A peer to Peer Card Game

Distributed Systems

Authors

Petar Hariskov (S3226735)
p.i.hariskov@student.rug.nl

Fthi Arefayne Abadi (S3074641)
f.a.abadi@student.rug.nl

Submitted to: Marco Aiello
Contact: m.aiello@rug.nl
Date: November 3, 2017

1 Context and Background

This Chapter introduces and describes what a peer to peer network is and the logic of the card game. The chapter is split into three parts: the logic of the game, computer card games, and a peer to peer card game.

"A peer-to-peer (P2P) network in which interconnected nodes ("peers") share resources amongst each other without the use of a centralized administrative system". A card game is a popular computer game study which can be played with at least two players depending on the type of the card game. A computer card game can be played human against computer, computer against computer, or human against human on top of a system built to handle the competitive game logic.

A peer to peer card game implements a middleware to create a communication network for player, provide services for selecting a leader, voting on a value and reaching a consensus, and a fault tolerant system to enable the game logic to run reliably over the network. The users can use a user interface to play the game on top of the transparent peer to peer infrastructure.

1.1 Game logic

The card game of war is simple . Players are given equal amount of cards at the beginning of the game faced down. Each turn consists of players drawing a random card from their hand and placing it for all to see. The player with the highest card gets all the cards on the table. In case of two or more players having the same cards they deal three more cards each. The player with the highest third card wins. In case a player does not have 3 remaining cards to play, he has to take cards from the opposite player, if this is impossible the player loses. This continues until a player emerges with a card higher than the other playing players. The game ends when only one player has cards left.

1.2 Computer card games

A real world card game involves players with cards at their hand, dealt by the game leader. Card games in computer science has been developed as a centralized and peer to peer system. In a centralized card game a client-server architecture, where the web clients are connected and register in the central authoritative server as a players.

As shown in figure 1 , all the actions of all the players(clients) are controlled by the central server which also serves as the source of the cards, keeps track of player's turns and checks if the game is over. Every client needs to stay connected to the central server to be able to keep playing. A client constantly receives update of the game state from the central server and locally creates a representation of the game state.

[0] https://en.wikipedia.org/w/index.php?title=Peer-to-peer&oldid=806452593

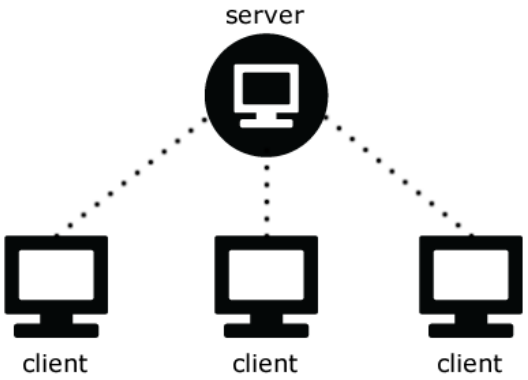

Figure 1: Centralized card game[1]

If a client performs an action, such as moving from one point to another, that information is sent to the server. The server validates whether the information is correct, then updates its game state. After that it propagates the updated game state to all clients, so they can update their game state accordingly.

The problem with the centralized poker game is that too much tasks and computations are done in one server which can get overloaded when too many players join the game. This creates a bottleneck in the system and all the clients need to trust that the server is fair, trustworthy and not faulty. As alternative, a peer to peer card game is introduced that works in a similar fashion to how card game is played in real life.

In a peer to peer card game, there is no central entity or server that controls and manipulates the game state. Instead every node(player) is involved in updating the unified game state. In a peer-to-peer (P2P) approach, a peer sends data to all other peers and receives data from them. In order for a peer to apply an action or change state, it needs a permission/acknowledgment from the rest of the peers or in some cases majority of the peers.

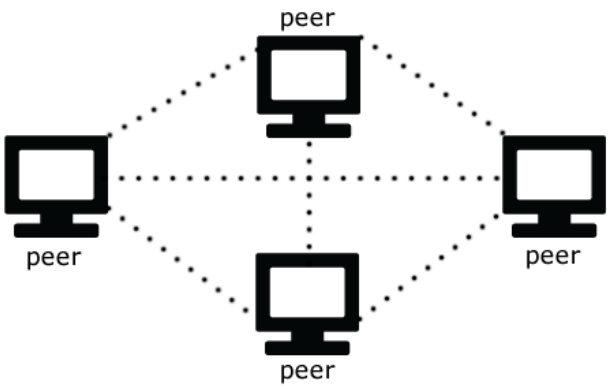

Figure 2: peer to peer card game[2]

[1] https://gamedevelopment.tutsplus.com/tutorials/building-a-peer-to-peer-multiplayer-networked-game--gamedev-10074

There is no single point of failure but rather the peer to peer system is tolerant to a fault whenever a peer crashes. The game state only changes upon approval by all the peers therefore becomes difficult to reach on agreement for a certain subject. As a consequence, additional communication coordination among peers, voting and selection strategies for a special(temporary) role need to be implemented on top of the game logic.

