

SYNG: A Middleware for Statefull Groupware in Mobile Environments

Mihail Ionescu and Ivan Marsic
Center for Advanced Information Processing (CAIP)
Rutgers — The State University of New Jersey
Piscataway, NJ 08854-8058 USA
{mihaii,marsic}@caip.rutgers.edu

Abstract— Computer supported collaboration systems, or groupware, are being used more and more in the real life. In the recent years, we are witnessing an increasing demand for supporting such systems in mobile environments. In this paper we address the following questions: “How are the limited and highly dynamic resources of mobile clients, like network connection, energy supply or display size, influencing the design and deployment of groupware systems?” and “What type of policies the users need to specify in order to be able to collaborate in such environments?”. We present a middleware system, called SYNG, that allows a statefull model, suitable for mobile environments. In our system, each user can define a state composed of a number of variables. Examples of such variables include battery usage, quality of network connection, display size, etc. Based on this state, the participant in a collaborative session specifies its own policy for receiving messages from the other participants. Experimental results show good performance and scalability of our approach.

Keywords: *Statefull Groupware, Location aware mobile systems, Software Engineering.*

I. INTRODUCTION

Computer supported collaboration systems, or groupware, are well known these days and are being used more and more in the real life, from the simplest form, like chatting and e-mail, to complex applications that allow geographically dispersed collaborators to edit text documents or to draw on a shared whiteboard. Usually these applications are built using the WYSIWIS (What You See Is What I See) paradigm, in which the actions of one user are instantaneously propagated to all the other participating users, so all of the participating users will have a consistent view of the global application state.

In the recent years, we are witnessing an increasing demand for supporting groupware systems in mobile environments. The presence of limited resources for the mobile clients, as well as the fact that these resources can change frequently due to external reasons impose new challenges for synchronous groupware systems. The availability of many devices (such as HP iPAQ h6315) that are able to take digital pictures and integrate various technologies, such as GPS, 802.11, GPRS is making the use of groupware systems in wireless environments even more appealing.

As also mentioned in [1], the main characteristics of a mobile environment include:

- Dynamic bandwidth and high latency. Wireless networks generally exhibit a variable bandwidth which can decrease or increase almost randomly as well as high latencies and high packet loss rate.
- Short battery life. Battery is a scarce resource for the current handhelds. Carrying additional sources of power can become too expensive. Saving the power is a must for any application designed to work in such environments.
- Small display size and low processing power.

In this work we address the following questions: How are the limited and highly dynamic resources of mobile clients, like network connection, energy supply or display size, influencing the design and deployment of groupware systems? and What type of policies the users need to specify in order to be able to collaborate in such environments?

In order to better understand the new challenges, let us consider an example application. Let us consider a civil disaster recovery scenario, after the collapsing of a big office building. A team of workers is deployed at the scene of the disaster. Each participant has a mobile device (a PocketPC or a laptop, for example) connected with a digital camera and a GPS receiver. The devices can communicate with each other using wireless connections. The workers will move around the disaster area and will need to collaborate by sharing information on various things of interest, such as the position of different objects, debris, hazardous materials or possible survivors.

We will first simplify the example. Let us assume that a worker will use a shared whiteboard type of application. He can place on his whiteboard three different types of objects: images, corresponding to the pictures taken by his digital camera, hazardous materials or conditions he might encounter and possible survivors. Ideally, any of the workers would be interested in receiving all of the information produced by any of the other participants. However, in mobile environments, the scarce and highly dynamic resources can negatively affect the quality of the collaboration based on the WYSIWIS paradigm. For example, a worker might want to receive only information about the nearby areas, or only about possible survivors and hazardous conditions when the quality of its connection goes

down or when he runs out of battery and synchronize later when the external conditions improve. In other words, he would like to deploy a **statefull** policy, in which the decisions on what type of notifications to receive to be done dynamically, based on the real-time conditions.

In this paper we propose a middleware, called SYNG (Statefull Synchronous Groupware), on top of which we can build synchronous groupware applications based on statefull policies. In our system, each user can define a state composed of a number of variables. Examples of such variables include battery usage, quality of network connection, display size, etc. Based on this state, the participant in a collaborative session specifies its own policy for receiving messages from the other participants.

