Project Final Report: Simon Multicasts

Introduction

Simon Multicasts is a massive multiplayer online game that uses central and advanced principles of distributed systems development.

It is based on the game idea of "Simon Says" with the small difference that what "Simon Says" is not to be executed in movements by the participating players; in our case, "Simon" distributes a string of characters to the players per turn.

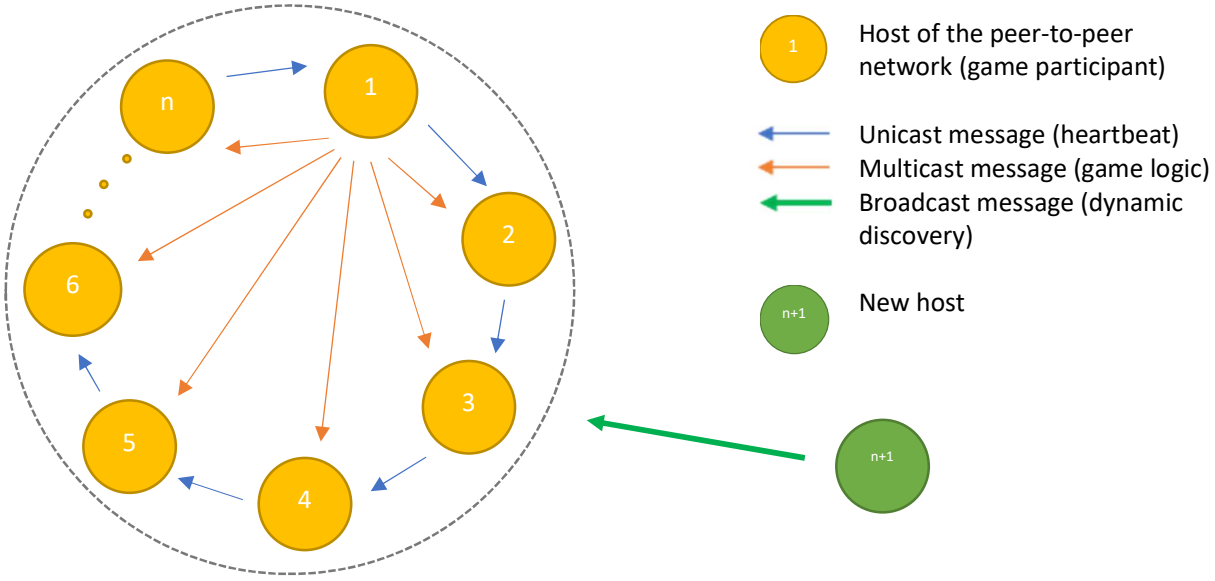
They have to play back this string via their input device and then send back a message with this input for evaluation. The first player to send back the correct answer string gets 10 points, the second player gets 5 points, and so on. A new "Simon" is chosen after each round.

Requirements analysis

- **Dynamic discovery:**
  - New host can join the game by broadcasting a message asking to join, if there is a game leader, he will update his player List with the new player and respond with the updated player List, so that the new host can participate in the game
- **Crash-Fault-Tolerance:**
  - Adjacent hosts keep track of their respective neighbor's state through heartbeats and update the player List when a host isn't reachable
  - Every host keeps a copy of the game's score board, so that the game can be continued, even when the leader crashes, only the current game round will get lost
- **Voting:**
  - After every game round or if the current master crashes a new voting gets started and a leader gets elected
- **Ordered reliable multicast:**
  - We do not have a central server, that stores the truth of the game status, so we need a way to ensure that every peer has the exact same game status
  - If each peer receives the messages in the same order and we have a definitive algorithm that derives the game state from the messages, we can ensure that everyone has the same game state stored locally.
- **Byzantine Fault Tolerance:**
  - To achieve Byzantine fault tolerance, we not only ping a node's neighbors with the heartbeat, but also query their current state to verify that the neighbor is functioning properly
  - If two neighbors cannot agree on a particular state match, a global consensus multicast mechanism routine is triggered to find an agreement and solution to the detected Byzantine fault

Jonathan Maier
Dustin Niklas Walker
Tilman Welsch

Architecture design



For the architecture design we decided to use a P2P network, so we don't need to provide a server instance. All important game information and the coordination code are distributed among the game participants.

