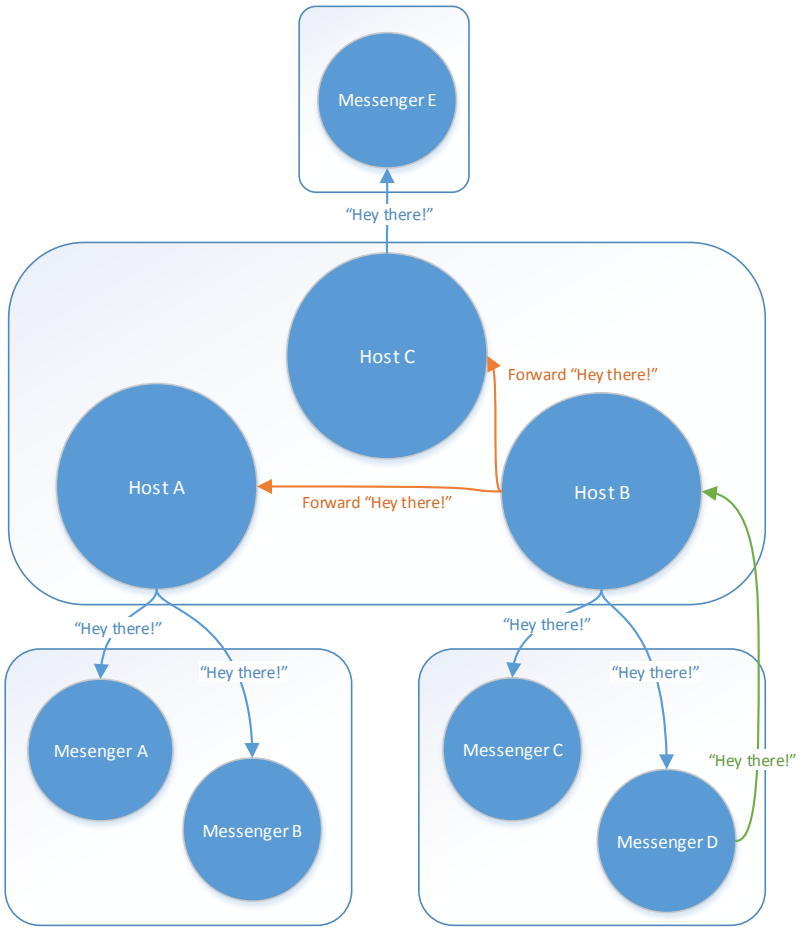
University of Groningen
Faculty of Mathematics and Natural Sciences
Department of Computing Science

Distributed Systems



MultiCom
A Decentralized Chat Messenger System

Authors
Gkortzis, Antonios (s2583070)
Hoekstra, Mark (s2015366)

Groningen, November 5, 2014

Contents

1 Context/Background

This report is dedicated to describe the system *MultiCom* written for the practical assignment for the course Distributed Systems. Our supervisors during the course were Prof. dr. ir. M. Aiello, and I. Georgievski.

MultiCom is a decentralized chat application written in Java. A user can open a MultiCom client and, after connecting, he or she send or receive messages from other users connected on the network.

The reason for choosing a chat application for this course is because it is simple to write and allowed us to implemented all the required techniques. More importantly, because it is a simple application to write, it allowed us to focus more on implementing the following requirements:

- Having a voting algorithm
- Dynamic discovery of hosts
- Reliable ordered multicast
- Tolerance for crash faults

These requirements are implemented in our project and next to them load balancing is implemented.

MultiCom is aimed at providing a chat service to users, in which they can perceive the system as a "black box" which allows them to chat without worrying about losing their connection or reliability. Next to reliability and transparency, MultiCom aims at providing performance and a basic level of integrity.

To give some background on distributed systems, the following question is asked: Why are distributed systems so popular? Distributed systems are great for use in projects which need to scale to a large number of users. The systems must be reliable, integer and it is expected that they have a high availability. This can be achieved in a system in which the responsibilities of crashed nodes are taken over by other nodes. These systems generally are transparent on many levels, by concealing errors and replacement of resources and clients.

This report is structured as follows. Chapter 2 contains the State Of The Art and describes related distributed systems. Following this Chapter is Chapter 3 in which the problem is defined. Hereafter Chapter 4 describes the concepts of MultiCom that are related to distributed systems. The solution is given in Chapter 5. This Chapter contains descriptions of the solution of each concept in the system. Finally, Chapter 6 concludes this report in which a retrospective is given. An Appendix Chapter at the end of this report contains an example of three users chatting with MultiCom.

2 State of the Art

In this Chapter we shortly present two projects which make use of several distributed system concepts and, just as MultiCom system, they provide communication between users. Both systems are commercial and widely used by the smartphone users' community.

2.1 iMessage

iMessage[1] is an instant messenger developed by Apple Inc, and allows users to send texts and other files through a network. It is based on the Apple Push Notification Service (APNS)[2] which is used to forward notifications of third party applications to the Apple devices.

2.2 WhatsApp

WhatsApp Messenger [3] is a cross-platform mobile messaging app which allows users to exchange messages through the Internet. WhatsApp makes use of the Extensible Messaging and Presence Protocol (XMPP) [4], a communications protocol for message-oriented middleware used by decentralized client-server applications. WhatsApp is one of the most popular message exchanging applications and it has recently been acquired by Facebook.

