SMS based Group Communication System for Mobile Devices

Christian Seeger
TU Darmstadt
Darmstadt, Hessen, Germany
cseeger@dvs.tu-darmstadt.de

Bettina Kemme
McGill University
Montreal, Quebec, Canada
kemme@cs.mcgill.ca

Huaigu Wu
SAP Research
Montreal, Quebec, Canada
huaigu.wu@sap.com

ABSTRACT

This paper presents a group communication system for mobile devices, called MobileGCS. Mobile communication is slow, expensive and suffers from occasionally disconnections, especially when users are in movement. MobileGCS is based on SMS and enables group communication despite these restrictions. It provides all primitives needed for a chat application and handles process failures. As mobile communication is expensive, MobileGCS is designed for small message overhead and, additionally, exploits SMS based message relaying to handle short-term disconnections. In this work, we present the group maintenance service and the multicast service of MobileGCS. In order to distribute the overhead of failure discovery over all processes we introduce the concept of a circle of responsibility for failure detection. We discuss informally that MobileGCS can handle the most common failures while keeping the message overhead very low.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*network communications, wireless communication*

General Terms

Design, Reliability

1. INTRODUCTION

Mobile phones have not only become a standard commodity for telephony but we also use them for online shopping, to find the nearest restaurants, and to chat with our friends. Text-messaging has become particularly popular, especially in Europe. Nevertheless, basically all interaction we currently do is between two mobile phones or between the mobile phone and a central service. While a central service might disseminate information (e.g., flight information) to many interested phones, a phone usually does not send messages to many recipients. Nevertheless, there are plenty of applications that would benefit from a communication middleware that allows mobile phones to participate in group communication. Two applications are chat among a group of friends or business partners, or information dissemination among a group of people with similar interest.

In this paper, we propose such a group communication system (GCS) providing both the primitives to manage a group of mobile phones as well as offering multicast to group members. A very special feature of our system is that it completely relies on SMS (the GSM Short Message Service) as underlying communication medium. SMS allows short messages to be sent from one mobile device to another without the need of a centrally maintained service that would charge extra service fees. Routing is done through the network carrier. Our decision on this communication medium has two main reasons. Firstly, not all mobile users subscribe to a data plan that would allow Internet connectivity, and access to the Internet through wireless access points is usually very sporadic. In contrast, SMS is basically always provided, continuously available, and many plans already include a high amount of free SMS messages. Secondly, even if a data plan or other wireless access exists, phones cannot be directly accessed by other phones through TCP or UDP as they do not own a permanent IP address. And even if they have for intermittent time, it is usually not possible to connect to them. Thus, any solution based on Internet communication would likely need to rely on a server on the Internet to which the phones connect. The server would be responsible of relaying messages to all phones. However, our goal was to design a truly distributed, server-less solution that is easier to deploy and run. Our GCS solution only relies on a network carrier that supports SMS and a Java-enabled phone. Compared to an ad hoc network solution, users do not need to be in the same communication area.

The solution that we present is a pragmatic one. Mobile communication is expensive and slow. Every message counts. Furthermore, mobile devices have low computing power and restricted memory. Thus, our solution provides much weaker properties than traditional group communication systems. For instance, we consider the communication overhead to maintain virtual synchrony [4] too high. Similarly, providing reliable message delivery [4] requires considerable communication and storage overhead that we are not willing to pay. Nevertheless, our system needs to be able to handle the fragile connectivity of mobile phones as phones can quickly disconnect for short, medium and long time periods. Thus, our approach includes extensive failure handling. However, it attempts to keep the overhead as

MobiDE10, June 6, 2010, Indianapolis, Indiana, USA.

small as possible. As a trade-off, it does not handle all failure combinations correctly. We believe this to be a compromise that users are readily going to accept.

Our solution was influenced by the requirements of the application that we believe will be the first one to adopt group communication technology, and that is chatting. Nevertheless, we believe that other applications can also benefit from our tool. Our GCS offers the chat application to create, join, leave and destroy a chat room and to send FIFO multicast messages. All message exchange is done via SMS and only among the phones. Failed phones are detected and removed from the group. The system handles short disconnections gracefully. In order to keep the message overhead for group maintenance small, we introduce the concept of *circle of responsibility* as our failure detection system. Our GCS design, named *MobileGCS* relies on one of the phones to be the master phone to manage group management. This expensive task can easily rotate among the members.

2. BACKGROUND

Group Communication Systems is a middleware that provides two types of services [4]: *group maintenance service* and *multicast service*. Group maintenance manages a list of all active members, called *view V*. At any given point of time a view describes the current set of members of a group. Processes can join or leave, and failed processes will be excluded. Members are informed about a view change through the delivery of a *view change* message containing the members of the new view. The big challenge is to find a consensus between member processes about the current view. View proposal algorithms usually involve complex coordination protocols in order to guarantee that all members agree on the same view. Advanced properties such as *virtual synchrony* [3] are even more costly as they provide a logical order between view change messages and application messages delivered in each view.