1.3 Peer to Peer card game

A player is a peer that is discovered by other peers that it has discovered. This implies a player is in a room with other players that want to play the same game. Once the peers discover each other the one peer is required to start the game by making the first move, but the player turns need to be maintained by one peer. Therefore, a leader(turn maintainer) selection is started by anyone of the peers after everyone has joined the group.

A leader selection procedure is necessary to temporally give one peer a special responsibility, in this case keeping players turn and telling who will play first and who will play next. Once a leader is selected all the peers can optionally ask the leader to get a copy of the players turn so the individual peers can easily check if it is their turn without asking the leader or a player can ask the leader for player turn approval only when the player wants to draw/throw a card.

Any action taken by a player needs to be communicated to the rest of the nodes to make sure that all the players have the same view of the action. For instance, a player who throws a card **X** needs to tell all the players what card is thrown. All the players need to agree on this value before executing this action and changing the game state. If there is a player that has a different view then a consensus needs to happen to agree on what the actual value is through a voting algorithm. The results of the vote are shared and every player will execute the action and change the game state. This continues until the game is over.

2 Problem statement

A peer to peer system needs a reliable network to communicate all the peers. There should be a configuration to enable them discover each other dynamically. Additionally, there needs to be an approach to handle new peers and peers that crash.

The card game has special roles that only one random player needs to execute. Therefore the peer to peer system should be able to select a player for the special role and substitute this player with another if it crashes.

During the game, a player must always inform the other player on the local actions that it wants to be executed. This action is globally executed if all(majority) of the peers agree on it. Therefore, a coordinated communication must be implemented among the peers, so they can agree or reach a consensus of what should be done. In case the peers do not agree on a value the action should be discarded and all the peers should revert to the previous game state.

To successfully implement a distributed multiplayer card game in a peer to peer network then the peer to peer network should have the following middleware features implemented:

- a dynamic discovering and registering players in the network

[2] https://gamedevelopment.tutsplus.com/tutorials/building-a-peer-to-peer-multiplayer-networked-game--gamedev-10074

- selecting a player for a special role
- a consensus on the current state of the game through voting
- reverting back to the previous stage when the current action is not approved or is faulty

3 State of the Art

In this section we look at few recent studies from the scientific community which use a technology related to the peer to peer distributed systems. We looked into some of the important achievements toward implementing the byzantine fault tolerance.

A byzantine fault tolerance is a distributed systems requirement for tolerating the famous byzantine generals problem [4] which is the most difficult class of failure. In a distributed system, a typical byzantine fault is when a machine gives different responses for different machines when the request is the same. Several practical byzantine fault tolerance has been attempted, using the early attempts [2, 3] as a base. Few of the most recent and interesting scientific findings are discussed.

One of the most appealing state of the art research is the implementation of a byzantine-resilient decentralized machine learning [10]. A distributed machine learning is gaining a big attention due to the need to process and use datasets that are scattered over different the local and global network. Despite the novel development of a distributed algorithms [8, 5, 6], resilience of the network to all sorts of failures is still a big issue in the real world. The research [10] focused on solving a distributed learning tasks in the presence of the byzantine failures by developing a two variants of an algorithm, "termed Byzantine-resilient distributed coordinate descent (ByRDiE)".

Again, another, appealing state of the art research is able to design a voting protocol for a human-rated spacecraft and achieve data integrity in the presence of a byzantine faults [7]. In the design of human-rated spacecraft that incorporates commercial off-the-shelf (COTS) and open standards it is difficult to meet the high reliability and fault tolerance requirements of implementing a critical spacecraft functions. Thus, the research proposes a voting protocol using a Time-Triggered Ethernet cable that allow components to reach consensus in the presence of a single Byzantine fault. "The approach is intended to reduce development and upgrade costs, lower the need for new design work, eliminate reliance on individual suppliers, and minimize schedule risk" [7].