To form the P2P network, each node has a UUID and holds a list of all the UUIDs of the game participants. In this way, each node knows all currently active nodes and can communicate with any other peer.

This not only provides communication capabilities, but also helps with crash fault tolerance. Because even if the leader crashes, all other nodes can replace the leader. The only information that is lost is the information of the current game round, which must be restarted.

To meet the requirement of the game and the project, we also must build a network ring. We're doing this by declaring the next lowest UUID as a node's neighbor. The node with the smallest UUID has the node with the highest UUID as a neighbor.

This ring is the basis for the heartbeat function and the voting algorithm, the heartbeat function checks whether its neighbor is reachable, and the voting algorithm send its voting messages to its neighbor.

New Peers can join the network by sending out a broadcast join message to all the hosts in the network on a specific port. The leader will always listen to join messages and is responsible for sending all the information a new Peer needs to participate in the game.

When the new Peer gets no answer to its broadcast join message, it declares itself as leader and waits for other Peers join messages.

Implementation

Remote Procedure Call (RPC) Broadcast/Multicast, Message based communication

All Network Communication is implemented in middleware.py

The class UDPUnicastHandler, TCPUnicastHandler and BroadcastHandler are responsible for the atomic parts in their field. The functionality of these classes is accessible through the Middleware class.

The class Middleware provides an interface (functions) for Network Communication to the model (Game). The class Middleware provides the abstractionLayer of OrderedReliableMulticast on top of the TCPUnicastHandler functionality.

The GroupView is saved in the Middleware. The GroupView is a dictionary (ipAdresses), containing the (ipaddress, Port) for all active Players. IP Addresses are saved exclusively in this dictionary.

The middleware can:
broadcastToAll(`command:str, data:str=''`) udp Broadcast
sendMessageTo(`uuid:str, command:str, data:str=''`) udp unicast
sendTcpMessageTo(`uuid:str, command:str, data:str=''`) tcp unicast
sendTcpRequestTo(`uuid:str, command:str, data:str=''`) tcp unicast; returns response
multicastReliable(`command:str, data:str=''`) tcp unicast to everyone in GroupView
multicastOrderedReliable(`command:str, data:str=''`) tcp unicast to everyone in GroupView (with Total Ordering)

The Observer-Pattern is used to act on incoming messages. You can subscribe to incoming udp unicast, udp broadcast, tcp unicast and ordered Messages ready for delivery, by calling something like subscribeBroadcastListener`(observer_func)`.

For all messages there is a command and data. The observer_func the command to decide if it shall act on a received message. Valid commands are: `'enterLobby', 'PlayerList', 'voting', 'hbresponse', 'hbping', 'lostplayer', 'updateIpAdresses', 'leaderElected', 'requestSequenceNumberForMessage', 'OrderedMulticast with agreed SeqNum'`

Command and data are combined into one string, seperated by a seperator symbol. After the Transfer over the Network the Arguments are split up with the .split() function.

Banned Symbols are: `_ , # $` as they are used to split() messages.

This is how the Remote Procedure Call is implemented. We are also able to send an array in the data string.

MY_UUID is always sent with every message over the network.

BroadcastHandler

There is one socket for sending and one socket for listening. The listening socket is in a separate Thread in a while loop, always listening for broadcasts. The Broadcast IP is dynamically calculated, based on the users IP address and subnet mask. The Broadcast Port is chosen by the user. (This allows multiple Game Rooms)

UDP Unicast

There is one socket for sending and listening. A Random available Port is selected. The socket is in a separate Thread in a while loop, always listening for udp unicasts.

TCP Unicast

There is one socket for listening to incoming tcp unicasts. This socket is bind() to the same Port as the UDPUnicastHandler. (UDP Socket and TCP Socket can be bind to the same Port simultaneously). With this implementation we only need to store one (ipAdress, Port) for each peer in the GroupView.

The listen socket is waiting (socket.accept()) in a separate Thread for connection Requests (socket.connect()) of other peers. Upon connection a new Thread is started on the receiver-Side to Handle the request.