The *multicast service* propagates application messages submitted by the application layer to all group members. In our notation, we say that the application layer of a member receives a message that the GCS layer delivers to it. There are two main demands on a multicast service: *ordering* and *reliability*. *FIFO ordering* requires that two messages sent by a particular node are delivered in sending order. *Causal ordering* requires that if an application first receives a message m and then sends a message m', then all members should deliver m before m'. And *total ordering* requires for every two messages m and m' and two processes, if both deliver m and m' they deliver them in the same order. Message delivery can be *unreliable*, *reliable* or *uniform reliable*. Reliable delivery (uniform reliable delivery) guarantees that if a message is delivered to an available member (to any member – available or one that crashes shortly after) then it will be delivered to all available members. The higher the degree of ordering and/or reliability, the more expensive and complex is the message exchange between the members in term of additional messages and message delay.

Network Environment of Mobile Devices. Mobile devices, especially mobile phones, usually connect to stationary *base station*s provided by network carriers which provide mobile devices with different speech and data services. The most common data services are *SMS, MMS, GPRS* and *UMTS*. SMS and MMS are designed for direct data communication among mobile phones. Messages are addressed by the receiver's phone number and can be sent even if the receiver is disconnected from the network. The network carriers store the messages and relay them when the receiver is connected again although the number of messages and the time messages are stored are limited. GPRS and UTMS enable mobile phones to establish an Internet connection. The base station allocates an IP address to the device and acts as a router enabling message delivery but only as long as the phone is connected to the Internet. Furthermore, IP addresses can change quickly due to two reasons. Phones automatically disconnect after a certain idle time. When the phone reconnects, the phone's base station might allocate another IP address. Furthermore, if a mobile phone moves from one cell to another, the base stations change and, hence, the allocated IP address changes, too. In addition to this, for propagating a phone's current IP address an additional server is needed.Phones could also connect to the Internet through wireless access points. However, such connectivity is very sporadic and not available everywhere. Therefore, we decided to use SMS as underlying communication layer due to its universal, bidirectional and fairly reliable services. MMS would be equally possible and we will look into this in future work. Disadvantages of SMS are an often higher message delay than for IP packets and a payment per message independently of the message size.

Application. We decided for a chat application as our example application and developed our GCS with regard to the primitives a chat application requires. In our opinion, chatting is a feasible scenario for a mobile application, because almost every mobile device fulfillls the hardware requirements for a chat application. Additionally, we assume that friends or colleagues have their phone numbers already stored in their mobile phones. Hence, the users do not need additional information from a server as long as the membership consists of known people. Since there is no need for a name server in a chat application with known members, we decided to design a completely decentralized group communication system without an expensive server. However, a server-based naming service could be easiy integrated into our GCS architecture. In a chat application typically all members multicast relatively short messages. While causal order would be desirable, FIFO order should be acceptable for most situations. While reliability is important, the emphasis is probably more on fast message delivery. We assume that a chat application on a mobile phone is not feasible with more than 20 users, as the message delay would be too high. For applications beyond 20 users, SMS and server-less communication will likely be problematic due to the high message costs and delay. With twenty users, view change messages can be easily propagated within one message assuming phone numbers are process identifiers.

3. RELATED WORK

Group communication systems are available for many different network types. The first generation of GCS has been mainly developed for local area networks (LANs) such as Totem [11], Isis [3], Horus [16] and Spread [1]. They provide basically all virtual synchrony and strong ordering and reliability guarantees.

There are also approaches for mobile networks. The authors of [13] propose an algorithm for consistent group membership in ad hoc networks. This algorithm allows hosts within communication range to maintain a consistent view

of the group membership despite movement and frequent disconnections. Processes can be included or excluded with regard to their distance from the group. Different groups can be merged when they move into a common geographical area and the partition of one group can be handled as multiple disjoint groups. Another further approach [12] uses not only the ad-hoc network, but also the cellular network and a *Virtual Cellular Network* (VCN). A *Proximity Layer* protocol monitors all network nodes within a certain area.Based on this information a three-round group membership protocol builds a group of mobile nodes.

Closest to our approach is SMS GupShup Chat [15] which is a commercial group chat application based on SMS and managed by a central server. Users are able to create a group by sending a SMS message to the special phone number of the server. Also invitation messages containing up to four phone numbers are possible. Once a group is created, users can join or leave the group. Users can post a message to the group by sending a simple SMS message to the special phone number. The message forwarding to all group members is done by the server. In contrast to SMS GubShup Chat, we build a GCS for chatting which is completely decentralized and does not rely on a central server.

Not all existing systems provide strong guarantees. Epidemic approaches only provide guarantees with a certain probability and will only achieve that messages are "eventually" delivered (such [2, 6]) or views "eventually" converge (e.g, [7]). The idea is to let nodes regularly exchange their past history of received messages. Given the low memory capacity and the high costs of communication, we do not consider epidemic protocols applicable for mobile phones. Also, in our application context of chatting we require much lower delivery delays as provided by epidemic protocols.

