Distributed Group Communication System for Mobile Devices based on SMS*

Bettina Kemme[1] and Christian Seeger[2]

[1] McGill University, School of Computer Science,
3480 University Street, Room 318, Montreal, Canada
kemme@cs.mcgill.ca,
[2] Technische Universität Darmstadt, Department of Computer Science,
Databases and Distributed Systems Group,
Hochschulstraße 10, 64289 Darmstadt, Germany
cseeger@dvs.tu-darmstadt.de

Abstract. This paper presents a group communication system for mobile devices, called DistributedGCS. Mobile communication is slow, expensive and suffers from occasional disconnections, especially when users are moving. DistributedGCS 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, DistributedGCS 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 DistributedGCS. 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 DistributedGCS can handle the most common failures properly while keeping the message overhead very low.

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 would allow mobile phones to participate in group communication. Two obvious applications are chat among a group of friends or

* The work in this paper is based on an earlier work by Christian Seeger, Bettina Kemme and Huaigu Wu: SMS based Group Communication System for Mobile Devices, that appeared in the Proceedings of the ACM Workshop on Data Engineering for Wireless and Mobile Access, (c) ACM, 2010.

business partners, or information dissemination among a group of people with similar interest (e.g., a common research project).

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 and continuously available. 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 for 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, 16] 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 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 and distribute it over all phones, we introduce the concept of *circle of responsibility* as our failure detection system. Group membership changes can be handled and propagated by every group member which automatically distributes the group maintenance overhead.

2 Background

This section depicts different aspects of GCSs and the network environment of mobile devices which we rely on. Additionally, we introduce the GCS requirements of a chat application and close this section with related work.

2.1 Group Communication Systems

GCSs are implemented as a layer between the application and the network layer and provide 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 in 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, requiring several rounds of message exchange among all members in order to guarantee that all members agree on the same view. Even more advanced, certain services provide a logical order between view change messages and application messages delivered in each view, such as *virtual synchrony* [16]. In this case, the view change protocol also has to agree on the set of application messages to be delivered at each process.

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 if the application layer of a process sends two messages, then these messages are delivered in the order in which they were sent. *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 afterwards) 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.

2.2 Network Environment of Mobile Devices

Mobile devices, especially mobile phones, usually connect to stationary *base stations* provided by network carriers which provide mobile devices with different speech and data services. The most common data services for mobile devices are *SMS, MMS, GPRS* and *UMTS*. SMS and MMS are services designed for direct data communication among mobile phones. Messages are addressed to 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 and this we want to avoid. 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 (and MMS) are an often higher message delay than for IP packets and a payment per message independently of the size of the message.

2.3 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 fulfills 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 easily 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. Similarly, 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 propagating information to more than 20 users in acceptable time. 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.

2.4 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 [12], Isis [2], Horus [8] 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 [14] 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 [13] 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 and forwards changes to the *Group Membership Layer*. Based on this information a three-round group membership protocol builds a group of mobile nodes.

Closest to the approach presented in this paper is SMS GupShup Chat [18]. SMS GupShup Chat 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.

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 [3, 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 than the ones provided by epidemic protocols.

The work presented in this paper, DistributedGCS, is based on MobileGCS [17]. Message dissemination and failure detection are very similar in both systems but in MobileGCS the group maintenance relies on one specific phone, called master phone. Since the master phone is responsible for distributing group changes, it suffers from a higher message overhead. Therefore, we introduced a *master move* operation for switching these responsibilities from one phone to another. Still, a master move costs additional messages that we want to save. Furthermore, if several membership changes occur the master phone could easily get overloaded. In DistributedGCS, every member can propagate membership changes and every member manages its own list of group members. This makes a master phone and a master move operation unnecessary. On average, membership changes cost the same number of messages in DistributedGCS as in MobileGCS, but DistributedGCS inherently distributes the overhead over all group members and saves the messages for master moves.

3 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.