The paper is organized as follows. We first present our main contributions and review related work in this area. Next, we describe our approach for supporting synchronous groupware in mobile environments. We then present a detailed example, elaborating on the process of creation and deployment of the user policies. System performance is evaluated using this example application. Finally, we discuss further work and conclude the paper.

II. OVERVIEW AND MAJOR CONTRIBUTIONS

We adopt a centralized architecture, where the participants in a collaborative session use wireless devices, such as PDAs, and communicate with each other via a centralized component, called server. The server maintains a state for each client. The state is a tuple-space and can be composed of any number of variables which describe the properties of that client. Examples of such variables are the quality of the connection, the battery utilization or the location. The policy of each user will be able to filter the messages that reach the user based on the message itself and the user state. A domain specific language is proposed, that will allow for supporting arbitrarily complex policies in a secure and efficient manner. We evaluate our approach based on a relative complex example application.

There are two major contributions of this paper. First, we introduce the concept of a statefull groupware system which is suitable for mobile environments, and present a new approach for specifying the collaboration policies. Second, we present an efficient middleware that implements this model, on top of which one can easily build different applications.

III. RELATED WORK

Synchronous groupware has been an active research area. Different systems have been proposed to deal with the centralized case, where a server is responsible of broadcasting the message to all of the participants. Examples of such systems are GroupKit [9], Rendezvous [8] or GroupDesign [5]. All of them offer important features to allow the user an easier development of the collaborative application (notifications, easy deployment of the communication bus, etc.). We believe that this work is complementary, in the sense that all of these features can be implemented on top of our statefull paradigm in order to support mobile clients.

DACIA [6] is a framework that facilitates the development of collaborative applications in mobile environments. A DACIA distributed application is viewed as a graph of connected components that typically implement data streaming, processing, and filtering functions. DACIA provides mechanisms for runtime reconfiguration of applications, to allow them to adapt to the changing operating environments. An application is reconfigured either as a result of applying adaptive algorithms embedded in the application as monitors, or through the manual intervention of a user or system administrator. We believe that our work is complementary, since DACIA only deals with the migration of the components in mobile environments, while in this paper we propose a novel approach to build statefull policies for the clients.

In [1], the authors propose YCab, a middleware that can be used to develop collaborative applications in mobile environments, assuming an ad-hoc network topology with no centralized component. There are two major differences with our work. First, in YCab, the authors do not take into consideration the difficulty of maintaining a consistent global state in such an environment, incorrectly assuming that the multicast capabilities of the devices will automatically solve this problem. Second, it is not possible to define statefull policies in YCab.

The closest in spirit with our approach are the concept of Notification Server (NSTP) [7] and the LambdaMOO [2] language. NSTP abstracts out the problem of application state consistency and provides a simple, general and open service to support the construction of synchronous groupware. LambdaMOO is a network-accessible, multi-user, programmable, interactive system well suited to the construction of text-based adventure games, conferencing systems, and other collaborative software. It provides a programming language with which users may create object behaviors. None of them have the concept of statefull policies, in which the decision of what to send to a particular client are made based on the current state of that client.

The need for a policy based groupware system was also recognized in the Intermezzo system [3]. Intermezzo can implement a policy like: “Do not let anyone bother me when I am working on my thesis, unless it is my advisor”, which is basically an access control based policy. We believe that it