2.3 Comparison with MultiCom

Both iMessage and WhatsApp systems are implementing the reliable delivery concept by notifying the sender that their message has been delivered to the receiver. This requires an acknowledgment mechanism just like the MultiCom system is using. They are also decentralized with no need of a static one-to-one connection between a server and a client, thus providing mobility, performance and failure transparency.

However, iMessage and WhatsApp both provide more functionalities than MultiCom does, like choosing private chat rooms and the ability to exchange not only text messages but also various types of files.

[1] iMessage, `https://www.apple.com/ios/messages/`
[2] Apple Push Notification Service (APNS) protocol `https://developer.apple.com/technologies/ios/features.html`
[3] WhatsApp Messenger, `http://www.whatsapp.com/`
[4] Extensible Messaging and Presence Protocol (XMPP), `http://xmpp.org/`

3 Problem Statement

In order to create the decentralized distributed system MultiCom, architectural decisions had to be made. These decisions are based on the goals mentioned in Chapter 1, reliability, transparency, performance and integrity. Moreover, the decisions have been influenced by the requirements for this project.

3.1 Architecture

The MultiCom project consists of two entities. One entity is the chat client, from now on named Messenger, which allows users to send and receive chat messages once they are connected to the network. The other entity is the Host which on startup allows Messenger to connect to the network. Messengers and Hosts are connected to each other in multiple ways. Figure 1, shows how different entities are connected in the system.

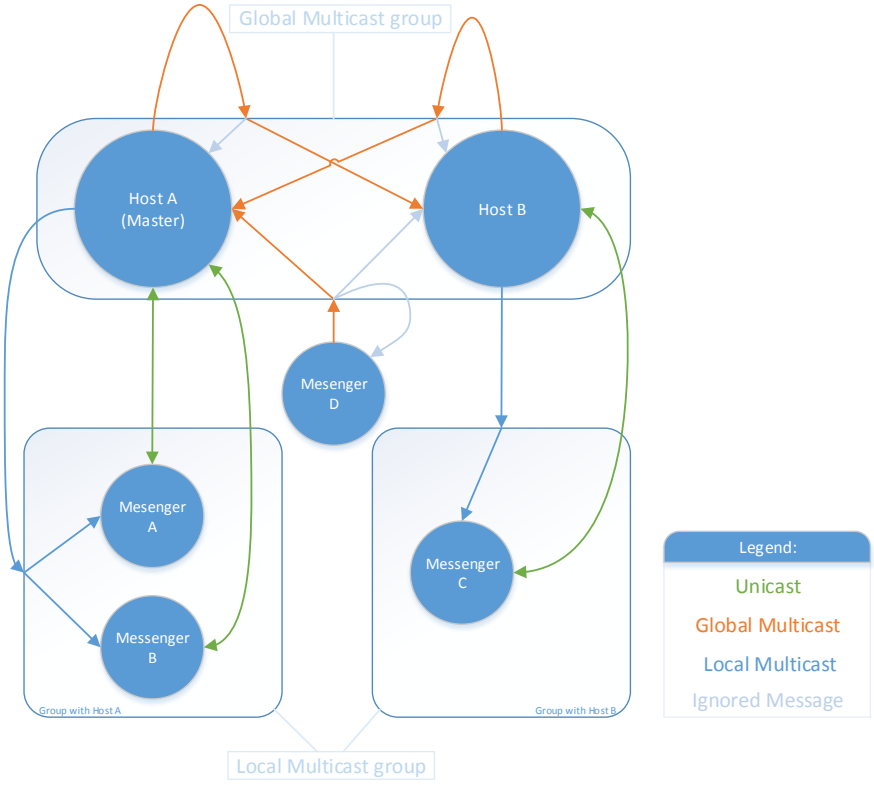


Figure 1: MultiCom's architecture

As can be seen in the above Figure 1 there are three different types of connections. Messengers are connected to a Host through a `Unicast` communication channel. This channel is unique for each Messenger-Host connection and is used by the Messenger to send *chat* messages to the Host and by both entities for sending and receiving *acknowledgement* messages. The second type of communication channel is the one between a Host and the Messengers connected to this Host. This type of communication is a Multicast group, which has one Host and zero or more Messengers as members, and is named `Local Multicast` throughout this report. Local Multicast is used, among others, by a Host to distribute *chat* messages received from Messengers

through Unicast or other Hosts. Receiving messages from other Hosts is provided by the Multicast group which contains Hosts as members and is named `Global Multicast` throughout this report. The Local and Global Multicast groups are closed groups and thus only members of the groups can send and receive messages.

Through these communication channels three types of messages are sent: *commands*, *chat* messages and *acknowledgements*. *Commands* are messages that can be sent by a Host or a Messenger and the receiver is expected to do a certain operation depending on the type of command. For example, a Messenger can send a *command* saying that it wants to connect to the network. The receiver of this command, a Master Host as explained in Section 4.1, must then find a Host for the Messenger to connect to. The second type of message is a *chat* message which is a message created when a *chat* message is sent by a user. The last type of message in the MultiCom system is the *acknowledgement*. As explained in Section 4.3.1 this type of message is used to increase the reliability of the system. If a *chat* message is sent, but no *acknowledgement* is received then the message will be resent.

3.2 Challenges

In order to achieve MultiCom's goals and make a distributed system that is perceived by the user as a single integrated computing facility, the following challenges have to be solved.