3.1 Multicast

We do not provide reliable message delivery to all available nodes. This would require a node to store messages that 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.

3.2 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, simply leads 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.

4 DistributedGCS without Failures

DistributedGCS provides a totally distributed group maintenance service. All group members are equal and allowed to handle group operations such as join and leave requests. In contrast to the predecessor of DistributedGCS, MobileGCS [17], where group maintenance was coordinated by a master phone, the additional overhead for group maintenance messages is distributed over all processes. This makes costly operations for changing the group master unnecessary. On the other hand, having a master process simplifies the group maintenance. In MobileGCS, when a mobile phone wants to join or leave a group, or if a failure occurs, the corresponding request is sent to the group master, which decides on a new view configuration and sends the new view to all affected phones. As a result, every process that receives the view change has a consistent view with the group master. As long as every message sent by the master is received by all members and the master does not fail, all members install the same sequence of views. In contrast, DistributedGCS allows every process to handle membership operations and to change the view. This prevents overloading a single phone, but makes it more difficult to find a consensus if two or more processes change the view simultaneously. Nevertheless, DistributedGCS eventually achieves the same view among all members after a feasible amount of time.

With regard to a chat application, we assume that it is more important to keep all active processes in the view than excluding left or failed processes. Thus, the view management of DistributedGCS has a higher priority for keeping processes than for excluding them.

4.1 Group Maintenance Service

In DistributedGCS, every process has to maintain the group membership on its own. Although not every process necessarily receives the same messages in the same order, the views of all processes should eventually converge if there are no further configuration changes. This is the main challenge DistributedGCS has to deal with. For this, we first describe the basic communication schemes in the case of only one event at a time. After that, in *View Management*, we explain the processing of more complex events.

In order to identify a process' status change, DistributedGCS uses additional flags behind process identifiers / phone numbers. A flag represents the status a process p_i has stored about a foreign process p_j. It might be that two processes have different flags stored for one process p_j. We distinguish between three different flags.

– “u” - up: process is in the group and has not changed its status recently
– “j” - join: process has recently joined the group
– “l” - leave: process has recently left the group

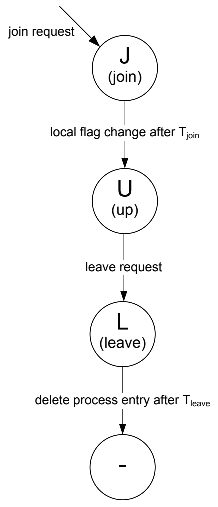
join request
J
(join)
local flag change after T_join
U
(up)
leave request
L
(leave)
delete process entry after T_leave
-

Fig. 1. Flag changes in DistributedGCS.

Figure 1 shows how flags are changed in DistributedGCS. A view change message consisting of a set of processes indicates that a process (e.g., D) has joined by tagging it with a join flag (e.g., D_j). When a process p that is already group member receives the view change message it first keeps this flag for the newly joined process. After a local timeout T_{join} exceeds, p sets the new process' flag set from join to up (e.g., from D_j to D_u). This is done individually at every process. As long as a process has an up or join status, it is considered a member of the group. If a process leaves the group, a new view change message is distributed marking the leaving process with a leave flag (e.g., D_l). When process p receives this message, it changes the leaving process' flag to leaving (e.g., D_l) but it does not immediately remove the process from its view. It is important to keep track of an already left process for a while in order to avoid that it is mistakenly added again. After T_{leave} time passes at process p, it finally removes the leaving process from the view. For simplification in the following figures, a process without a flag has always an up flag.

Create/Destroy Since we avoid the usage of a central server the existence of a new group has to be propagated. The idea is that when a user wants to create a group, it invites other phones to be members of the group. This means group creation is combined with group invitations. This is useful for chatting as it allows the creation of a new chat room and to invite other people to join it.