4. SYSTEM OVERVIEW

Our GCS layer provides the typical primitives to the application: create, join, leave and destroy a group. The application receives a view change in form of an SMS message every time the group configuration changes. The application can write an SMS and submit it to the GCS layer. The GCS layer will deliver this messages to all group members.

Multicast. We do not provide reliable message delivery to all available nodes. This would require nodes to store messages it receives from other nodes in order to be able to relay them in case of the failure of the sender. We consider this unfeasible for mobile environments. However, as mentioned above, we can assume each individual SMS message to be delivered reliably, even when short periods of disconnection occur. Therefore, we implement multicast by simply sending the message via SMS to each phone that is currently in the view of the sending phone. This achieves what we call *sender reliability*. A message sent by a node that does not fail during the sending process is delivered to all available members that are in the view of the sending process. If the sender fails during the sending process, some members might not receive the message. If a phone disconnects before the message is received, it will very likely receive it upon reconnection. Furthermore, as SMS offers FIFO delivery, we automatically also provide FIFO delivery.

Group membership guarantees. Considering a chat application, we think that virtual synchrony, although desirable, is not absolutely needed. Thus, view membership is decoupled from the delivery of application messages.

Ideally, we would like to have an eventual agreement, that is, all available members of a group will have eventually the same view of the group if there is a sufficiently long time without membership changes. We achieve this if we assume a strong failure detector that allows for the correct detection of a failure by choosing a sufficiently large timeout interval. In most cases, wrongly suspecting a non-failed node will simply lead to the exclusion of an available node from the group, something that we consider acceptable. However, in some rare cases, a wrong suspicion or short-term disconnections might lead to partitioned, and thus, incorrect views. Nevertheless, we tolerate many forms of concurrent failures, and we believe that our properties are acceptable for chat applications. As a result, we do not offer more than best-effort membership that will handle the most common errors but might not converge in some cases.

The remainder of this paper is dedicated to the discussion of the membership protocols.

5. MOBILE GCS WITHOUT FAILURES

Groups in Mobile GCS have one process that is responsible for the group membership service, called group master. The group master is responsible for creating and destroying the group, for processing join, leave and failure requests, and for sending the corresponding view change messages to all members. When the application calls the create primitive of the GCS layer, the corresponding phone becomes the master of group. As the group master has a higher load than other phones due to this coordination overhead, we provide a mechanism to move the master responsibility to another node. All mechanisms are designed to use very few messages as this will be sufficient if there are no failures in the system. In this section we assume there are no failures. Failure handling is described in Section 6.

In the following descriptions we use letters instead of phone numbers. Letters in braces under a phone represent the current view of this phone. The group master is indicated by an asterisk behind its identification letter and by a black phone in the figures. Phones of group members are gray shaded and phones of non-members are white. Time steps represent the time a phone has to react to a previous stimulus (e.g. incoming or missing SMS message).

5.1 Create/Destroy

Since we avoid the usage of a central server the existence of a new group has to be propagated. The idea is to invite other processes to a group and combine this with the creation procedure. This is useful for chatting as it allows the creation of a new chat room and to invite other people to join it. Figure 1 shows how the creation and invitation is done. In time step T0, the user of the upper phone creates a new group. The create method requires a group name and a list of other phones that are invited to become group members. For a chat application, this will open a chat room and invite others to join the group. The group name only needs to be unique over its lifetime across the phones that might want to participate. Given that it is unlikely that a given user will create many chat rooms concurrently, a group name containing the creator's identifier and a sequence number suffice. A phone that calls the create method automatically becomes the group master (black color) and the group cre-

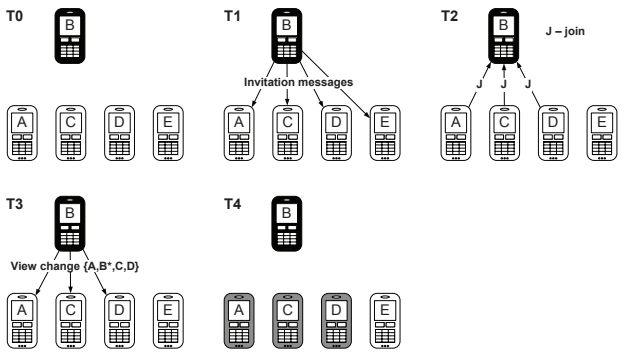

Figure 1: Create

ation is completed only including the calling phone as group member. The next step is the invitation of contacts chosen by the user. The chosen phones receive invitation messages including the group name from the group master in T1. The GCS layer of these phones relay the message to the application which can now indicate whether it wants to accept the invitation. If this is the case, the GCS sends a join request to the master. In the example, each phone except of phone E sends a join request in step T2. In T3, the group master adds all joining processes to the view and sends a view change message to all members of the new group. The master only waits a limited time to send the view change. If a further join request is received later, it simply sends a further view change message. Each view gets an increasing sequence numbers as identifier. At T4 phones A-D are all members of the group and have the same view.