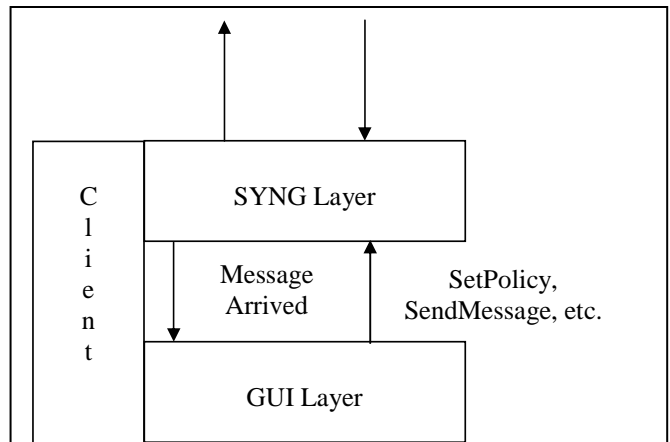


Figure 1: The Structure of a Client Application

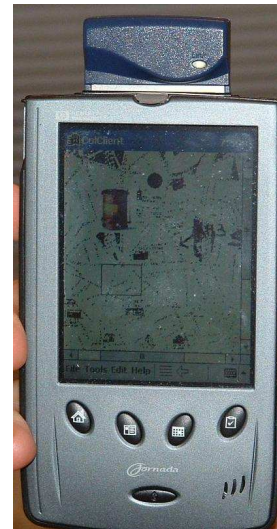
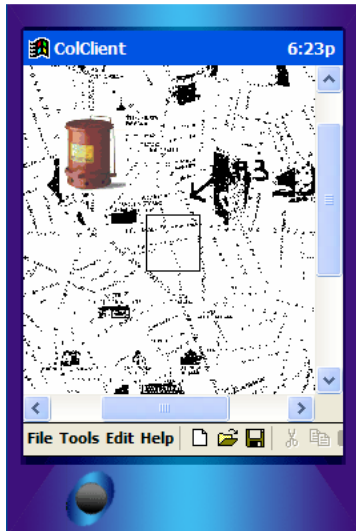


Figure 2: A SYNG application in session. Left: On the PocketPC Desktop Emulator. Right: On the HP Jornada 540

would be fairly easy to implement “access control” like policies using the techniques presented in this paper.

IV. A MIDDLEWARE FOR STATEFULL GROUPWARE IN MOBILE ENVIRONMENTS

A. Client-Server vs Peer-to-Peer

One of the first decisions a designer for a groupware system has to make is whether to use a client-server based approach or a peer-to-peer one. Many groupware systems were developed using a centralized component and many others were developed using a peer-to-peer architecture. The jury is still out there for the best approach in wired environments. The majority of the commercial applications are built on a centralized architecture (such as Microsoft NetMeeting [10]), mainly because the consistency of the shared state is much easier to be maintained in a centralized approach.

Starting with the seminal work from [4], many algorithms were proposed to allow for the maintenance of the consistency of the shared state in a peer-to-peer environment, without a centralized component. One of the main assumptions of all these algorithms is that the peers can directly and reliably communicate with each other at any given moment in time.

The situation is different in a mobile environment. In an ad-hoc network a peer can only communicate with its physical neighbors. While the ad-hoc networks are studied extensively for some time, the applications that were deployed on top of such networks are usually reduced to data collection (in sensor networks) or file sharing. We believe the properties of the current ad-hoc networks dramatically reduce the possibility of deploying complex groupware applications, because of the difficulty to maintain the consistency of the global state. On the contrary, the high availability of base-stations (802.11, GPRS) makes the use of a centralized system a much more appealing solution for complex groupware applications. Studying the development of such applications without a central component is an active area in our future research.

B. Client architecture

The middleware is based on a client-server architecture, in which the clients are the participants in a collaborative session.

The structure of a client SYNG application is shown in Figure 1. It is composed of two layers: the SYNG layer, a generic library provided by our middleware which is responsible with all communication with the SYNG server, and the GUI layer, which is application dependent. The GUI can call the SYNG layer using a well-defined API which provides methods for joining and leaving a collaborative session, sending the user policy, sending and receiving a message and resynchronization. The SYNG layer passes all received messages to the GUI layer, as shown in Figure 1.

A screenshot of a SYNG collaboration client is shown in Figure 2. The users are working together in a civil disaster scenario. Each of the users is equipped with a GPS receiver and a digital camera. The background is a digital map of the covered area and each user position is shown as a colored circle. A participant can insert different types of objects such as pictures taken with the digital camera (showing the location of hazardous materials for example), or possible locations for debris or survivors (modeled as different geometric objects, such as rectangles or circles).

Internally, the collaborative application state is maintained as a collection of such objects (rectangles, circles, images, etc.) each of them with a specific state (such as position) and a unique object ID. Each user has its own unique ID (generated randomly or based on the IP address of the device) and a local counter which is increased every time an object is created by the particular user. The object ID is the user ID together with the local counter.

C. Overall architecture

The overall architecture is presented in Figure 3. The server maintains a Global Application State, which can be used to accommodate latecomers or clients that worked offline for a period of time.