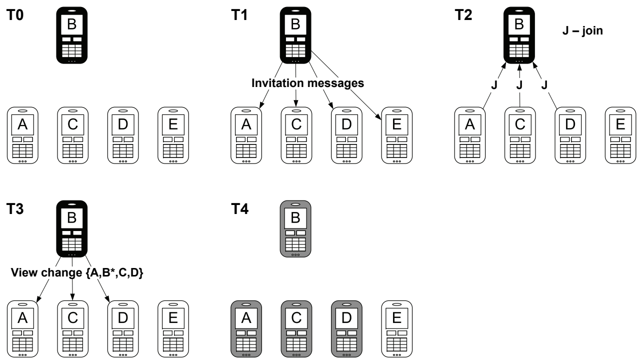

Fig. 2. Create

Figure 2 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. The phone numbers to be invited must be provided by the user. 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. For a chat application, group creation will open a chat room and invite others to join the group. A phone that calls the create method automatically becomes a temporary group master (black color) and the group creation is completed only including the calling phone as group member. The next step involves sending invitations to contacts that are chosen by the user. The chosen phones receive invitation messages including the group name from the temporary 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 it does accept the invitation, the GCS sends a join request to the initiating phone. In the given example, each phone, except for the phone E, sends a join request in step T2. In T3, the black phone adds all joining processes to the view and sends a view change message to all members of the new group. After that, the black phone stops acting as a master. The temporary master only waits a limited time to send the view change as described in the following section. If a further join request is received later, it simply sends a further view change message. 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 which is automatically done after/once the last processes leaves the group. Therefore, DistributedGCS does not provide a special destroy operation.

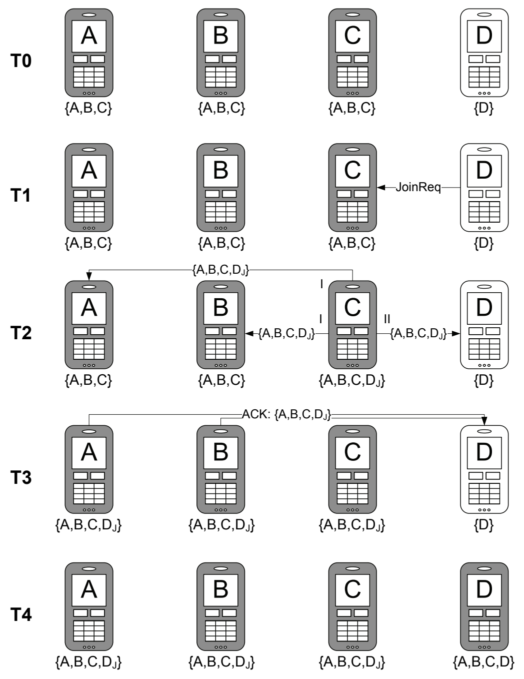

Fig. 3. Join

Join If a phone wants to join after the initial creation has completed, it has to send a join request to one of the group members. Our GCS processes join requests in a completely distributed manner. Figure 3 depicts a simple join request from process D. Time step T0 shows an already existing group of three processes A, B, C that have the same view $\{A, B, C\}$ ($\{A_u, B_u, C_u\}$ including status flags) installed. At T1, process D sends a join request to process C. As all processes are equal, D can send a join request to any group member. In this case, process C is requested and adds the new process D to its view and sets its flag to *join*: $\{A, B, C, D_j\}$. In the next time step process C sends the new view first to the old view members and then to the joining process. At T4, all group members that have received the view update send an acknowledge message to D and D checks whether the join succeeded or not.