For a chat application we think it makes sense that a group can only be destroyed when there is only one process left. Since every group must have a group master and there remains only one process in the group, this one is the group master. It can submit a destroy request which removes the last member and removes the group. If a group master likes to leave the group without destroying it, a new group master has to be elected. Section 5.4 describes how this is done.

5.2 Join

A joining phone needs to know the group name and can send a join request to any group member. This group member forwards the request to the master which processes it. It includes the new phone into the group and sends a group join message containing the new view to all members of the new view. It sends this message to the joining node last. This is important for failure handling as discussed in Section 6. Each node, upon receiving the message, updates its view and informs the application.

Figure 2 depicts a group that first consists of the processes A, B^*, C and all processes have the same view installed. In time step T1, D sends a join request to the group member C. C is not the group master and forwards the join request to B^*. At T2, the group master adds D to the view and sends a view change message first to the old view members A and C, and then to joining process D.

5.3 Leave

When a phone other than the master wants to leave the group, the GCS layer sends a leave request message to the master process and deletes its local information regarding

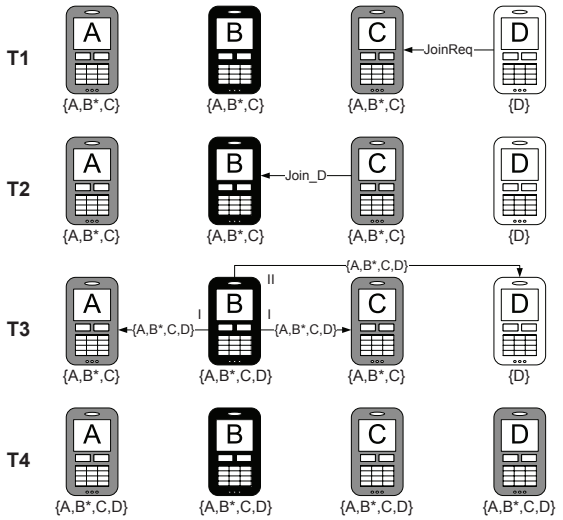

Figure 2: Join

the view. The master deletes the phone from its group and sends a view change message to all members in the new view which adjust their local views and deliver the new view to the application. The master also sends the leave message to the leaving node so it knows that the leave was successful. All messages the leaving node might receive after having sent the leave request, are suppressed by the GCS layer.

Before the mater process can leave the group, it has to evoke the master move operation to determine a new master. Once the new master is established, the old master sends a leave request to the new master.

5.4 Master Move

The *master move* operation can be called by the group master to elect a new group master. The master process has additional message overhead when membership changes occur, leading to higher costs. Frequently alternating group masters distribute the additional master costs among all phones. The operation could be exposed to the application or be executed internally in the GCS layer, e.g., after a given time period or after a certain number of processed view changes. It will also be executed when the current master wants to leave the group (see Section 5.3). To initiate the master move, the current (old) master sends a *master move* message to the new master. Upon receiving this message, the new master changes the master flag in its view to point to itself and sends a view change message first to all other members before it sends it to the old master. Sending the view change massage to the old master at the end is needed in case of failures. It allows the old master to check whether the master move succeeded or not. Section 6 discusses the cases where the move fails. Whenever a process receives the view change message it adjusts the master flag accordingly. While the master move protocol is in process there exist two group masters. Nodes that have received the view change message know that there is a new master, while those who have not yet received it don't. In order to handle this, the old master stops acting like a master process once it has sent the master move message and only stores incoming view change requests in this period of time. Upon receiving the view change message from the new master, the old mas-

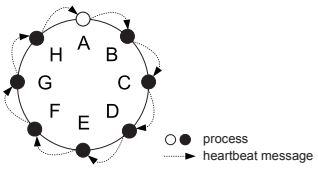

Figure 3: Circle of Responsibility

ter knows that the master move succeeded and forwards all view change requests to the new master.

6. FAILURE DETECTION

SMS does not establish a connection to other phones nor does it provide a method to check whether a phone is available or not. Hence, the GCS has to detect failures by its own. Failure detectors are a standard component of GCS. They typically require members to send heartbeat messages to each other. Once heartbeat messages are not received for a certain period of time, the member is suspected to have failed. Then, an agreement protocol is run to remove the suspected node. As we mentioned before, we do not want to have a complex protocol requiring many messages, neither heartbeat nor agreement messages. Thus, we use a pragmatic approach where each member only sends heartbeat messages to one other node, and this node makes a solitary decision to remove the node if it does not receive the heartbeat messages anymore. A simple approach is to put the failure detector on the master and let all phones send the heartbeat messages to the master. But this overloads the master and needs enhancements to handle master failure.