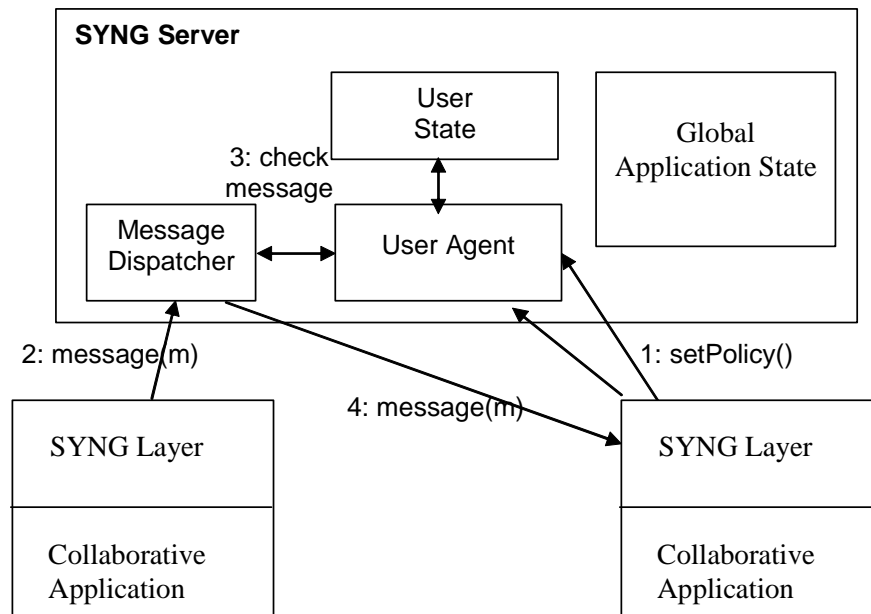


Figure 3: The SYNG middleware

A client will need to join the collaborative session by sending its own policy to the server, using the `setPolicy` command (Step 1 in Figure 3). We will detail on how clients can specify their policies in Section 5. Based on the policy, the server generates an agent, which will be associated with this client. In our current implementation, the agent is a dynamic link library (DLL), which is obtained from compiling the given policy. The main function implemented by the agent is the `arrivedMessage` function which specifies how each message needs to be processed. The `arrivedMessage` function will receive two parameters: a pointer to the message (`pMessage`) and a pointer to the current state (`pState`).

As we can see in Figure 3, the server maintains a state for each of the clients. The state is a tuple-space and can be composed of any number of variables that describe the properties of the user. We distinguish between two types of variables: automatic, which are maintained by the SYNG middleware and manual, maintained and specified by the user policy. Examples of automatic variables are the quality of the connection, the battery utilization or the device size. The values of these variables are updated by our system and they can be either computed in real-time by the server and the client (such as the quality of connection), or can be simply send by the client in a periodic heartbeat (such as batter utilization).

When a message arrives at the server (Step 2 in Figure 3), the `arrivedMessage()` for each agent will be invoked. In our current implementation the policies are specified in a C++ -like syntax. In addition to all the operations supported by C++, the policy can invoke a number of system operations that are implemented by the server. The currently supported grammar is presented in Figure 4. There are three operations associated with the handling of the properties maintained in the client state (`getProperty`, `setProperty` and `removeProperty`). The policy can also specify how to create a message, using the `createMessage`

operation, and how to manipulate the content of a message. We will give a detailed example of a policy in Section 5.

Since the policies are executed on the server site on behalf of the clients, the resources of the server (in terms of memory capacity for example) can be consumed in an unfair way by malicious or bogus policies. One thread (with a given maximum amount of memory) is allocated to each client. In order to prevent any unexpected crashes, all of the possible exceptions are caught when the policy is translated in C++.

Currently, the system supports three automatic variables: the location of an agent, the battery utilization and the network connection quality. The latter is an indicator of how many messages were lost in the wireless network. For example, if the network connection is 75%, it means that 25% of the messages sent to this user were lost in the network.

There are two main features that distinguish our solution than other collaborative middlewares. First, we propose a novel approach for a collaborative system, based on statefull policies. Based on the policy, the server will create at runtime a user agent which will be responsible with determining what messages the particular user is interested in. Second, we propose an automatic way to generate these policies, based on the user preferences. We will detail on this in the next section.

V. CASE STUDY: A DISASTER RECOVERY APPLICATION

In this section we present a reasonable complex application built on top of the SYNG middleware. Our main focus will be on the definition and usage of the statefull policy. We will detail the performance of our system in the next section.