Given that several studies has been already done since the the byzantine faults gained attention, the goal of the peer to peer card game is build a distributed system that functions in the presence of a minority number of byzantine peers. To reach a consensus a voting algorithm will be implemented together with Paxos protocol, thus making the peer to peer system resilient to Byzantine faults. A similar study [9] is done to reach an agreement in three rounds with a bounded-byzantine faults. In the peer to peer card game, a three round of consensus together with a forth round of acceptance messaging is developed to ensure the peer to peer card game is resilient to Byzantine faults.

4 Our Architecture

The architecture of this application consists of a Tomcat container acting both as server and as a unique client. Multiple clients connect to each other via a host discovery, but instead of having a traditional server maintaining all clients, our architecture is structured around clients communicating in a clustered manner via their respective servers.

The main problem with the architecture resides in how to structure it in such a way to allow for multiple clients while also allowing for a ton of computation to be done in the background. Since there are going to be multiple calls between clients for each step of the sequence it was decided to not have a single processing unit take control of the entire computation but rather break it down as to allow for each client to contain its own background computation possibilities. As it can be seen in figure 3 each client consists of a back-end and a front-end parts.

As far as web servers go , there are plenty of options on what to use. The main possibilities that were looked into was having a JAVA based server for each client, or have the front-end JavaScript run its logic in the background.

Performance is also an important part of the pick of technologies used. According to comparisons made by third parties there is a clear advantage of using Java as the main computational language . The comparison used as a reference shows a benchmark comparison between Java and Node.JS [1], the latter being perhaps the fastest JavaScript framework and it shows how much more useful for a server Java would be.

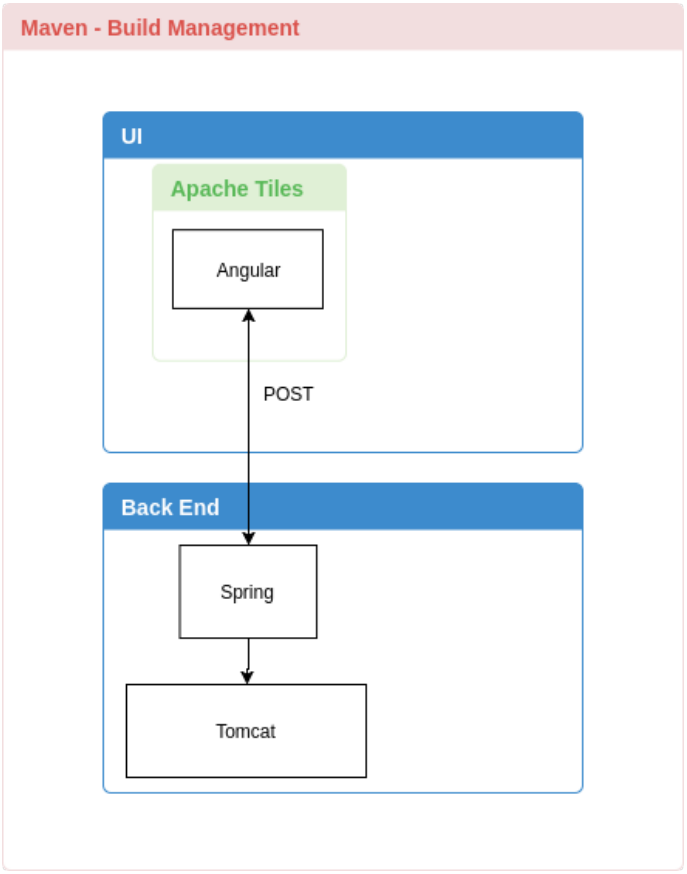

Figure 3: Basic Architecture for one client

[2][1]https://benchmarksgame.alioth.debian.org/u64q/javascript.html

4.1 Back-End

Spring

Spring is one of the most popular Java frameworks designed speficially at enterprise Java. It is extremely lightweight and easy to use and also contains multiple extensions for different type of applications, one of which, the web module, is the reason Spring is the backbone of this project. Spring development allows for a unique and clear programming model which separates logic in a manner similar to aspect oriented programming. This code separation forces developers to write in a layered manner which provides easier development possibilities based on tests and debugging and all calls in this project follow the same structure as shown in 4.