3.2.1 Message Passing

Entities in the distributed system, Hosts and Messengers, must have a way of communicating with each other. As mentioned before, MultiCom provides three types of communication channels and three types of messages. Basic multicast however does not ensure that message are delivered in the correct order or are delivered at all. To ensure ordering and increase the integrity of the system, MultiCom orders the messages as explained in Section 4.3.2. To ensure the delivery of messages and increase the reliability of the system, MultiCom lets receivers send *acknowledgements* to the sender upon receiving a message. This approach is further explained in Section 4.3.1.

3.2.2 Crash Failure

To increase the user experience, MultiCom aims at minimizing the hinder caused by failing entities. For example a Host may shut down because of a crash failure. When this happens the Messenger will automatically try to connect to a new Host, if any. This approach increases the transparency of the system and is further explained in Section 4.5.

3.2.3 Automatic and Dynamic Configuration

Several techniques, such as load balancing, leader election and dynamic discovery of hosts all explained in Chapter 4, must be implemented in order to have a system that is autonomous and balanced. All of these techniques help in creating a distributed system.

3.2.4 Synchronization and uniqueness of exchanged messages

In a distributed system there can be a lot of different entities sharing information via message passing. There must be some way to differentiate between messages from different entities, but also in some cases an entity

must be able to detect when it received or sent a message. Examples of when and why synchronization or differentiation between messages is needed are given in Section 4.5.1.

3.2.5 Concurrency

Every distributed system has operations which are performed at the same time. This also holds for MultiCom in which every Host performs several services (threads), each aimed at providing a specific functionality in the system. When designing MultiCom we had to take care of concurrent modifications performed by these different services. For example when a message is received and another message is sent concurrently.

4 Relation to Distributed Systems

In this Chapter we describe how concepts of a distributed system can assist in solving the challenges that we described in Chapter 3. For satisfying the requirements of this assignment we implemented the following features. All these concepts and the algorithms that we implemented in order to apply them are presented in details in Chapter 5.

- Leader elections (Voting algorithm)
- Dynamic host discovery
- Reliable multicast delivery
- Ordered delivery
- Load balancing
- Crash failure tolerance

4.1 Leader elections

The MultiCom system consists of several instances of the Host entity. When a new Messenger is requesting a connection to the system only the Master (Leader) Host is going to serve this request and respond accordingly. In order to do that we need a voting algorithm for electing the Master Host. The Master Host is a regular Host with the responsibility of serving Messengers' connection requests and balancing the load between other Hosts as explained in Section 4.4.

Elections are triggered by any of the Hosts if no Master is detected in the group of Hosts and all available Hosts are participating. The Election process consists of several steps which contain message exchanging in the Global Multicast group like *participate in election requests*, *status updates*, *votes*, and *master announcements*. All afore-mentioned messages are described in Chapter 5. The Elections are restarted in the cases where no one announces himself as a Master or when the most voted Host does not have the majority of the votes.

4.2 Dynamic host discovery

Dynamic host discovery is a process in which the already existing Hosts can meet and exchange information with a Host that has just entered the group of Hosts. A newly added Host sends his current *status* upon connecting and requests *status* messages from other Hosts. Hosts exchange messages between them through the Global Multicast channel. Detecting a dead/crashed Host is presented in Section 4.5.

4.3 Message passing

As mentioned in Chapter 3 the MultiCom system uses three different channels (Unicast & Local/Global Multicast) for sending several types of messages. Messages are marshalled (Serialized) before sending and unmarshalled upon receiving.

- The Unicast channel is used for exchanging *chat* and *acknowledgment* messages between a Messenger and its Host. *Chat* messages can be sent by a Messenger and be received by a Host. On the other hand, *acknowledgments* can be sent or received by both the Host and the Messenger. A Messenger creates and sends *acknowledgment* messages only for *chat* messages received in his Local Multicast group.

- The Local Multicast channel is used by a Host for sending *chat* messages to the Messengers connected in his group. The text contained in these messages is going to be presented in the Messengers screen. This channel is also used by the Host to re-send targeted *chat* messages to those Messengers of his group from whom he hasn't received an acknowledgment yet.
- Finally, the Global Multicast is used for exchanging *command* messages between Hosts. These messages should be parsed and Hosts should perform the necessary operation. These messages contain commands *like request status update*, *votes*, *forward chat message*, etc.

Figure 2 gives an overview of the traffic in the system when one *chat* message is sent. Messenger A sends a *chat* message in <1>. This message is sent to the Local Multicast group by Host A in <3> and forwarded to other Hosts in <8>, which also forward the message to their Local Multicast group in <14>. From the Figure it can be seen that some messages are ignored, <9> and <12> and that acknowledgments are sent by the entities receiving the *chat* message. An example of an acknowledgment is <2>, which is an acknowledgment from Host A that it received the *chat* message from Messenger A.