Let us consider a civil disaster application in which a team of workers is investigating an area hit by a disaster (see the screenshot from Figure 2). Each worker has a wireless device (a PocketPC, for example), equipped with a GPS receiver and a

digital camera. Each user will be able to see any other user location. We can consider that the devices can communicate with the server using a wireless LAN, or a cellular connection if the wireless LAN is not accessible. It is also possible for a user to work offline should the network connectivity fails for a period of time.

Any user can generate three types of objects: images representing pictures taken in different locations, information regarding possible survivors and information about hazardous materials.

Suppose that a user wants to have the following policy, called P_D , composed of the following rules:

- **RQ1:** When the battery is charged and the connection is good, the user is interested in receiving all messages and would like to resynchronize with the server to obtain the latest global application state every five minutes.
- **RQ2:** If the battery usage is below 50% of the usual value and the network connection is lower than 50%, the user is interested in messages regarding the possible survivors and compressed versions of the images.
- **RQ3:** If the network connection is below 75%, the user is interested in messages regarding the possible survivors and hazardous materials that happen within a given radius of his current location.

A. Specifying the policy using the Policy Wizard

Specifying the policy in a direct manner is not an easy task for an ordinary user. Our main focus was to design an easy and intuitive graphical user interface, called Policy Wizard, which will allow the user to generate the policy automatically. An advanced user will be able to manually change and tune the policy, but for the big majority of users the generated policy will be enough.

A policy is logically composed of a number of rules. In

Figure 5 we present a screenshot of the Policy Wizard which is used to implement the P_D policy. A user can add conditions or changes the existing ones (left image on Figure 5). Each condition will have a unique name, assigned by the user. In the second snapshot, we show the user interface used to define conditions. It is the implementation of the **RQ2** rule from the P_D policy. The user can select the variables he wants to use for creating the condition (in our case the battery usage and network connection), create the conditions (in our case Battery < 50% AND Network < 50) and then select what objects he will be interested in obtaining images (in a compressed manner) and information about survivors. The other rules are generated in a similar manner.

We show in Figure 6 the automatically generated code for the RQ2 rule. As mentioned in Section 4.2, pMessage is a pointer to the initial message and pState is a pointer to the current state. The parameters specified by the user in the PolicyWizard for rule RQ2 are retrieved from the state (lines 2 and 3). If the condition specified by the user is fulfilled (line 4), the message type will be tested. If the message type is TYPE_SURVIVORS, the message will be sent to the user (line 15). If the type is TYPE_IMAGES, a new message is created (with a compressed version of the image) and is sent to the user (line 13). Otherwise, the message is dropped (line 17).

B. Adding a manual variable

As mentioned in Section 4.2, our system supports manual variable, which are policy dependent variables and can be used as the user sees fit. For example, in order to implement the rule **RQ1**, we will need a manual variable to keep the last time when a resynchronization was done. Because of space restrictions, we will not insert here the whole implementation of the **RQ1**, as generated by the ResyncCondition wizard. Intuitively, a manual variable will be created in the user's state, with a given name ("Timestamp" for example). The value will be checked every time a message is evaluated and saved afterwards, using the getProperty and setProperty functions,

Operations on the associated state	
getPos()	Returns the current position of the user.
getProperty(name)	Returns the value of the specified property, or null if the property is not currently defined in the state
setProperty(name, value)	Adds a property with the specified name and value
removeProperty(name)	Remove the property with the specified name from the state
Operations on messages	
synchronize()	Send the global application state to that client.
getData(m)	Get the data associated with a message
setData(m,p)	Set the data for this message
createMessage()	Create an empty message
compressImage()	Compress an jpeg image with a specified compression ratio