The acknowledge messages sent to the joining process fulfill two requirements. First, they allow a joining process to check whether the join succeeded or not. And second, by attaching their originator's view, these acknowledge messages allow to capture further join requests. Figure 4 depicts an example that shows how two simultaneous join requests sent to two different phones are processed. Again, we start with a group of three processes A, B, C. Processes D and E want to join the group. Process D sends a join request to C and process E sends a join request to A at T1. In the next step, both join requests are processed in the same way as already described for a single join. The requested

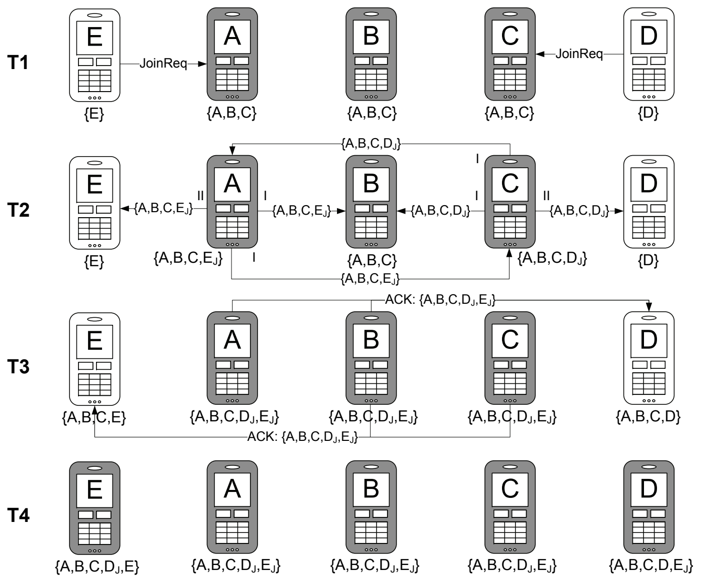

Fig. 4. Two Joins

processes add the new process to their views and set the *join* flag. Then, they send the resulting views first to the old members and then to the joining processes. Hence, process C sends $\{A, B, C, D_j\}$ to B, C, D and process A sends $\{A, B, C, E_j\}$ to A, B, E. In order to include all joining processes, every process that receives a foreign view builds a union of its own view and the incoming view. Process B, for example, has the view $\{A, B, C\}$ installed and receives both update messages in T2. Assuming the message from C is processed first, it calculates the following view: $\{A, B, C\} \cup \{A, B, C, D_j\} = \{A, B, C, D_j\}$. Then, it processes the second update message sent by A and builds the following view: $\{A, B, C, D_j\} \cup \{A, B, C, E_j\} = \{A, B, C, D_j, E_j\}$. The order of incoming update messages does not affect the result of a union. T3 in our example highlights the reason for attaching the current view to the acknowledge message. At T2, process C was not informed about E when it sent the update message to D. Hence, the update message to D does not include process E. However, at T3 the processes A and B already added E to their views and attached them to their acknowledge messages. On receiving the acknowledge messages from A and B, process D gets informed about the new process E and adds it to its view. And process E gets also informed about the new process D by receiving acknowledge messages from B and C. This way, attaching views to acknowledge messages enables DistributedGCS to detect simultaneous joins. At T4, the processes D and E have joined the group and all processes have the same view installed.

Leave Figure 5 shows how a leave request is processed. In the first step, the leaving process D sends a leave request to any group member (C in our example).

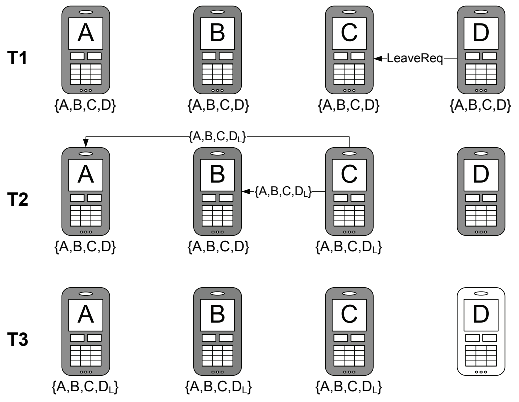

Fig. 5. Leave

At T2, process C changes D's flag from *up* to *leave* and propagates the view change among all members. Process D does not get an acknowledge message for its request. We let another process than the leaving process propagate the leave request to make this procedure similar to what is done when nodes fail (cp. Section 5). At time step three, every available process has set D to *leave* and, therefore, D is excluded from the group.