Interceptors

Interceptors are classes that monitor each process and allow for additional code being executed at any stage of each process. This project uses multiple interceptors, some of which are designed to check for failures at any stage, others are used to create a new starting point for data sharing between servers.

Restful interface

Every single call between client-server and client-client is an HTTP call based on the RESTful API. As such each call is mapped to a particular url and contains information in its header, mainly data that is passed around. The data is always in JSON format, meaning that the UI will be able to use it without any special conversions done. Spring allows for Java to receive and send JSON objects via HTTP as it has in-build object constructors which will build any json array into an object at any end.

Controllers

The Controllers are the main incoming entry-point. Each controller is mapped to a particular URL and is only active when this url has been accessed. They are managed by specifying the type of incoming data thus allowing for the same url to be used for different http calls. This is useful as it allows for interceptors to monitor the same url with the same logic behind them while simultaniously allowing for different data to be passed around.

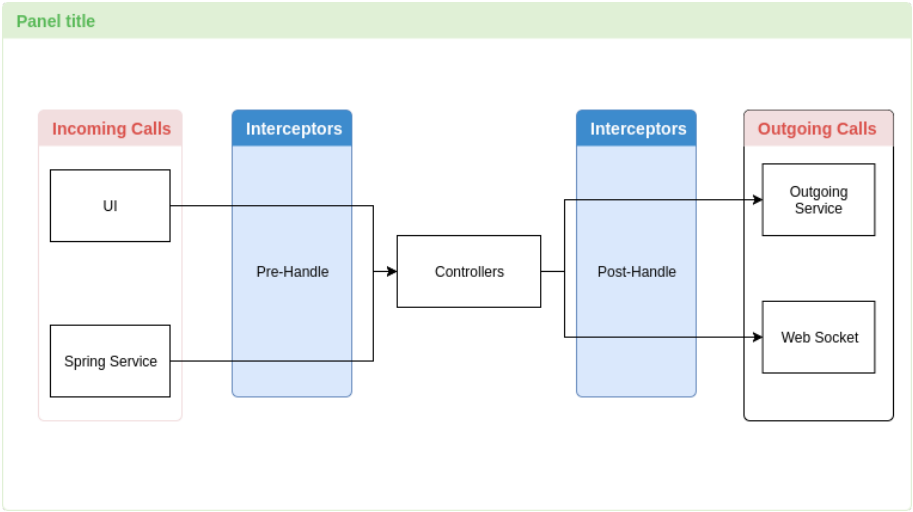

Figure 4: Back End process flow with regard to Controllers and Interceptors

4.2 Tomcat server

Tomcat is an open source container for a web application. It provides pure java http web server environment which includes Web sockets and a Java Servlet. The servlet is a java program that can request and respond any type of request. It is of imperative importance as it allows for HTTP calls to be accessed via the java code running in the back end. The Web Sockets are a protocol over a TCP connection and are what allows for the server to talk to its client without the need for a request-response logic.

4.3 User Interface

The UI was designed to be as simplistic as possible as it is not a significant part of the project, therefore the tools that are used were picked to be as simple and quick to deploy and develop as possible.

4.3.1 Angular

AngularJS is a top level framework for JavaScript development. It allows for MVC modularization of components thus creating a layered architecture. One of its biggest advantages comes with providing different controller objects for different HTML elements in the same page. This is very handy as it allows for the cards of different players to be displayed differently and have different objects to control their behaviour in the same client.

4.3.2 Tiles

Apache Tiles is a template framework for java applications. Its primary use is to fragment resource load for web applications. These fragments can be loaded separately and in an inherent order so that resources are only loaded where they are required. This allows for structuring the UI components in such a manner that the angular components will be loaded only when and where required, thus lowering the bandwidth use and load speed.

4.4 Maven build mechanism

Maven is a project management tool mainly used for project build and resource gathering/deploying. It is used to define how the project will be compiled, packaged and deployed. It is responsible for providing a developer with all the libraries he/she requires for any project automatically while also keep track of their correct versions. Maven enhances communication between team members with regard to the tools used for a project.

5 Relation to Distributed Systems

This chapter discusses aspects of the card game that are related to the distributed systems and how they are Incorporated.

5.1 Leader selection

Due to the decentralized nature of a distributed system a coordinator or a leader is required to monitor the system state changes. A leader selection is a crucial part of the peer to peer network built for the card game. The simplest and intuitive form of leader selection is the bully algorithm where a peer with the highest identity number (*i.e.* IP address) is selected to be the leader. For the peer to peer card game the bully algorithm has been adopted and implemented as a dealer selection procedure.