Figure 4: The system operations

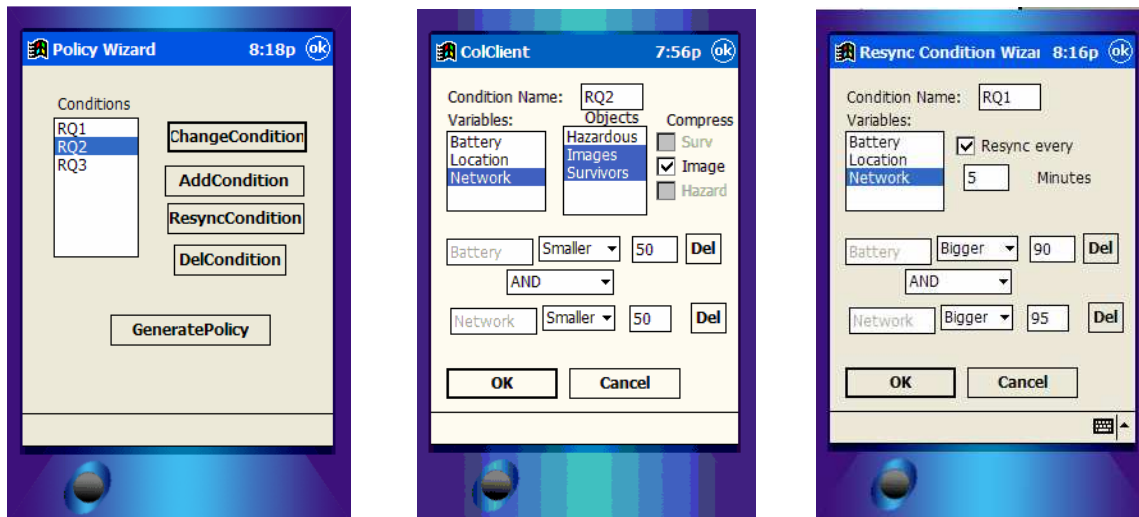


Figure 5: Left: Policy Wizard for the P_D policy. Middle: The AddCondition Wizard for RQ2. Right: The ResyncCondition Wizard.

presented in Section 4.2.

Using manual variables, an advanced user will be able to extend the functionality of the policy, by adding more complex logic. For example, the user can specify that he wants to aggregate multiple messages, or modify the content of messages, etc.

VI. IMPLEMENTATION AND PERFORMANCE

The statefull model for synchronous groupware systems is generic and flexible, allowing for a large variety of applications and policies. The performance of the system cannot be evaluated for a generic case, since it is affected by the specific application, the format of the data that is transferred, as well as by the policy of a participant in the collaborative session.

For the performance evaluations, we will use the Disaster Recovery application, with the policy specified in Section 5. The SYNG server is running on a dual Pentium Xeon at 2.8GHz with Linux 2.4.0 kernel. Ideally, we would want to use a PDA on which we can attach both a digital camera and a GPS receiver. Such devices already exist; an example is the HP iPAQ h6315. However, due to lack of funding, we could not buy this device to do the experiments. For our tests, we used an HP Jornada 540, with 32 MB of RAM and a SH3 processor at 133 MHz equipped with a NETGEAR ma701 wireless card. We emulated the GPS by allowing the user to manually specify its location on a given map. Some predefined pictures are available on each device, which can be selected by each user and send to the other participants in lieu of the dynamic pictures that can be obtained using a digital camera. While these limitations would impact the usage of such a device in a real-life situation, they will not affect the validity of the performance evaluation.

As a general remark for our experiments, we will drive the load into the system using one active client that generates messages and a variable number of listening clients. The active client can generate messages at a variable rate, ranging from 20 to 120 mpm (messages per minute). The traffic introduced by

the active client is the equivalent of the overall traffic in a real-life scenario. For example, if we assume that there are 10 participants in the collaborative session, 60 mpm will be equivalent with every participant sending a message every 10 seconds.

We will first study what is the overhead introduced by maintaining the state for each user. When the server receives the first message, it records this time as start time. The server records the ending time when it finishes processing all the messages it has received from any user. The duration of the experiment is determined by the difference between the end time and the start time.

We compare against a “blind” groupware system, in which there are no agents associated with the users and no states maintained. Each user receives all the messages sent by all of the other participants; the server acts just like a router in this case. As the metric we consider the duration of the experiment, as defined above and we define the overhead as: $O = (D_{Full} - D_{Blind}) / D_{Blind} * 100$, where D_{Full} is the duration of the experiment for the statefull system and D_{Blind} is the duration of the experiment for the “blind” groupware system.