The authors in [8] and [14] introduce distributed failure detectors that distribute the workload for failure detection to more than one failure detection modules. Each module monitors a subset of nodes and, thus, has a reduced workload compared to a central approach. We use the same idea by introducing a circle of responsibility among all processes. The GCS runs on mobile phones and every phone has a unique phone number. Since we use phone numbers as process identifiers, every process knows all phone numbers in the current view. By sorting the phone numbers and connecting the first number with the last number, we get a unique circle of phone numbers which is known by every process. As a result, every process knows its successors and predecessors. Figure 3 illustrates such a circle of responsibility. For simplification, we use again letters instead of phone numbers. The white process A is monitored by the successor process on its right side and, therefore, it sends heartbeat messages to B every time period t. Every successor process also knows its predecessor process and expects heartbeat messages from it.

6.1 Failure of a process (not Master)

If an expected heartbeat message is missing for a period T (T is significant larger than t in order to handle message delay variations), the failure procedure is started. The monitoring process performs a self test, and if it succeeds it sends a *process down* message to the group master. The group master, upon receiving this request, removes the failed process from the view and sends a view change message to all members of the new view and the failed node. In principle, when node B does not receive the heartbeat from A, A could have failed or be disconnected, in which case it should be excluded from the group. Alternatively, B itself could be temporarily disconnected from the network. If the latter is the case, B should not send the process down message to the master. The self-test allows B to detect whether it is currently connected and is described in Section 6.5.

6.2 Adapting to Process Leaves/Failures

For the circle of responsibility, it makes no difference whether a process leaves the group or has failed. In both cases, the process will be excluded from the failure detection and the circle of responsibility has to be adapted. The adaption is done as follows: the successor process of a leaving process has to change the process it monitors and the predecessor process has to change its heartbeat receiver. Assume process p_i leaves or fails. Then the successor of p_i, i.e., p_{i+1} must now monitor the predecessor of p_i, i.e., p_{i-1}. That is, p_{i-1} has now to send its heartbeat messages to p_{i+1} instead of p_i. If the leaving process p_i has a temporary status (temporary processes are described in next section), p_{i+1} only deletes p_i as a heartbeat receiver and p_{i+1} stops monitoring it. No other process needs to adjust its monitoring activity.

6.3 Adapting to Process Joins

If a process joins the group, the responsibilities change and the circle of responsibility has to adapt to it. A joining process p_i is only then completely included into to the circle when p_i actually knows that the join was successful. This is necessary because it might be that the group master failed in the middle of sending the corresponding view change message. As the group master sends the change to p_i last, it can be assured that when p_i receives the view change message, all others will receive it, too. But if p_i does not receive it, it will not consider itself in the group, and thus, will not initiate the circle of responsibility. In order to avoid a gap in the circle of responsibility, a joining process gets first a temporary status. Upon receiving the first heartbeat message from a joining process, it is assured that all available processes will receive the new join. Only the processes p_{i-1}, p_i and p_{i+1} have to adjust their monitoring activity upon receiving the view change message including p_i: (i) p_{i-1} marks p_i as temporary and starts sending heartbeat messages to both p_i and p_{i+1}, (ii) p_i starts sending heartbeat messages to p_{i+1} and monitoring p_{i-1} and (iii) p_{i+1} marks p_i as temporary and starts monitoring p_i (it still monitors also p_{i-1}). Upon receiving p_i's first heartbeat message, p_{i+1} stops monitoring its former predecessor p_{i-1} and deletes p_i's temporary status. In addition to this, p_i sends a *stop heartbeats* message to p_{i-1}. Process p_{i-1}, upon receiving p_{i+1}'s stop heartbeats message, deletes p_i's temporary status and stops sending heartbeat messages to p_{i+1}.

If there are two or more joining processes in a row, they are all first monitored as temporary processes.

6.4 Master Failure

If the master fails, a new master process has to be found. In order to avoid a costly agreement protocol, our failure detection system elects the master's first available successor process. This is done as follows: If the master p_i fails and its successor p_{i+1} is still alive, this process detects the failure and elects itself as the new master. Accordingly, it removes p_i from its view, sets the master flag to itself and sends a view change message containing a master move. In the case that the master and one or more of its successors fail, the

first available successor p_j detects the failure of its predecessor p_{j-1}. As a consequence, it sends a down message to the (already failed) group master p_i. As p_j does not receive a view change from p_i after a certain timeout, it suspects p_i to have failed and forwards its failure assumption about p_i and p_{j-1} to the master's successor p_{i+1}. If still no response, it continues until it receives a response from one of the processes. If not, it has unsuccessfully probed all successors of the old master and, hence, p_j itself is the first available successor. Therefore, p_j elects itself as the new master and sends a view change message excluding the old master and all unsuccessfully probed processes.
It might be that a further failure somewhere else in the circle occurs. The successor p_u of this failed process will not receive a view change message either. Hence, it also starts probing all old master's successors. If it sends to an available successor p_j which has not detected and elected itself as the new master so far, p_j stores the down message from p_u and sends an acknowledge message to p_u. When p_j finds out being the new master, it combines its own failure assumptions with those of foreign processes and sends a view change message. Process p_u frequently probes p_j for availability as long as it has not received any view change message. If p_u does not receive further acknowledgment messages, it assumes that p_j has failed and continues probing for the next available successor. It could also be that p_u receives a new view change message excluding the old master before it receives any confirmation from one of the probed processes. In this case, it simply sends the process down message to the new master from which it received the view change.