To send a tcpUnicast a new Thread is started. A new socket with a random Port is created. This socket tries to connect to the given address and then send the bytes. After that it closes itself again

There is also a function that is able to send a (TCP) request. It sends a request, waits for the requested data and then returns this data to the caller of the sendTcpRequest() function. This function can be used to request a sequence Number for the ISIS-Algorithm

Dynamic Discovery of Hosts

We need Dynamic Discovery of Hosts to provide the ability that other hosts can join a game room without knowledge about any specific existing group participant, the only information that a host needs to join a game is the broadcast port of all the current game rooms.

The broadcast IP address gets calculated from the own IP address and the subnet mask, which both gets provided by the DHCP server in the network or manual user input. This allows the game to be played in different Local Area networks, without changing anything of the source code. If a host wants to join a game, it only has to be in the same subnet as the game room he wants to join.

If the host is on the same Local Area Network, but in a different Subnet, the calculation of the broadcast IP address will only be valid for the hosts subnet and it won’t be able to find hosts in another subnet.

Every game room has its own unique broadcast port, on which a new host, that wants to join a game, has to send a broadcast join message. To provide the ability to join games without user knowledge about the broadcast port of a current game we implemented a default broadcast port, which will be used if the user does not specify a port.

Using this implementation, it is possible to join games, using the default port, without user knowledge of the broadcast port, while still providing the ability to run multiple games on the same subnet using different broadcast ports. Hosts that want to join a game, that is not running on the default port, have to figure out the broadcast port for these custom games by themself.

The procedure to join a game looks like this:

- New host sends a broadcast message to the calculated broadcast message asking to join the game
- The message gets send to all Hosts in the subnet
- The Hosts are always listening for new join messages
- When they receive a new join message, they update their UUID – IP dictionary, which contains a UUID to IP resolution, and also add the new Host to the player list
- After adding the newly connected node to the UUID – IP dictionary and to the player list, the nodes send the dictionary and the list to the new Host
- When the new Host receives the dictionary and list it has all the important information to participate in the next round of the game

The Dynamic Discovery of Host enables new hosts to easily connect to current games on the default broadcast port.

If a Host sends a join message, but does not receive a join command response, it assumes that currently there is no game room on the specified broadcast port. Making that assumption it starts a new game room and waits for other hosts to send join messages to the same broadcast port.

Crash Fault Tolerance

In order to keep the distributed system online even if one of its hosts in the peer-to-peer-network crashes or goes offline, an algorithm needs to be implemented, that detects and handles such an event adequately.

Description

The implemented algorithm consists of three parts: formation of a ring, detection of crashes, and distribution of information to other peers.

The ring is formed such that each host finds its respective neighbor by iterating over the aforementioned UUID-IP-dictionary (“GroupView”). The host with the next smallest host-UUID is then determined as the neighbor. The host with smallest UUID neighbors the largest. This process effectively forms a ring.

Each host in the ring is responsible to frequently survey its neighbor’s state (*“alive”*/*”dead”*) while the system is running. This is done by sending a UDP-unicast-message (*“hbping”*) to the neighbor once every other second requesting a specific response message (*“hbresponse”*), also through UDP-unicast. If said response is received within, the neighbor is deemed *“alive”* and no further actions are taken.

However, if there is no response following four consecutive *“hbping”*-messages the neighbor is assumed to be offline. Now, this information needs to be distributed to the other peers in the network. To do so reliably, every other host in the system is sent a TCP-message containing the UUID

of the crashed host. This allows them to update their UUID-IP-dictionary ("GroupView") accordingly. Additionally, voting is initiated if the crashed host was the leader.

Whenever necessary, e. g. in case of a host crashing or a new host joining, the ring formation is updated with affected hosts in the system identifying new neighbors for themselves.

Implementation Details

Messages that are used to survey a neighbor's state ("hbping" and "hbresponse") are sent using UDP unicasts. In comparison to TCP unicast messages, this allows for a reduction of message-related overhead. However, to reduce the risk of missed response messages leading to a false judgement, the counter allows for a maximum of three consecutive *"hbping"*-messages to be unanswered before declaring the neighbor offline.

Voting

Why do we use LCR?

For the Game Simon Multicasts, we considered two voting algorithms. The Bully Algorithm and the LeLann-Chang-Roberts (LCR) Algorithm.