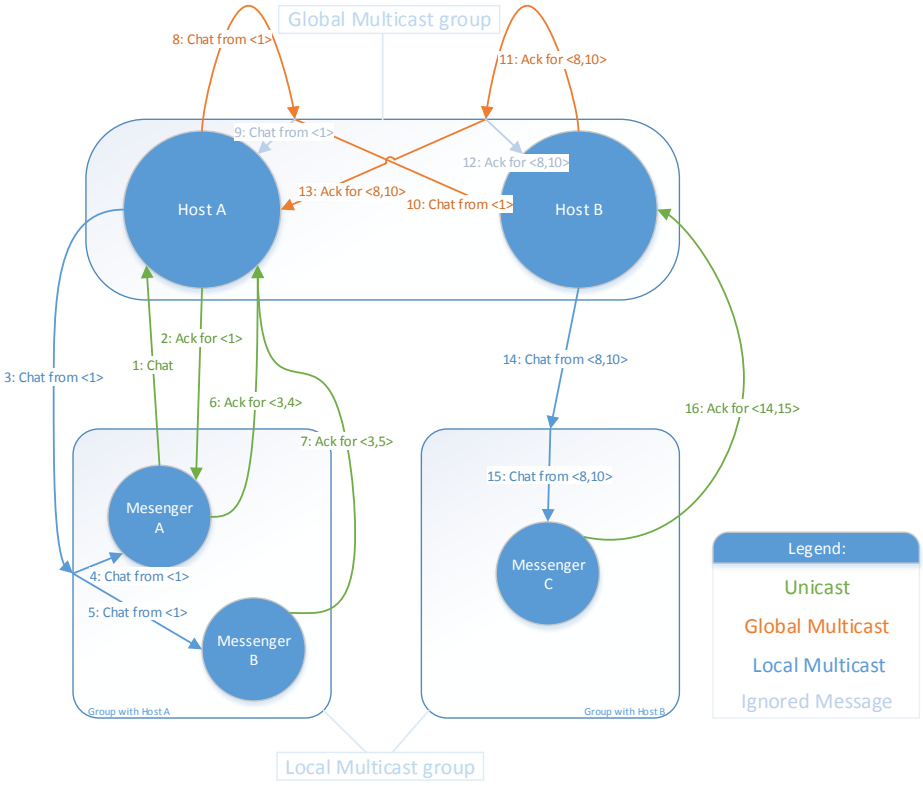


Figure 2: Traffic for sending one message

4.3.1 Reliable delivery

In order to ensure the delivery of messages between Messengers belonging in different groups we implemented three levels of reliability, one for each communication channel. This was necessary in order to cover all paths that a *chat* message takes in the system.

Initially, a chat message is created by a Messenger and sent to its Host though the Unicast communication channel. Upon receiving this chat message, the Host creates and sends back an *acknowledgment* that contains the information of the original *chat* message. The original *chat* message is stored in a queue and is resent in the case that no *acknowledgment* is received by the Messenger.

In the second step, the *chat* message is sent to the Local Multicast group by a Host which keeps it in a queue until he receives an *acknowledgment* from each Messenger in the group.

The third and final level of reliability is implemented on the Hosts side while exchanging *chat* messages that should be forwarded by other Hosts. These messages are traveling through the Global Multicast channel and are hold in a queue until an *acknowledgment* is received.

4.3.2 Ordered delivery

Just like in reliability, ordering should be implemented in three different levels according to the processes that receive and store *chat* messages. Messengers store messages that they receive from their Host through the Local Multicast channel in a queue ordered by their id. These messages are not presented directly in the Messengers screen but are kept for an amount of time in order to ensure that if any other delayed *chat* messages with a lower id arrive, they will be stored and presented in the correct order. This queue is a holdback queue and its functions are described in Chapter 5

Hosts store *chat* messages received from their Messengers through the Unicast channel in a holdback queue before they sent them to their Messengers through the Local Multicast channel.

Finally, Hosts store the received *chat* messages from other Hosts through the Global Multicast channel also in a holdback queue before forwarding them to other Hosts.

4.4 Load balancing

The system can dynamically change its size when new Hosts and Messengers are joining or leaving the groups. In order to keep the system balanced we need a mechanism that can identify an unbalanced load condition. A balanced load is defined as having the same number of Messengers, on average, connected to each Host. This condition is defined as:

$$toReroute = maxNrOfMessengersInHost - \frac{\#Messengers}{\#Hosts}, \text{ with } \#Hosts > 0$$

where $toReroute$ is the number of Messengers that need to be rerouted in order to achieve a balance in the system, $maxNrOfMessengersInHost$ is the number of Messengers of the Host that is serving the largest number of Messengers, $\#Messengers$ and $\#Hosts$ are the total number of Messengers and Hosts respectively. This operation is executed by the Master Host and is responsible for redirecting Messengers to the Hosts with less load.

For better understanding of this operation we will give an example of an unbalanced load condition as shown in Figure 3. In the current snapshot of the system we have two Hosts and two Messengers but both Messengers are served by Host A. When the Master Host identifies this unbalanced load condition it decides that one of the Messengers should be redirected to the other Host. This operation uses the Global and Local Multicast channels for exchanging messages. More details about the implementation of this algorithm are going to be presented in Section 5.5.

The load balancing discussed here gives the system `mobility transparency` as no operations are affected by moving the Messengers. The load balancing also gives `performance transparency` as the system is reconfigured in a way as the loads vary to increase its performance.