The results are shown in Figure 7 averaged over 20 executions for each case. As we can notice, the overhead is 0 even for a large number of users (50), when the sending rate is not very high. Even for a high rate like 60 mpm, the overhead is only 4.5% when the number of users is high (100). This is a very promising result showing that the overhead is almost negligible, even for relatively complex statefull policies like P_D , for all practical scenarios.

We will now present how our architecture can improve the collaboration in wireless environments. First, we define the quality of collaboration as: $Q = N_i / N_T * 100$, where N_i is the number of messages the user is interested in receiving, according to his policy and N_T is the total number of messages received by the particular user. This metric basically quantifies how many “important” (according to each user definition)

/ Policy Wizard 1.0. Automatically generated code for condition RQ2 */*

```

1.  CDataMessage *pNewMessage;
2.  float syngBattery = pState->getBattery();
3.  float syngNetwork = pState->getNetwork();
4.  if ((syngBattery < 50) && (syngNetwork < 50) )
5.  {
6.      int messageType=pState->getMessageType();
7.      int messageType=pState->getMessageType();
8.      switch(messageType)
9.      {
10.     case TYPE_IMAGES:
11.         pNewMessage = createMessage();
12.         pNewMessage->setData(compressImage(pMessage->getData(),pState->getQuality()));
13.         return pNewMessage;
14.     case TYPE_SURVIVORS:
15.         return pMessage;
16.     default:
17.         return 0;
18.     }
19. }
20. return pMessage;

```

Figure 6: Implementation of Rule RQ2

messages are lost by the user. In this experiment we have one listening client, who is moving around.

The definition of “important” messages is user-dependent of course and therefore difficult to measure or quantify in a generic groupware system. Moreover, the importance can be dynamic throughout the collaborative session, depending on various factors. The existence of the user’s policy proves to make the definition of an “important” message easier by just using the ranking provided by the user in his policy. Looking at the P_D policy presented in Section 5, we can conclude that a message that creates or modifies an image is an important message, for example (because of Rule RQ2). More specifically, we will consider that a message is “important” for a user, if, according to its policy, the user is interested, in the given conditions, in receiving that particular message.

We are measuring the quality of collaboration with the

statefull P_D policy compared with a stateless policy, as in the previous experiment. The average size of a message is 85 KB and the duration is 2 hours for each experiment. The results are presented in Figure 8.

As we can notice, when the rate of the generating messages increases, the quality of collaboration decreases for a classic groupware system with almost 45%. This is because the system has no way to adapt itself to the dynamic changes in the bandwidth, a well-known attribute for wireless environments. For the statefull case, the quality remains basically flat, even under heavy load.

Our last measurements are related to the effect of the statefull policy on battery duration for a device. We present in Figure 9 the time needed for the battery to reach the 0 power level (the battery life), under statefull and stateless conditions. The first line in Figure 9 is the battery life, when no messages

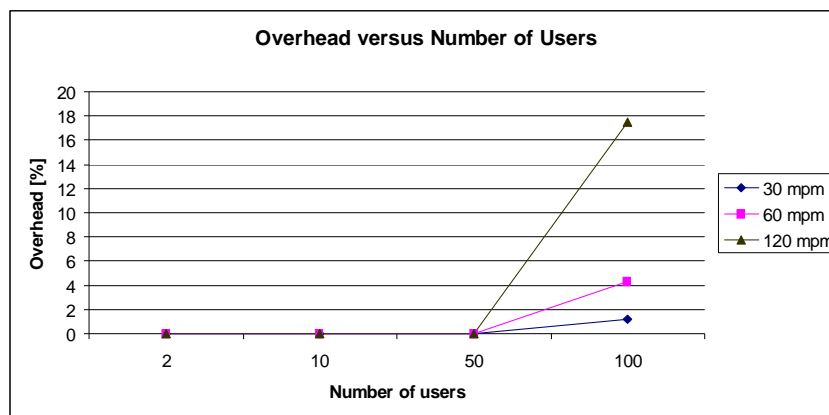


Figure 7: Overhead versus number of users

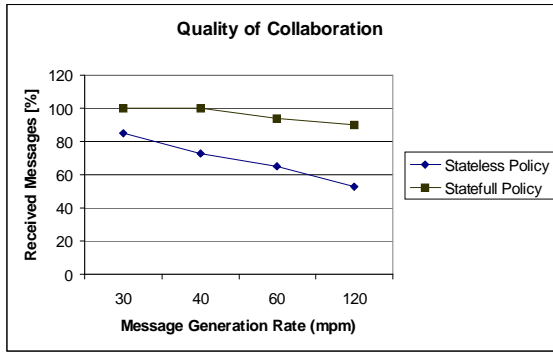


Figure 8: Quality of Collaboration

are sent in the system. We always start the experiment with the battery of the device fully loaded.