6.5 Self Test Message

With a self-test, a mobile phone checks whether it is connected to the network. A phone does so by sending a self-test SMS to itself. SMS does not distinguish between a message sent to a foreign phone number or the own phone number. It will always use the network carrier to send the message. Thus, we can use SMS to test our own network status. As long as a phone is able to send and receive a self test-message, it is also able to receive foreign messages. If a phone does not receive the own self test message (identified by a random number), we can assume that this phone is currently disconnected from the network and, hence, we can avoid wrong failure assumptions. Thus, after not receiving its own self-test message, it will suppresses all process down and heartbeat messages until connectivity is re-established and the self-test message is received.

6.6 Down Status

Mobile phones can be frequently disconnected for short time periods, for instance, while its user takes the metro for two stops. The network carrier forwards messages sent to a disconnected phone after reconnection. We do not want that short disconnections completely expel a phone from the group. Therefore, we take a two-step approach for removing phones from group activity. When the failure detection mechanism is triggered for a process p_i from which no heartbeat messages are received anymore, p_i is removed from the circle of responsibility. This leads to a view change message excluding p_i. However, the remaining processes keep p_i's phone number and set a *down* flag. They continue sending the application messages to p_i. If p_i does not reconnect within a certain time period, p_i's phone number will be completely deleted and no more messages sent to it.
At the same time, p_i itself will detect that it is disconnected as it does not receive any heartbeat messages from its predecessor and performs a self-test which fails. It will set itself to down status and queue all messages that the application wants to send. It also informs the application that there is a disconnection. If p_i does not become connected within a certain time period, it drops all queued messages and informs the application about being removed from the view. When p_i becomes connected it will receive all messages sent to it, including the view change excluding itself. It will deliver all received application messages. These might not be all messages sent within the view during the downtime because each process handles down flags individually, but the application is aware of this best effort since it received the temporary disconnection message. From there, p_i will join again and then send any message it might have locally queued.

6.7 Reasoning of correctness

In this section we argue about the correctness of our approach by showing that many common failure cases are handled correctly by our approach. We will illustrate some of these failure cases by assuming a group of six processes A, B, C, D, E, F. Process A is the group master. In each of the situations below, we assume there are no further joins, leaves and failures than the ones explicitly mentioned. The descriptions provide some special actions in failure cases that we haven't described previously for simplicity. For space reasons, we do discuss some cases only very shortly.

In the following, we first assume that all processes that are suspected to have failed, have actually failed (or are disconnected from the network), and do not recover until the view change protocols have terminated. We will later discuss false suspicions and the effects of intermittent connectivity. In this case, we will show situations where our protocol will not work correctly.
One Failure. Assume only one process p_i fails. Then p_i's successor p_{i+1} will detect the failure by not receiving heartbeat messages from p_i. If p_i is not the master it forwards a p_i-down message to the master (not needed if p_{i+1} is the master) and the master will send a new view change message. Everybody adjusts the circle of responsibility guaranteeing that process p_{i-1} monitored by p_i will receive as new monitor p_{i+1}. Although all nodes will still send application messages to the failed node for a time period after exclusion (as long as the down flag is set), the failed process is removed from the view. If p_i is group master p_{i+1} will detect the group master failure and immediately elect itself as the new group master, because p_{i+1} is the direct successor of p_i. All adjust their view and the circle of responsibility. After the takeover there is again only one master in the system and every node has a monitor.
Several Failures (not Master). Assume some processes fail, but not the group master. If the failures are not consecutive corresponding to the circle of responsibility, they will be detected concurrently. Every monitor process detects the failure of its predecessor and forwards the information to the group master. When master A receives the down messages, it might bundle several changes in one view change message.

As no consecutive processes fail, the adjustments to the circle of responsibility are independent of each other. If there are consecutive failures (for e.g., p_i and p_{i+1}), the last process in row (p_{i+1}) will be detected first (by p_{i+2}). After a

new view was sent and the responsibilities were adapted, the next process (p_i) will be detected (again, by p_{i+2}) and so on.

Concurrent Joins and Leaves. Concurrent joins and leaves are not a problem. The master process serializes them. If a process p sends a join request to a leaving process, the leaving process does not need to react. Process p will timeout receiving the view change and send the join request to another process.