In the peer to peer card game, a leader is required in order to carryout certain roles. One of the main roles is shuffling the deck of cards and dealing out cards to the peers. Additionally, a dealer is expected to be the first player in the game and sets the player turn. There is no specific preference for becoming a dealer therefore all the peers have the same chance of becoming the dealer, thus start a dealer selection procedure.

After all the players have discovered each other they select a peer they prefer to have as a dealer and send it to every other peer. All the peers will do the same and they end up with the list of the peers and a peer they prefer to have as a dealer. If there is a peer selected by a majority of the peers then that peer is acknowledged as a dealer. If there is no peer with a clear majority, then a peer with the highest identity (IP) is selected as a the dealer.

Once the dealer is announced, it is responsible for shuffling the cards and starting the game as a first player. Additionally, it is required to set the player's turn (*i.e.* who play after who) and announcing this to the other peers so they can have a totally ordered view of the players turn.

5.2 Fault Tolerance

One of the main reasons for building distributed systems is the single point of failure in a centralized architecture. For instance, if the central server in the centralized card game dies then no one can play any card game until the server is up and ruining again. Building a distributed system, solves the issue by making the system resilient to the failure of a single node(peer) and removing a single point of failure like a centralized server.

In the peer to peer card game a fault tolerance is needed to enable the game to go on even though some of the players are crashing. In addition, another form of fault tolerance is included to enable the system to revert to the previous state when the current state runs into errors.

A peer that crashed after a successful game state change is removed from the list of players by all the healthy (*i.e.* not crashed) peers and the game is continued by the next player. If a peer crashes in the middle of an action, *i.e.* when all the players are deciding if they should execute the action or not, then the player is discarded but the system (all the peers) reverts to the last game state change and the game is continued by the next player.

To do a system revert, all the peers will keep a linked list of all the successful state changes. Only state changes that are agreed on everyone and executed by everyone are attached to the linked list. To have the agreement and reach a consensus among themselves a voting algorithm is required.

5.3 Voting algorithm

In a distributed environment more often than not every decision has to be made by someone at some point in time in order to allow for a successful execution of events. In the case of a system

where there is one main server, this decision is taken by the server. In the case of this project, however, such a decision can not be taken by only one server and has to be done based on a peer to peer system. This multisided decision based system suggests that every participant has an equal vote over a decision.

A Voting system has been proposed as a solution on how to monitor, evaluate and execute decisions by multiple peers. This system forces all peers to have a say on how a certain event will be proceeded and based on a majority of votes, executed. This means that at every event, all peers involved will take a vote on how to continue and the action that will take place depends on which action has the majority of votes.

In the current architecture, it is preferred that no leader exists for all the actions, but the peers will proceed based on results from a voting system.

5.4 Reliable Multicast

Due to the existence of multiple receivers of a message in the peer to peer network it is mandatory to have a reliable and ordered multicast for several reasons. Depending on the type of messages, some messages are required to be viewed by all the peers in the order they were sent, some messages in the order they arrived, some messages tolerating a happened-before ordering.

For the reliability, the peer to peer system needs to make sure a message sent by a peer gets to all the peers no matter what happens to the sender. This is Incorporated to the peer to peer card game by letting the peers forward(gossip) the messages they received.

From the card game point of view a total ordering of the actions is required, so that all the peers have the same view of the game state. For instance, if card **X** is thrown before card **Y** then all the peers need to have this actions executed in the same order that it happened. This is handled by having all the peers agree on a certain action before anyone executes it with a Byzantine resilient consensus protocol.

For the message passing among the peers during the consensus and voting procedure, a total ordering is not required. In fact a partial ordering is only required to enforce collecting the votes before computing a majority vote, or to ensure sending a blank(prepare) request before accepting any vote.

6 Solution details

This chapter looks further into the most complicated and crucial parts of the architecture, how they are designed to work and the reason for their particular usage.