The Bully Algorithm uses broadcast messages to send its election messages to all other processes with higher IDs. On the other hand, the LeLann-Chang-Roberts-Algorithm only uses Unicast messages for voting messages, but it requires a reliable ring structure of all the Nodes that take place in the election.

Because the LCR only requires unicast messages, where the Bully Algorithm needs broadcast messages, we choose to use the LCR Algorithm. A lot of broadcast messages could lead to network overload, which results in package loss, what we need to avoid.

Because we always have a consistent ring view of all the nodes, this gets ensured by the heartbeat constantly checking for the availability of its neighbor, we already meet the requirements of the LCR Algorithm.

How does LCR work?

LCR works by sending voting messages in the network ring. The voting message gets passed through the ring until a new leader gets elected. We achieve this by following the following instructions:

- One node starts the election with its own identifier
- Every node in the ring receives the message and checks weather the identifier is greater, smaller, or same than the own identifier:
    - If the received identifier is greater then the nodes identifier, the node just passes the message to its next neighbor
    - If the received identifier is smaller then the nodes identifier, the node exchanges the identifier in the message with its own identifier and sends to its neighbor
    - If the received identifier is the same as the node's identifier, the message passed the whole ring and the node got elected as new leader. The node then broadcasts this to all other nodes

How do we implement LCR?

We implement the LCR by using every nodes UUID as identifier and sending it to its ring neighbor. We get the neighbor by sorting all nodes dictionary and then picking the node with the next lower index.

The functionality is implemented in the middleware and consists out of two functions:

- initiateVoting(self) is used to start a new voting, it sends a Unicast TCP message to its neighbor, container the command 'voting' and its own UUID. The function gets called by a nodes heartbeat function when it can't reach the leader, or by the leader itself after successfully ending a game round
- _checkForVotingAnnouncement(self, messengerUUID:str, command:str, data:str) is used to handle incoming voting Announcements, it subscribes to all incoming Unicast messages and checks for the following conditions:
    - If the command is 'voting' it either declares itself as leader, sends the message to its neighbor, or replaces the message UUID with its own UUID and then sends it to its neighbor. (based on, weather the incoming UUID is equal, greater, or smaller than the nodes own UUID). When a node declares himself as new leader, he sends a reliable multicast message to all nodes, containing the 'leaderElected' command and its own UUID and sets the leaderUUID variable as its own UUID and switches it game state to 'simon_startNewRound'
    - If the command is 'leaderElected' the node replaces its leaderUUID value with the incoming UUID of the message and it also changes its game state to 'state_player_waitGameStart_f'

To ensure that we do not have any data loss during the voting process, we only use TCP connections (also for the 'leaderElected' multicast → Reliable Multicast). This way we ensure a reliable voting process.

How do we prove that the voting is correct?

We proved that the voting is correct by performing a simple voting test. We implemented a user input that manually triggers the voting function so that we can test it atomically.

We tested the voting functionality with 5 processes running on two different devices. We also tested the whole scenario 3 several times, including restarting all the processes.

All three times the player with the highest UUID got elected as new leader, which proofs that the implementation of the Voting Algorithm works correct.

We also tested to trigger the voting function through the heartbeat function, by killing the current leader process. We tested this scenario only one single time, with the same setup, and can confirm that the voting algorithm worked correct; one new leader got elected and was accepted by all other nodes.

Ordered Reliable Multicast

What kind of Reliable Multicast do we need?

- We need want to sure, that every peer always has the same game state
- In our implementation of the game only the first player receives the points
- Therefore, it is very Important that everyone has the Messages in the same order
- To achieve this, we need Total Ordering
- This gets implemented by the ISIS Algorithm

How does the ISIS Algorithm work?

- To align with the ISIS Algorithm the P2P network must handle multicast messages in the following way:
- The multicast sender sends the message to all processes.
- Recipients add the received message to a priority queue, tag the message undeliverable, and reply to the sender with a proposed priority (i.e., proposed sequence number). Further, this proposed priority is 1 more than the latest sequence number heard so far at the recipient, suffixed with the recipient's process ID. The priority queue is always sorted by priority.
- The sender collects all responses from the recipients, calculates their maximum, and re-multicasts original message with this as the final priority for the message.
- On receiving this information, recipients mark the message as deliverable, reorder the priority queue, and deliver the set of lowest priority messages that are marked as deliverable.