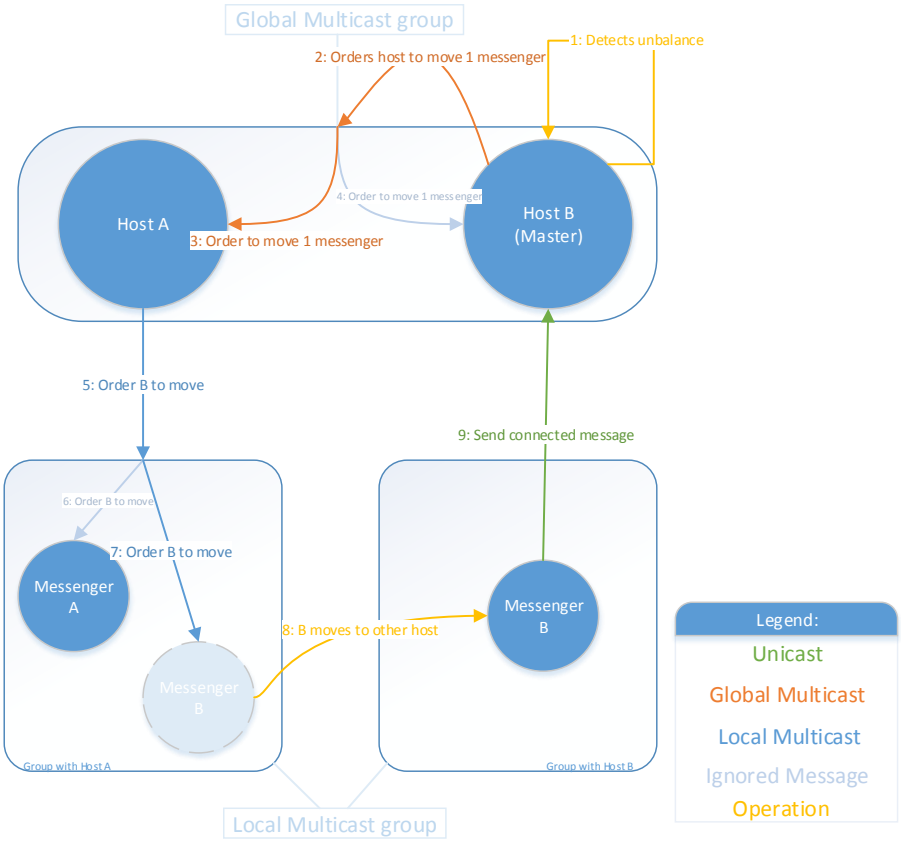


Figure 3: Handling an unbalanced load condition

4.5 Crash failure tolerance

There are three different crash failure types that can occur in the MultiCom system and each one of them is handled differently as presented below.

- **Host crash**.
  When a Host is crashed/dead he stops sending *status update* messages which leads other Hosts to remove him from their list in case that his last *status update* is older than a specified time limit. The Messengers can detect that the connection to their Host (Local Multicast) is broken and they send a connection request to the Global Multicast channel. The Master Host will respond to them with an available and suitable Host. Messengers are not allowed to send messages during the reconnection process.

- **Master Host crash**.
  Every Host is sending a *status update* message in a specified time interval as mentioned above. Also in a different time interval *(status update time interval)*2* each Host checks if any of his known Hosts or himself is the Master Host. In case that none of the Hosts is the Master we assume that the Master Hosts is dead and in that case the current Host can request Elections. As long as no Master is present in the Hosts group new connection requests from Messengers cannot be served.

- **Messenger crash**.
  Each Messenger sends a *heartbeat* message in a defined time interval in order to inform its Host that he is alive even if he is inactive. A Host performs a check in a different time interval *(heartbeat time interval)*2* to detect and remove any Messengers that have not sent a heartbeat for a time larger than a defined time limit.

Byzantine failures are impossible to be detected because MultiCom is an asynchronous system. To increase the integrity of the system, each *chat* message contains a checksum created by the sender. The receiver can verify if the received message is corrupt using this checksum. If it is corrupt no acknowledgment is sent, leading to receiving the message a second time upon the retry of the sender.

Because of the fault tolerance described in this Section, MultiCom has `failure transparency`. This means that the system allows users to continue on working without affecting their operations.

4.5.1 Synchronization and Uniqueness

As mentioned above, the MultiCom system is an asynchronous communication system where data is not transmitted at regular intervals. At any given moment, multiple messages can be transmitted or received by any of the participants in the system. In order to identify duplicated messages and also to ensure the ordering of the received messages we assigned a unique identifier (`id`) to each message of an entity. That means that different entities can create messages with the same identifier, a situation that is not desired in a communication system. In order to avoid this situation we distinguish messages by their sender's `processId` and messages from the same entity by their incremented `id`. Whenever a message is forwarded, the message gets a new `id` and `processId` of the new sender.

5 Solution Details

This Chapter gives details on the application of the concepts covered in Chapter 4 by giving specific algorithms or by explaining the process.

5.1 Leader elections (Voting Algorithm)

The leader election algorithm is designed by ourselves, but loosely based on the Bully algorithm and its purpose is defined in Section 4.1. The idea is that the Host which should be elected as the Master, must be the one which has the least amount of work to do. The least amount of work is defined as serving the least number of Messengers. Thus, during the election Hosts are voting on the Host with the least number of Messengers. There must also be a tie breaker for cases in which this number is the same for multiple Hosts. Therefore, when this number is equal, the Hosts will vote for the Host with the lowest port number. When this port number is also the same then the Hosts will vote for the Host with the lowest address (string comparison). Only when a Host has the majority of votes, it will announce itself as the new Master. Otherwise new elections are started. When no Master announces itself, new elections will be started as well.

Before explaining the algorithm, the state machine used for the election process is described in the following Enumeration:

- normal: When a Host is not participating it is in the *normal* state.
- voting: As soon as a Host is starting to participate in an election it goes in to the *voting* state.
- voted: The Host advances to the *voted* state when it has voted on a certain host.