4.2 View Management

Building the union of two views only works as long as two incoming views do not carry different flags for the same process. In the case of different flags, a consensus among all processes has to be found. DistributedGCS does not have a group master for conflict resolution and we also want to avoid expensive view proposal algorithms. Therefore, we will present a *view management* scheme that eventually finds a common view with local decisions and with a minimum of message exchanges among processes. It does not use any kind of voting algorithm that guarantees that every member installs the same view. Our goal is to find a common view with local decisions and without sending any additional message. Incoming views are processed sequentially. A foreign view can be received as a *view change* message, an acknowledge message after a join request or as a *safety* message. A safety message is the view from a process to which the process' own view was sent before. It is the response to a strong inconsistency between two views. For example, an up message about a process which has already left the group. Safety messages are not necessary, but they accelerate finding a common status.

As said before, each process can have one of three different flags (*up, join, leave*) in both the current view of another process and in an incoming message. Figure 6 shows how a process p_i changes the local flag of another process p_j triggered by an incoming view or triggered by one of the timeouts T_{join} and T_{leave}. The first entry in each row is the local flag of p_i for the process p_j ("-"

local flag	incoming flag			timeout	
	J	U	L	T_{leave}	T_{join}
J	J	J	J		U
U	U	U	L		
L	J	L	L	-	
-	J	U	-		

Fig. 6. View change table of p_i for process p_j.

stands for no entry). The following columns show how p_i changes p_j's flag upon receiving a foreign view or when a timeout exceeds.

First, we take a look at incoming views. For simplification, we say a *local flag* is the locally stored flag for a process p_j and an *incoming flag* is the flag for this process p_j received from another process p_u. If the local flag and the incoming flag are the same, nothing has to be done. If there is an incoming join flag and the local flag is leave or there is no entry, the local flag is set to join. If the local flag is up, the incoming join is ignored. Incoming up flags are ignored unless this process is not in the local view. In this case, a process is added with an up flag set. If the local flag for a process p_j is join and p_i receives a leave from p_u, p_i sends a safety message to p_u and does not change the local flag. In the case of a local up and an incoming leave flag, a process is set to leave. A second reason for a safety message is an incoming up flag when the local view has the leave flag set.

Upon receiving an incoming leave of a process p_j a local timer t_{leave} is started. As soon as t_{leave} exceeds T_{leave} process p_j is finally deleted from the local view and incoming leave flags for p_j are ignored. If a process deleted p_j immediately upon receiving the leave message, an incoming view that still contains an up flag for p_j would add the already left process p_j to view again. Therefore, p_j stays in view for T_{leave}. Upon receiving the incoming join of a process p_j a local timer t_{join} is started. As soon as t_{join} exceeds timeout T_{join} the local flag for p_j is changed from join to up. Keeping a local join flag for a while is not necessary but helpful in order to inform simultaneous joining process (cp. Figure 4) about the other recently joined process(es). Assuming that processes can fail, this timeout becomes more important and will be discussed in the next section.

5 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 on 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 very 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.

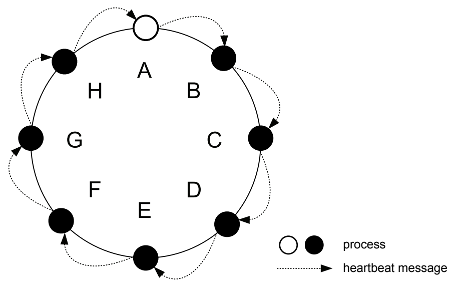

Fig. 7. Circle of Responsibility

The authors in [9] and [15] introduce distributed failure detectors that distribute the workload for failure detection to more than one failure detection module. 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 7 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.

5.1 Failure of a Process

If an expected heartbeat message is missing for a period T (T is significantly 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 failure* message to the group. This means that it marks the suspected process as *down* in its view and distributes the new view among all group members of the new group. It also sends the new view excluding the suspected process to the suspected process. 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 failure message to the group. The self-test allows B to detect whether it is currently connected and is described in Section 5.4.