How do we implement this?

Reliability:

We build a Unicast to multiple receivers through our middleware, this Unicast implements a TCP connection, so that we know if a message was transmitted successfully, this way we don’t have to use Piggyback or Negative acknowledgements

Ordering:

To achieve total ordering, we must implement a priority que, which stores all incoming messages, that are marked as reliable multicast messages.
- This que is a list of Messages. A Message is implemented as a class (dataclass).
- A Message object contains the Attributes: `messageSeqNum: int, messageCommand: str, messageData: str, messageID:str, messengerUUID:str, deliverable:bool`
- This list can be sorted by the messageSeqNum.

The Implementation for this is in middleware.multicastOrderedReliable(command:str, message:str): First the message gets a UUID. Then the sender calculates a own ProposedSeqNum, which is `max(highestbySelfProposedSeqNumber, highestAgreedSequenceNumber) +1`
`This Message gets queued in the holdbackQueue and marked as undeliverable`

Now the Sender needs to request a Sequence Number from every peer (in the GroupView). In a for loop every peer is sent a request (with the sendTcpRequestTo() function). The Problem here is that a tcp request is blocking, because sendTcpRequestTo() function needs to wait for the response. The solution is that for every request in the for loop a new Thread is generated. This thread calls the sendTcpRequestTo() function. The Threads get added to a list, so that they can later be joined again, and their return value collected.

Out of the returned SeqNumber suggestions the highest one is picked. Now a reliable Multicast can be sent to all peers. Everyone updates the message in their holbackqueue with the agreed Sequence Number. The Message gets set to Deliverable. Upon the append a checkForDeliverables() is triggered automatically. This function tests if the Message on top of the Queue (lowest seqNumber) is deliverable.

All processes need to update `highestbySelfProposedSeqNumber` and `highestAgreedSequenceNumber at their respective events.`

https://cse.buffalo.edu/~stevko/courses/cse486/spring19/lectures/11-multicast1.pdf

https://cse.buffalo.edu/~stevko/courses/cse486/spring19/lectures/12-multicast2.pdf

Byzantine Fault Tolerance

- The heartbeat is extended to not only ping a node's availability, but also query their current state (using multicast), to verify the neighbor is working properly
- If two neighbors do not agree on a specific game of state, then a global consensus mechanism routine is triggered to find an agreement and resolution for the detected byzantine fault
- To reach consensus the sum of the faulty nodes must be smaller than a third of the sum of all nodes

Discussion and conclusion

We have multiple points that can be discussed for the Project:

First, the Voting Algorithm uses the UUIDs, that are also used to build the ring. This has no influence on the reliability or performance but could be implemented better using different identifiers for the voting.

Second, the join message to join a game could also get send to the broadcast IP address: 255.255.255.255. This would result in the ability to join all games on the same Local Area Network

instead of only the same subnet. The reason for this behavior is, that we currently send the join message to the, by IP address and subnet mask, calculated broadcast IP address. The router will send this message only to the hosts on the same subnet. If we would use 255.255.255.255 the router would convert the IP broadcast to an ethernet broadcast and the message would reach all hosts on the same ethernet network, even if they are in a different subnet.

Further, the implemented crash fault tolerance aims to reduce message overhead through the usage of UDP-unicast messages for frequently occurring messages. However, it must be noted, that to minimize the impacts of potential message losses using the UDP protocol, multiple request messages are being sent and the loss of up to three responses is tolerated. This, in turn, leads to a time-lag in the discovery of crashed hosts.

We can conclude the documentation by stating out, that we fulfilled the requirements for Dynamic discovery of Hosts, Crash-Fault-Tolerance, Voting and Ordered reliable multicast.

To provide Byzantine Fault Tolerance, we figured out an implementation to fit the requirements, but we were not able to implement it in the application code.

The team is aware that there are better and more lightweight implementations then the ones we choose, but all implementation choices were made to serve the game's needs.

Source and Demo-Video

https://github.com/Dustin-dusTir/distributed-systems-game