Concurrent Join and Failure (not Master). Assume a view $V_i = \{A^*, B, C, E, F\}$ (with identifier i) and process D joins the group. If D sends the join request to a process that does not fail, this process forwards the join request to A. At the same time, the monitor of the failed process sends a failure message to A. Both events are serialized by master A and distributed by one or two view change messages. If D sends the join request to the failed process, it will time out and resend the request. In fact, non-master processes might fail in any combination concurrently to the join, and the master might combine view changes or send one after the other, all failed processes will be detected and removed and at the end the circle of responsibility is set correctly at all remaining processes.

Let's have a look at some interesting cases. If joined and failed node are consecutive in regard to the circle of responsibility, e.g., D joins and E fails, and A sends $V_{i+1} = \{A^*, B, C, D, F\}$, then the predecessor C of the failed node will send the heartbeat message both to the new node D and the successor F of the failed node. F will be monitor for both processes until it receives the first heartbeat message from the joining process D.

Now assume D joins the group and C fails shortly after. Before the join of D, process E was responsible for C. After the join, D is responsible for C. If E detects the failure before receiving the new view $V_{i+1} = \{A^*, B, C, D, E, F\}$ it forwards the failure to A. A sends a new view V_{i+2} excluding C, and D starts monitoring B. If E detects the failure after receiving V_{i+1} but before receiving the first heartbeat message from D, both processes might send the failure message to A, but E for sure. Independent of the originator, A excludes C and sends the new view V_{i+2}. If E receives the first heartbeat message from D before suspecting C as failed, D is the only process that detects and forwards the failure of C. A process join and concurrent failures can be handled, even if the joining process fails immediately after V_{i+1} is installed. The reason is that the successor of the joining process only stops monitoring the former predecessor once it knows that the joining process has started monitoring.

Several Failures including Master. If the master's monitor does not fail, it detects the failure and elects itself as the new master. Concurrently, other nodes might send process down messages to the old master due to other failures. If they time out receiving the corresponding view change they forward the failure assumptions to the master's monitor. In this case, the first view change from the new master will contain these changes. If they receive the new view change from the new master first, they resend their process down messages to the new master.

A special case occurs when the old master crashes in the middle of sending a view change and some but not all all members have received the new view V. If it has sent the view to its successor before the crash, then, the successor will receive V before it suspects the master to have failed (as the view change can be considered as a heartbeat message that is sent before it was actually due). In this case, processes that have not received V will be updated when they receive the first view change V' from the new master, as it subsumes the changes of V. Another issue is that messages from different nodes do not obey causal order. Thus, it is possible that a process receives V' before V. But as V' has a larger identifier than V (by one as the counter is increased upon each view change), V can be safely ignored. If the master has not sent the view V to the successor before the failure, then the first view change V' sent by the new master will not contain the changes of V. Furthermore, it will also not have a larger identifier than V but the same identifier (as at most the sending of one view change might be incomplete). It is important that other processes eventually install V'. If a process receives first V' and then V, it can detect that V is an old view as it has the same identifier as V' and V' indicated a master move which is not the case for V.

If the group master and at least one of its successors fail, the first available successor process will elect itself as the new master.

Concurrent Join and Master Failure. Compared to the case of a concurrent join with a non-master failure, there is only one additional situation to consider. Assume D is the joining node, and the master fails after having sent the view change V including D to some but not all processes. In this case, D will not receive the view change V as the master sends the view change last to the joining process. D will time out and resend its request to another process. For space reasons, we only discuss one scenario in which A is the master and B its monitor. (a) If B does not fail and has not received V, it sends a new view change V' (with the same identifier) neither including A nor D. When D eventually resends its request, it will succeed. (b) If B does not fail and has received V it sends a new view change V' (with a larger identifier than V) that excludes A but includes D. There are now two things going on concurrently. First, E will timeout receiving a heartbeat from D and request the exclusion of D to the master. The master will send a new view V''. Second, D resubmit its join request to another process, e.g., F. If F receives V'' before D's request, it will forward the join request to the new master, and D will eventually join. If F receives D's request before V'' it actually assumes D to be already a member and ignores the request. D will again time-out and has to resend the request.

Concurrent Leave and Master Failure is in spirit similar what was discussed for the join.

Master Failure during Master Move. Assume B initiates a master move B sends a new view V, excluding A if it had failed, and setting itself as the new master. If B fails after having sent V to C, C elects itself immediately as new master. If B fails before sending V to C, C sends a B down message to A. If A has not failed, it will inform C that it is no more master, and C will take over as master. If A has failed, C will time out, and, being the next successor, take over as master. Failure of C will trigger similar action at D.