5.2 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 circle of responsibility which 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.

5.3 Adapting to Process Joins

If a process joins the group, the responsibilities change and the circle of responsibility has to adapt to it. In order to avoid a gap in the circle of responsibility, a joining process p_i is only then completely included into the circle when p_{i+1} actually knows that the join was successful and p_i becomes a permanent member of the circle. For this, a joining process gets a temporary status first. Upon receiving the first heartbeat message from this process, it is assured that the join succeeded. 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.

5.4 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 its 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 suppresses all process down and heartbeat messages until connectivity is re-established and the self test message is received.

5.5 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. The *down* flag is similar to the *leave* flag with the exception that processes with the leave flag will not receive any application messages anymore as they left the group voluntarily and explicitly.
At the same time, p_i itself detects that it is disconnected as it does not receive any heartbeat messages from its predecessor and performs a self-test which fails. It sets itself to down status and queues 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 receives all messages sent to it, including the view change excluding itself. It delivers 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 receives the temporary disconnection message. From there, p_i joins again and then sends any message it might have locally queued.

6 Reasoning for Correctness

In this section we argue about the correctness of DistributedGCS. For this, we show that many common join requests, leave requests and failure cases are handled correctly by our approach. But we also show that some cases in DistributedGCS are not handled as well as they were in MobileGCS that we presented in [17]. We will illustrate some of these failure cases by assuming a group of six processes A, B, C, D, E, F. In each of the situations below, we assume there are no further joins, leaves and failures than the ones explicitly mentioned.

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. As a result, p_{i+1} will set p_i to down and send a view change message. As no further process fails, all these actions will succeed, and 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.

Several Failures. Assume some processes fail. 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 sends a view change message. For example, if processes B and D fail, C detects B's failure and E detects D's failure. C sends a view change message setting B to down and E sends a view change message setting D to down. Theses view change messages are sent to every member and 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. For our example, if B and C fail, D first detects C's failure. After C is excluded, D becomes monitor of B. But as B has also failed, it does not receive the view change and does not send heartbeats. Thus, D detects B's failure. Additional non-consecutive process failures are detected concurrently.

Concurrent Joins and Leaves. Concurrent joins and leaves are not a problem. As shown in Figure 4, concurrent joins are handled simultaneously. The same applies for one or more simultaneous leave requests. Since a joining process waits for acknowledge messages of members, the missing acknowledge message of a leaving process could be a problem. There are three cases we have to analyze. Assume a group of five processes A, C, D, E, F which have the view $V_i = \{A, C, D, E, F\}$ (with identifier i) installed. Process D sends a leave request to C and process B wants to join the group. Let's take a look at three cases. First, process B requests the leaving process D to join the group and D does not react. Process B will timeout receiving the view change and send the join request to another process. Second, B requests process A and A distributes the new view $V_{i+1} = \{A, B, C, D, E, F\}$. Process D sends a leave request to C. If C receives V_{i+1} after the leave request, C's acknowledge message to B already contains the leave request of D and, hence, B does not wait for D's acknowledge message. Third, if C receives V_{i+1} before the leave request, C automatically forwards D's leave request as it has V_{i+1} already installed. Therefore, concurrent joins and leaves are handled properly.

Concurrent Join and Failure. Assume a view $V_i = \{A, B, C, E, F\}$ and process D joins the group. If D sends the join request to a process that does not fail, this process sends a new view including D. At the same time, the monitor of the