The algorithm is defined as follows:

1. A Host wanting to start an election sends a command on the Global Multicast and goes in to the *voting* state, just as each Host receiving this command.
2. Each Host sends each other status updates.
3. After a certain delay no Host can enter the election anymore. This is realized by saving the start of an election locally and only consider Hosts that sent their status update after the start of the election[5].
4. Each Host votes on the Host serving the least number of Messengers. If this number is the same the lowest port number is chosen and if the port numbers are also identical the Host voted on is the one with the lowest address.
   (a) If there is only one Host it will select itself as the Master.
5. The vote is sent with Global Multicast and each Host keeps track of the number of votes on each Host.
6. After voting, the election advances to the *voted* state.
7. The Hosts wait for a certain amount of time to collect all the votes.
8. Each Host checks if it has received a majority of votes from the participants of the election. If a Host has a majority it announces itself as a leader. Otherwise new elections are started.

[5]Every time a status update is received the time of receiving is stored. This is also used to detect Hosts that died.

Next to the steps above, the algorithm is designed to behave correctly in the degenerate situation where an election is started during elections. What happens is that the running election is interrupted and all the messages of old elections are ignored. In order to do this, the starter of an election sends his time with the start command. The time of the starter is saved locally by each Host together with its process ID. Every message part of the election is sent with this time and process ID. For any other start of an election received after this, the starter time and process ID is overwritten. Then, for any message part of an election, a Host can check if this message is part of the last election. So each Host uses a clock from another Host, but never to compare it or use it in any way related to its own clock.

5.2 Dynamic host discovery

Every Host must be aware of the other Hosts in the network. This can be realized by dynamic host discovery and its use is specified in Section 4.2. In MultiCom each host maintains a list of all the other Hosts in the network. When they receive a status update from another Host they update the information on that Host in their Host list. If the information received belongs to a Host not yet known, then a new Host is added to the list of Hosts.

5.3 Reliable delivery

Reliable delivery, as mentioned in Section 4.3.1, is implemented in three levels in MultiCom. For each level the basic idea is the same. An entity A sends a message to entity B and puts this message in a queue for possible retries. When storing each message, the time of when the message was sent is stored in the message. If an acknowledgment is received from entity B, then the message is removed from the retry queue as it does not have to be sent again. A Thread monitors the retry queue and checks if there are messages older than 2 seconds in the queue. If there are, it means that no acknowledgment was received in time and that these messages must be resent. Before resending a message however, its time is set to the current time and the number of times the message has been resent is incremented. A message is removed from the retry queue if it has been resent three times in total. This means that after retrying the message 3 times it gives up. It is decided that resending the message more than three times would decrease performance and that if the message is not received in four times it is assumed that it will never be received. This is assumed, because if a message is not received in four times, then either the channel is not working or the receiver failed. This is the basic idea of reliable delivery. The differences between the three channels are given as follows:

- **Messenger sends a *chat* message to its Host**
  Whenever a Messengers sends a *chat* message it will put the message in the queue for retrying. When no acknowledgment is received it will resent the message to the Host. This is not different from the basic idea.
- **Host sends a *chat* message to the Local Multicast group**
  Whenever a Host sends a *chat* message to its Messengers it will store the message in a list for retrying. This message is stored in a special object called *ForwardMessage* which, for each message, also stores a list of Messengers that should receive the message. For each Messenger the number of retries is stored as well. Whenever a Messenger sends an acknowledgment, the Host removes the Messenger from this list in *ForwardMessage*. If this list is empty it means that every Messenger has sent an acknowledgment and the *ForwardMessage* can be removed, because it does not have to be resent anymore. There is also a monitor which checks whether Messengers have sent an acknowledgment. If not then the message is resent incrementing the number of retries. If the message is resent more than three times the Messenger will be removed from the list.
- **Host sends a *chat* message to the Global Multicast group**
  The process of forwarding a *chat* message to other Hosts is the same as when it needs to be sent to

the Local Multicast Group. However instead a list of Messengers, a list of Hosts is maintained in the *ForwardMessage* object.

5.4 Ordered delivery

As explained in Section 4.3.2 a holdback queue can ensure the ordering of messages. In MultiCom the holdback queue is a priority queue sorted on the id of a message from the sender. The three levels of ordering have the same structure as that of reliable delivery. If a Messenger sends a *chat* message to its Host it has one priority queue in which to store messages to hold back. A Host which forwards a *chat* message to Messengers or other Hosts respectively maintain a holdback queue for each Messenger and Host.

Whenever a message is received it is put in the correct holdback queue together with the time of receiving. A monitor keeps track of the messages in the holdback queues by looking at the first message in the queue only. If this first message (head of the queue) is older than 300 milliseconds it is removed from the queue and stored in the delivery queues. When the head of the holdback queue is not yet ready to be sent (<300 ms) no operation is done on the queue. This allows for reordering operations (using message identifiers) if messages are sent in the wrong order but withing 300 milliseconds of each other. When, for example, a message of id 2 is received at time T=0 ms and a message with id 1 is received at time T=50 ms, they will be reordered in the holdback queue. The message with id 2 is only sent when the message with id 1 is sent first.