Wrong timeouts. Timeouts have to be set conservatively, as the system might not continue to work correctly if timeouts are chosen too short. Let's go through some examples. If a non-master node is wrongly suspected due to a too short timeout, it will be excluded from the view. As it is informed about its exclusion, it can rejoin. If the master node A is wrongly suspected by B, two master processes exist concurrently. A will know about the takeover, when it receives the

new view V from B that excludes A. It can simply rejoin. A problem occurs if there have been concurrent joins, leaves or failures and they were still sent to A before A has received the exclusion message. It handles them and can send views that are truly concurrent to V. If there are several, their identifiers can be even larger than V. And they can arrive at the different nodes before or after V. V should eventually be the view installed at all processes. The processes can detect views concurrent to V as they have the same or a higher identifier as V and come from the old master.

In general, however, while basic sequences of failures and wrong suspicions might be handled, there will be situations, where two masters might start acting independently. In the worst case, they might come up with disjoint views, and then the system is partitioned.

Short Disconnections. If a node is only disconnected for short time, it might be suspected by its successor but actually be available again before the view change. Nevertheless, it is excluded and has to rejoin. Similar problems as with too short timeouts might occur.

7. PERFORMANCE ANALYSIS

In this section we provide an overhead analysis for simple multicast messages, and single joins and leaves. We consider both the number of messages as well as the communication steps needed to finish the operation. The overhead of heartbeat messages is ignored. We assume we start with a group of n phones. Each **multicast** takes n-1 messages. As messages can be sent concurrently, there is only one time step. For a **join**, assume the joining process does not contact the master directly but another process (1 message; 1 step). This process forwards the request to the master (1 message; 1 step). The master sends the message to all members in the new view except itself (n messages; 1 step). Once the successor of a joining process p receives the first heartbeat from p (1 step) it sends a stop heartbeat message to p's predecessor (1 message; 1 step). Thus, we have a total of $n+3$ messages in 3 steps until the joining process is included and 5 steps until the circle of responsibility is completely adjusted. A **leave** request takes n messages and two time steps. One message for the request itself and n-1 view change messages to all group members except the master. The same holds for **failures** as there is one process down message and n-1 messages for the view change. These two time steps, however, do not contain the delay until a failure is detected.

8. IMPLEMENTATION

Our GCS layer and a corresponding chat application layer have been fully implemented based on Java ME [9]. We decided for Java ME as a very common environment for applications running on mobile devices. It allows us to test our GCS on many different devices. Additional toolkits [10, 5] for Java ME supported our analysis. Java ME is divided into two base configurations: *Connected Limited Device Configuration (CLDC)* and *Connected Device Configuration (CDC)*. We use CLDC as it is designed for devices with limited capabilities like mobile phones and best fits our purpose. For incoming messages, we utilize a synchronous message listener that listens at SMS port 2000. Thus, messages are redirected to the GCS layer and do not end up in the mailbox of the user. We have thoroughly tested scenarios with up to four phones.

9. CONCLUSIONS

This paper presents a novel, completely decentralized group communication architecture for mobile devices that uses SMS based message passing. It's main target application is chatting but we believe that it can be used for other applications with similar reliability requirements. The system has a thorough failure detection mechanism that keeps the overhead for failure handling very low while at the same time handles the most common failure scenarios. Our approach handles short disconnections as this is a common phenomenon in mobile environments. Furthermore, failure handling is equally distributed over all nodes, and while view changes go through a master process, it is easy to rotate this task among the group members. For future work we will focus on integrating additional communication channels and, hence, supporting a wider spectrum of applications.

10. REFERENCES

[1] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical report, 1998.

[2] K. P. Birman et al. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[3] K. P. Birman, A. Schiper, and P. Stephenson. Lightweigt causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

[4] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[5] S. Ericsson. SDK 2.5.0.3 for the Java ME Platform. `http://developer.sonyericsson.com/`, 2010.

[6] P. T. Eugster et al. Lightweight probabilistic broadcast. *ACM Trans. Comp. Sys.*, 21(4):341–374, 2003.

[7] R. A. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, University of California at Santa Cruz, 1992.

[8] M. Larrea, S. Arevalo, and A. Fernandez. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Symp. on Distributed Computing (DISC)*, pages 34–48, 1999.

[9] S. Microsystems. Java ME. `http://java.sun.com/javame/index.jsp`, 2009.

[10] S. Microsystems. Java Wireless Toolkit. `http://java.sun.com/products/sjwtoolkit/`, 2009.

[11] L. Moser et al. Lingley-papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39:54–63, 1996.

[12] R. Prakash and R. Baldoni. Architecture for Group Communication in Mobile Systems. In *IEEE Symp. on Reliable Distributed Systems (SRDS)*, 1998.

[13] G.-C. Roman, Q. Huang, and A. Hazemi. Consistent group membership in ad hoc networks. In *Int. Conf. on Software Engineering (ICSE)*, pages 381–388, 2001.

[14] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distrib. Comput.*, 10(3):149–157, 1997.

[15] SMSGupShup. SMS Gup Shup Chat. `http://www.smsgupshup.com/apps_chat`, 2009.

[16] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Commun. ACM*, 39(4):76–83, 1996.