failed process sends a view excluding the failed process. Similar to a concurrent leave request as explained in the previous paragraph, D gets informed about the failure and does not expect an acknowledge message from the failed process. The join request and the failure detection will succeed.
If the successor p_{i+1} of a joining process p_i fails, p_{i+2} will detect p_{i+1}'s failure. After excluding p_{i+1}, the circle of responsibility adapts. Hence, p_i monitors p_{i-1} and p_{i+2} monitors p_i.
If the predecessor p_{i-1} of a joining process p_i fails and p_i has not sent its first heartbeat message, then p_{i+1} will detect the failure and p_i might detect it. Both processes send the same resulting view. If p_i has already sent its first heartbeat message, only p_i excludes p_{i-1} from view and sends a view change message. In both cases, p_{i-1} will be correctly excluded from the view.
In fact, processes might fail in any combination concurrently to the join, and as long as the process that processes the join request does not fail, every process might combine view changes. All failed processes are detected and removed and at the end the circle of responsibility is set correctly at all the remaining processes.
Let's have a look at an interesting case. If D requests a process, e.g., A, that fails while it sends the view change message V_{i+1} including D, D will timeout receiving the view change and send the join request to another process. Within the failure of A and the second join request of D exists a period of time with different installed views. Some processes have already installed V_{i+1} and some have V_i installed. If D's successor E has already installed the view V_{i+1}, it detects that D is not in group and send V_{i+2} excluding D. If E and C have still V_i installed, the unsuccessful join of D is not detected. If D does not try to join the group again, there will be two different views installed until a new view change message is sent. In the current version of the system, we do not handle this problem properly. Since heartbeat messages contain the originator's view, a view including D will be propagated slowly and D isexcluded when E is receiving this view. This might take some time.

Failure while Sending View Change Message The previous example depicts a general problem of DistributedGCS. The distributed approach works fine unless a process fails during view change transmission. In MobileGCS, if the master process fails, a new master is elected and sends a new view to all members. The previously installed views are dropped and, thus, it is not important which view a non-master member had installed before. In DistributedGCS, a new view is the union of the previous view and the incoming view. Therefore, it is important which view was installed before. If a process fails while sending a view change message containing V_i, some processes have more actual information than others. If a further view change V_{i+1} does not contain the information sent before because its originator has not received V_i, group members have still different views installed. Exchanging view information by sending heartbeat messages helps finding a consensus, but since they are only sent to a process' successor, the information flow is very slow.

In particular, there are two cases of missing events: a mistakenly included (join) or a mistakenly excluded (leave, down) process which is only installed at a subset of group members. Let's start with the mistakenly included process. Assume a process p_i likes to join the group and process p_j which processes the join request fails while transmitting the view change and, thus, the view change is not distributed among all group members. If p_i retries to join the group and the new requested process does not fail while transmitting, the join will succeed. If p_i does not retry to join, there could be two situations. First, p_{i+1} already received the view change from p_j and detects a failure of p_i. Process p_{i+1} will send a new view excluding p_i and p_i is completely excluded from the group. Second, p_{i+1} has not received the view change from p_j but at least from one other group member. In this situation, p_{i+1} does not monitor p_i and, hence, it does not recognize the failure of p_i, but other processes which have received the view including p_i assume that the join succeeded. In this case, p_i is mistakenly included in some views. By adding the current view to each heartbeat message and treating inclusions of each incoming heartbeat message like a view change, the group members that have received p_i's join will propagate its inclusion to their successors along the circle of responsibility. With every set of heartbeat messages sent among the group the mistaken inclusion of p_i is forwarded towards p_{i+1}. In worst case, if n denotes the total number of processes in the group and p_{i+2} is the only process that has received the join of p_i, it takes $n-2$ steps until p_{i+1} is informed about the mistaken inclusion of p_i and, thus, it will exclude p_i because it does not receive heartbeat messages from it. In summary, if no further view changes occur, a mistakenly added process is eventually detected after $n-2$ heartbeat steps in the worst case.