Because the delay is 300 ms it might happen that messages are sent in the wrong order when they are sent 1000 ms after each other for example. Then, the messages will not be reordered. This design, however, is chosen on purpose. First of all the delay of 300 ms increases the time between sending and receiving a message and increasing the delay would only increase the difference in time between sending and receiving a message. Secondly, the MultiCom system does not ensure that messages are received exactly once. A message is resent three times at a maximum. The holdback queue cannot be made to block when a message is missing, because no messages will then be delivered anymore. This avoids having a single point of failure in the system.

5.5 Load Balancing

To implement load balancing, a Master Host monitors the balance in the system by looking at the number of Messengers connected to each Host. The formula for this condition is mentioned in Section 4.4. To explain the process of moving Messengers, to reduce the load of one host, the situation of Figure 3 is considered.

The Master, Host B, detects in <1> that there is an unbalanced load (Host A has two Messengers and Host B zero). It calculates the Host from which Messengers need to be moved and the Host to which Messengers need to be moved and stores this information in a command message. In <2> and <3> it sends a Global Multicast with this command targeted at Host A to order it to move 1 Messenger. Host A receives the order and selects a Messenger to be moved. In <5> and <7> the Messenger B is ordered to select another Host. This other Host being Host B. The Messenger knows which Host to move to as it received the information in the message <7>. In <8> and <9>, Messenger B connects to Host B. Host A has removed Messenger B from its Messenger list and Host A registers Messenger B to its Messenger list.

5.6 The lifecycle of a Messenger's connection

In this Section we will give a thorough description of all the processes executed from the moment that a Messenger is initialized until the moment the he decides to quit the system. In order to make that clear we will make references to Figure 4 which present the initial connection of a Messenger.

1. Messenger A uses the Global Multicast channel to send a *connection request* in <1> message which will be received by all Hosts but it will be parsed in <2> and answered by the Master Host. When no response is received after some time, the Messenger resends the connection request up to a total of four retries.
2. The Master Host looks in his list of Hosts and finds the most suitable Host which is the one with the least number of connected Messengers. The Master Host sends a *connect information* message to the Messenger through the Global Multicast channel in <7>.
3. When the Messenger receives the *connect information* message that the Master Host sent at the previous step he uses the information in order to join the Local Multicast group of Host B and makes a Unicast connection with this Host.
4. Upon joining the new group, the Messenger sends a *send connected* message to his Host in <8>.
5. When the initial connection is finished the Messenger leaves the Global Multicast group since there is no reason for him to listen to all these messages.

When a Messenger wants to stop his process he leaves the Local Multicast and closes all open sockets. In case that his process is unexpectedly ended his Host detects his death by the heartbeat mechanism which was described in Section 4.5. This process, with a slight difference in the first steps, is executed when a crash of the Host is detected and a reconnection is required as presented in Section 4.5

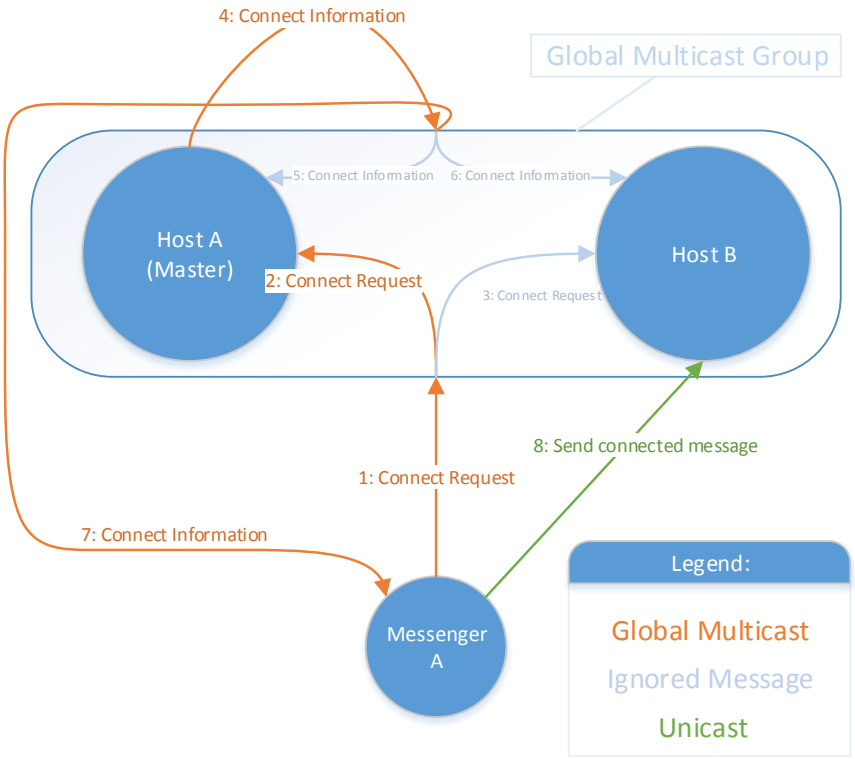


Figure 4: Initial connection message exchange

6 Results

In the beginning of this report (Chapter 1) the requirements and goals of MultiCom are stated. For each requirement and goal a short summary of how this requirement and goal is accomplished is given.

The first requirement named, is to implement a voting algorithm. The voting algorithm is specifically designed for MultiCom and its purpose is defined in Section 4.1, following by the algorithm in Section 5.1.

Implementing a dynamic discovery of Hosts is the second requirement. Section 4.2 describes how dynamic discovery of Hosts is used within MultiCom. More detail on the application of this requirement in MultiCom is given in Section 5.2.