These results show we can extend the battery life by as much as 30%, dramatically increasing the usability of groupware systems in wireless environments.

We can conclude that the SYNG middleware offers very good performance for all practical scenarios, scaling very well up to 50 users. In addition, using this approach, one can dramatically improve the quality of collaboration, by reducing the loss of important messages and can extend the battery life up to 30%.

VII. CONCLUSIONS

There are two major contributions of this paper. We first introduce the notion of statefull groupware systems and argue that the current groupware systems are not suitable for mobile environments. A novel approach for expressing and enforcing the user policies is also presented. This technique allows the policies to interact with the current state of each user and decide whether or not to send it to the user and how to modify the message if necessary. Our approach can be incorporated in the current groupware systems in an incremental manner, by allowing for some of the clients to maintain a state at the server.

Second, we presented a motivational example where our techniques can be used. Experimental results demonstrate good performance and scalability of our approach. The quality of

collaboration, as defined in Section 6, improved with more than 45%, while the battery life can be extended with 30%.

REFERENCES

- [1] Buszko, D., Lee, W.H. and Helal, A. Decentralized Ad-Hoc Groupware API and Framework for Mobile Collaboration. In Proceedings of International Conference on Supporting Group Work (GROUP'01), Boulder, CO, Oct. 2001.
- [2] Curtis, P. LambdaMOO Programmer's Manual. Xerox, August 1993. ftp://ftp.research.att.com/dist/eostrom/MOO/html/ProgrammersManual_toc.html.
- [3] Edwards, W.K. Policies and Roles in Collaborative Applications. In ACM Conference on Computer-Supported Cooperative Work (CSCW'96), Boston, MA, pp.11-20, November 1996.
- [4] Ellis, C.A. and Gibbs, S.J.: Concurrency Control in Groupware Systems. Proceedings of the 19th ACM SIGMOD Conf. on the Management of Data, pp. 399-407, 1989.
- [5] Karsenty, A., Tronche, C., and Beaudouin-Lafon, M. GroupDesign: Shared editing in a heterogeneous environment. USENIX Computing Systems, 6(2),pp 167-195, Spring 1993.
- [6] Litiu, R. and Prakash, A. Developing Adaptive Groupware Applications Using a Mobile Component Framework. In Proceedings of the 2000 ACM Conference on Computer-Supported Cooperative Work, (CSCW 2000), Philadelphia, PA, Dec. 2000.
- [7] Patterson, J. F., Day, M. and Kucan, J. Notification servers for synchronous groupware. In ACM Conference on Computer-Supported Cooperative Work (CSCW'96), Boston, MA, pp.122-129, November 1996.
- [8] Patterson, J. F., Hill, R. D., Rohall, S. L., and Meeks, W. S. Rendezvous: An architecture for synchronous multi-user applications. Proc. ACM Conference on Computer-Supported Cooperative Work (CSCW'90), Los Angeles, CA, pp.317-328, October 1990.
- [9] Roseman, M., and Greenberg, S. Building real-time groupware with GroupKit, a groupware toolkit. ACM Transactions on Computer-Human Interaction, 3(1): 66-106, March 1996
- [10] Microsoft NetMeeting. <http://www.microsoft.com/windows/netmeeting/>

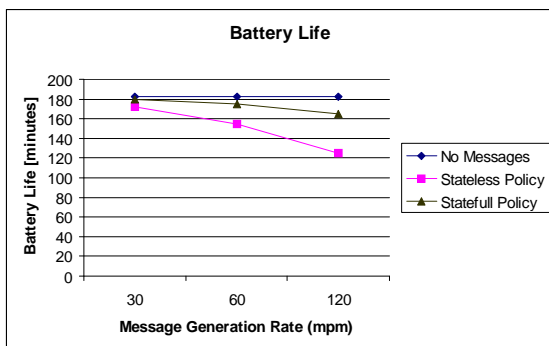


Figure 9: Battery life