A mistaken exclusion could occur when a process p_i suspects its not-failed predecessor p_{i-1} to have failed and fails while transmitting the view change. As a result of p_i's failure, p_{i+1} will send a view change message excluding p_i. If p_{i+1} received the view change from p_i before it failed, the failure assumption of p_{i-1} will also be propagated within the group and p_{i-1} will be excluded. If p_{i+1} has not received the view change before, it will propagate a view including p_{i-1}. Processes which have received the previous exclusion of p_{i-1} will ignore the *up* information until $T_{leave/down}$ exceeded and they have completely deleted p_{i-1} from their view. Once p_{i-1} is deleted, an incoming heartbeat message including this process will add it again. In worst case, a mistakenly added process will be eventually detected after $n-2$ heartbeat steps as soon as $T_{leave/down}$ is exceeded. We would like to mention here that exclusion information carried by a heartbeat message are discarded which emphasizes our assumption to focus more on the inclusion than on the exclusion of processes.

7 Performance Analysis

We only want to provide a rough 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. In our analysis, we would like to make an assumption that we have a group of n phones. Each **multicast** takes n-1 messages as each group member receives its own copy of a message. As messages can be sent concurrently, there is only one time step. For a **join**, the joining process contacts a group member by sending a join request (1 message and 1 step). Upon receiving this request, a view change message is sent to all group members including the new process except the sending process itself (n messages in 1 step). All group members, except for the requested process, send an acknowledge message to the joining process ($n - 1$ messages and 1 step). Once the successor of the joining process p receives the first heartbeat from p (1 step) it sends a stop heartbeat message to p's predecessor (1 message and 1 step). Thus, we have a total of $2n$ 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 $n - 1$ messages and two time steps. One message for the request itself and $n - 2$ view change messages to all group members except the requested process. For processing a **failure**, DistributedGCS takes one process down message and $n - 1$ messages for the view change to all group members including the failed process. Thus, we have a total of n messages and 2 steps. 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 [10]. We decided for Java ME as it is a very common environment for applications running on mobile devices. It allows us to test our GCS on many different devices. Additional toolkits [11, 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 on a testbed consisting of 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. Compared to our MobileGCS, DistributedGCS inherently distributes the management overhead and does not need special master move operations. 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 handling 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. For future work we will focus on integrating additional communication channels and, hence, supporting a wider spectrum of applications.

References

1. Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. The Johns Hopkins University, 1998.
2. K. Birman and R. Cooper. The ISIS project: real experience with a fault tolerant programming system. In *EW 4: Proceedings of the 4th workshop on ACM SIGOPS European workshop*, pages 1–5, New York, NY, USA, 1990. ACM.
3. K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.
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/wportal/devworld/article/java-sdk-versionhistory`, 2010. [Online; accessed 28-October-2009].
6. P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.
7. R. A. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, Santa Cruz, CA, USA, 1992.
8. Horus. The Horus Project. `http://www.cs.cornell.edu/Info/Projects/HORUS/index.html`, 2009.
9. M. Larrea, S. Arevalo, and A. Fernandez. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proc. 13th Symposium on Distributed Computing (DISC'99), Bratislava (Slovakia)*, pages 34–48. SpringerVerlag, 1999.
10. S. Microsystems. Java ME. `http://java.sun.com/javame/index.jsp`, 2009.
11. S. Microsystems. Java Wireless Toolkit. `http://java.sun.com/products/sjwtoolkit/`, 2009.
12. L. Moser, P. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39:54–63, 1996.
13. R. Prakash and R. Baldoni. Architecture for Group Communication in Mobile Systems. In *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, Washington, DC, USA, 1998. IEEE Computer Society.
14. G.-C. Roman, Q. Huang, and A. Hazemi. Consistent group membership in ad hoc networks. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 381–388, Washington, DC, USA, 2001. IEEE Computer Society.
15. A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distrib. Comput.*, 10(3):149–157, 1997.
16. A. Schiper, K. Birman, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.
17. C. Seeger, B. Kemme, and H. Wu. SMS based Group Communication System for Mobile Devices. *ACM Workshop on Data Engineering for Wireless and Mobile Access*, 9, 2010.

18. SMSGupShup. SMS Gup Shup Chat. http://www.smsgupshup.com/apps_chat, 2009.