard deviation
Palm to PC (wired)	3.57	0.45
PC (wired) to Palm	4.92	1.71
PC (wireless) to PC (wired)	2.00	0.33
PC (wired) to PC (wireless)	1.32	0.15

The measurements show that latency is significantly higher when Palms are involved. The same is true with the synchronization time. This is due to significantly lower processing power and due to the need to convert commands to text string. The difference between Palm-to-PC and PC-to-Palm is probably caused by much slower display updates on the Palm compared to the PC and by the way the service provider for the OmniSky Internet service is handling packets coming to and from the client. The much larger deviation for the PC-to-Palm case perhaps indicates that network latency is the dominant factor.

We next evaluate the overhead introduced to the application at rejoining the session when the states are merged. According to the merging algorithm, all the active clients are notified about the modifications of interest made while the rejoining user was offline. The overhead is presented in Figure 2 as a function of the number of objects (UForms) in the application state created while the client was offline. (This does not depend on the number of objects created before the client went offline.) We implemented an automated robot to randomly generate objects, so we are able to test the system against a large number of objects.

Figure 2 shows that in the wire case the overhead is very low, even for a large number of UForms. Also, the overhead scales very well as the number of UForms is increased. However, for the wireless case, the overhead becomes significant as the number of UForms increases. For the Palm, we could not transmit the entire state in one message because of limited heap size of Java 2ME CLDC for Palms [7]. The problem appears when generating a text string message holding the application state. We first tried sending one UForm per message. However, as the graph shows, this results in very bad performance. Thus, we aggregate a variable number of UForms per message (in our case it was 20). Unfortunately, even in this case we could not work with more than 30 UForms due to our current implementation of the state-to-text conversion algorithm, which again suffers from the heap limitation. Since the targeted disconnection time is short, this may not be a limitation.

Our last measurements are related to the scalability of the system when the number of clients increases. We evaluated the overhead introduced to the application at rejoining the session as a function of the number of active clients. While offline, each client creates a fixed number of UForms, 20 in our measurements. The results show that the overhead does not increase with increasing number of users. We also measured the overhead introduced when multiple clients rejoin concurrently. These measurements (Figure 3) show a good scalability of the system, even

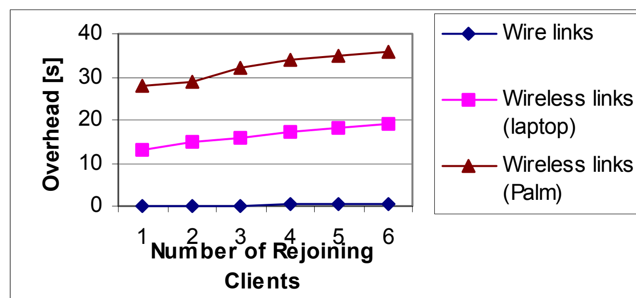


Figure 3: The overhead as a function of the number of clients that concurrently rejoin.

when multiple clients are concurrently rejoining the session. Although we did not perform measurements for a large number of clients, the results are adequate since the number of collaborative sites in typical applications is no greater than six.

6 CONCLUSIONS

In this paper we presented a novel approach to support collaborative applications on mobile devices with wireless network connections. The connection can be lost in any moment, but the user is allowed to work offline and its local application state will be merged with the global state when the connection is reestablished. The algorithm is flexible, independent on the application semantics, optimal with respect to the amount of data sent through the network and completely transparent to the user.

We present two example applications. We are in progress of developing a shared calendar, a text editor, and a resource management application, which we plan to use internally in our group. We will monitor the use of the system as a field trial to obtain information about its usability.

Our continuing work includes the architecture for very small devices (future cell-phones and pagers), distributed global repository implementation, and dynamic adaptation of communication to changing connection quality. The DISCIPLE source code, sample beans, and documentation are freely available at: `http://www.caip.rutgers.edu/discipl e/`

ACKNOWLEDGMENTS

The authors are indebted to the anonymous reviewers, whose comments helped us improve the quality of this paper. The research reported here is supported in part by NSF KDI Contract No. IIS-98-72995, US Army CECOM Contract No. DAAB07-00-D-G505 and by the Rutgers Center for Advanced Information Processing (CAIP).

REFERENCES

1. Dorohonceanu, B., B. Sletterink, and I. Marsic (2000): A novel user interface for group collaboration. *Proceedings of the 33rd Hawaii Int'l Conference on System Sciences (HICSS-33)*.

2. Menasce, A., Popek, G.and Muntz, R. A locking protocol for resource coordination in distributed databases. *ACM Transactions on Database Systems*, 5(2):103-138.

3. Patterson, J. F., Day, M., and Kucan, J. Notification servers for synchronous groupware. *Proc. ACM Conference on Computer-Supported Cooperative Work (CSCW'96)*, Boston, MA, pp.122-129, November 1996.

4. Preguiça, N., Legatheaux Martins, J., Domingos, H., and Duarte, S. Data management support for asynchronous groupware. *Proc. ACM Conference on Computer-Supported Cooperative Work (CSCW'00)*, Philadelphia, PA, pp.69-78, December 2000.

5. Ramduny D., Dix A., Rodden, T. Exploring the design space for notification servers. *Proc. ACM Conference on Computer-Supported Cooperative Work (CSCW'98)*, Seattle, WA, pp.227-235, December 1998.

6. Satyanarayanan, M. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5), May 1990.

7. Sun Microsystems, Inc. Java J2ME Connected Limited Device Configuration (CLDC). Online at: http://www.javasoft.com/products/cldc/