To be able to send messages reliably and deliver them in the correct order, MultiCom uses reliable ordered multicast. Reliability and ordering are implemented independently from each other and are both implemented on three different layers. Sections 4.3, 5.3 and 4.3.2 explain how reliability and ordering is used and implemented in MultiCom.

As users want to continue their operations even if there is a crash failure, tolerancy for crash faults is implemented. The types of crash failures and their effects are outlined in Section 4.5.

As an extra requirement load balancing is implemented. Section 4.4 gives arguments why load balancing is needed and Section 5.5 gives implementations specifics.

Besides these requirements, MultiCom aims at providing reliability, transparency, performance and integrity. Reliability is ensured with the reliable multicast requirement. MultiCom is transparent in the way that it moves Messengers without the user's knowledge and that several failures of the system are concealed. Performance is increased because of the balancing of the load of Hosts. Finally, MultiCom's integrity is ensured by ordered multicast and checksums on *chat* messages.

In conclusion, MultiCom meets the requirements and goals and is a decentralized distributed system which the users perceive as a black box.

Appendices

In this Chapter we present a conversation of three friends using the Messenger application of the MultiCom system. In Figure 5 we can see that in the current snapshot, the system consists of five entities. Two Hosts listening, one with Local Multicast group set on port 58192 and the other on port 49519. Also, three different Messengers are exchanging messages with two of them connected on the Host's 49519 Local Mulsticast group and one on Host's 58192 Local Multicast group. Each username has a different color which is presented on the users main screen.

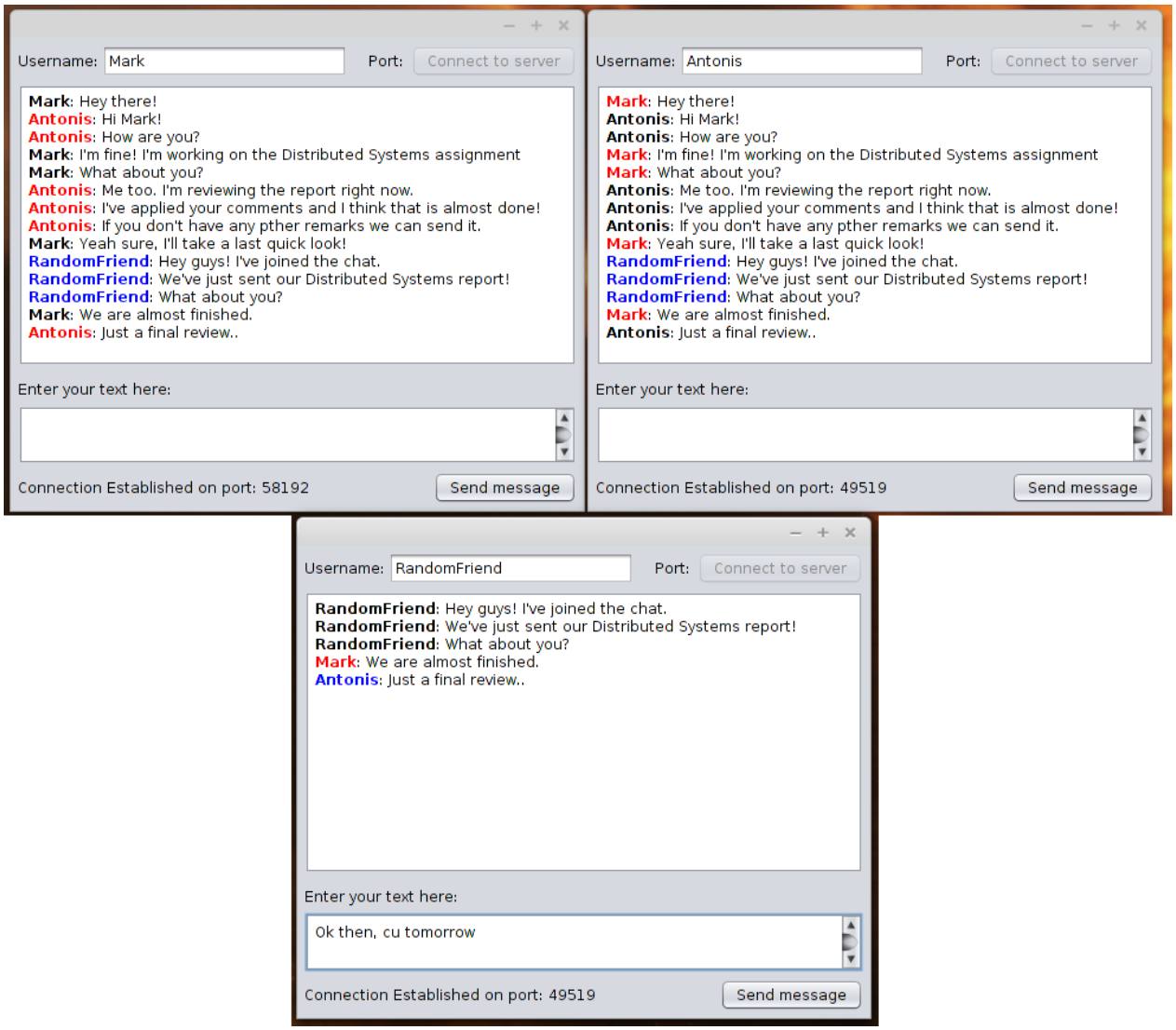


Figure 5: Conversation between three users of the MultiCom system