6.1 Dynamic Discovery

The way this project has the dynamic discovery incorporated is to take the local IP of the machine and loop through its last digits with a range from 1 to 255. At every step of the loop it will ping the address to see if one is reachable, in order to check whether a machine is even online. After that it will send a request to the machine and expect an answer. What makes this method secure and reliable is the fact that a call to a device will be done in a particular url mapping, so only servers which respond to this particular mapping will be added. An additional step to secure is that this call will expect a particular result, being an object which only this application will know about. Meaning that

even if there is a chance for any random localized server to answer to this url, it will only be accepted as part of the application if it also obides by the rules of the application and return a correct result.

```
1   public void discoverDevices() {
2               byte[] ip = localhost.getAddress();
3
4               for (int i = 1; i <= 254; i++) {
5                   try {
6                       ip[3] = (byte) i;
7                       InetAddress address = InetAddress.getByAddress(ip);
8                       if(address.getHostAddress().equals(
9                           localhost.getHostAddress())) {
10                          continue;
11                      }
12                      if (address.isReachable(100) ) {
13                          Device discoveredDevice =
14                              discoverDevice(address.toString().substring(1));
15                          if (discoveredDevice != null) {
16                              addDevice(discoveredDevice);
17                              break;
18                          }
19                      }
20                  }
21                  } catch (Exception e) {
22                      e.printStackTrace();
23                  }
24              }
25          }
26
27
28
```

Listing 1: Code for discovering devices

6.1.1 Drawbacks

An obvious drawback to this implementation is the requirement to have all nodes in the same localized cluster. Only servers which reside in the same ip range will be called. This can, of course, be extended to loop through more IPs but this will inevitably slow down the system to an extend where it is no longer viable as a solution. As seen at line 12 in 1 the server will wait for 100ms for a response from a different machine, meaning that for every variable in the range 255.255.iii.0-255 it will wait for 25 seconds, or if allowed, the server will wait a total of 108 minutes.

6.2 Leader selection

To select a leader, every player creates a message by putting its IP and IP of the player it prefers to have as a leader. The message created is then sent to the rest of the players in the game.

In the general sense after one step every player will have a list of all the players and their corresponding preference for a leader. Since the implementation of the communication among the peers is point to point HTTP calls, a player will iteratively send its message to the rest of the players. The other players will do the same with their respective messages.

All the players will look for a peer preferred by a majority inside the list for the leader preferences of all the players. If found then the peers send acknowledgment to the leader and the leader will start the game. If there is no majority a player with the highest IP is chosen as a winner and thus get the acknowledgement from all the players.

6.3 Paxos algorithm

When a player(peer) wants to execute some action, then a three stage procedure is taken by all the peers before executing the action successfully. The action can be taking a card, throwing a card, or selecting a new leader. In the Paxos consensus algorithm there are three different roles: the proposer, acceptor and learner. There is no restrict rule to how many peers play how many roles, but rather the peers can play any of the roles depending on the current game state. For instance, a player that initiates an action is a proposer and the the rest of the peers act as the acceptors, and eventually all the peers will be learners. The three stage voting and consensus is explained below, followed by the the different faults tolerated.

Before proceeding, a player can only start an action if the leader approves that it is its turn. Therefore, this eliminates the risk of having multiple votes in the peer to peer network. Paxos algorithm relies on sequence numbers to achieve its guarantees but the sequencing is irrelevant to this peer to peer card game because a leader allows only single player at a time to propose a value or action.

6.3.1 Paxos stages

Stage 1 - Request

First, we have the prepare phase. A player sends a blank (prepare) request to the other players. When a peer receives this blank vote it adds its IP(identity) to the message and forwards it to all the players, and everyone does the same. This blank vote is re-transmitted by all the peers to make sure everyone gets it despite any crash failures. As shown in figure 5, at the end of this stage all the peers will have a list of all the devices participating in the voting.

Stage 2 - Prepare

Next, we have the actual voting phase. When a peer has received a complete list of the players then each player appends its vote in the previously created blank vote and sends again to all the devices. All the peers do the same and end up having a copy of every player's vote in their local machine. Here, a byzantine fault can occur, if a device sends different votes to different players. By the end of this phase all the peers will have received the votes of all the players.

Stage 3 - Majority

The third phase is the actual majority voting computation, which is done by all the peers in their local machine. The peers calculate their local majority result and send it to the other peers.

If there were no crashes, failures, or even byzantine peers then all the peers will return the same majority results. However, if there are byzantine peers or players with arbitrary faults then all the peers will not have the same majority results.

Stage 4 - Acceptance

As a forth stage all the majority results from all the peers are collected. If all the peers reached to the same majority result then everyone executes the action and the game is continued by the next player.

On the other hand, if all the peers do not respond with the same majority results, due to any reason, then the voting procedure is restarted from stage 2.

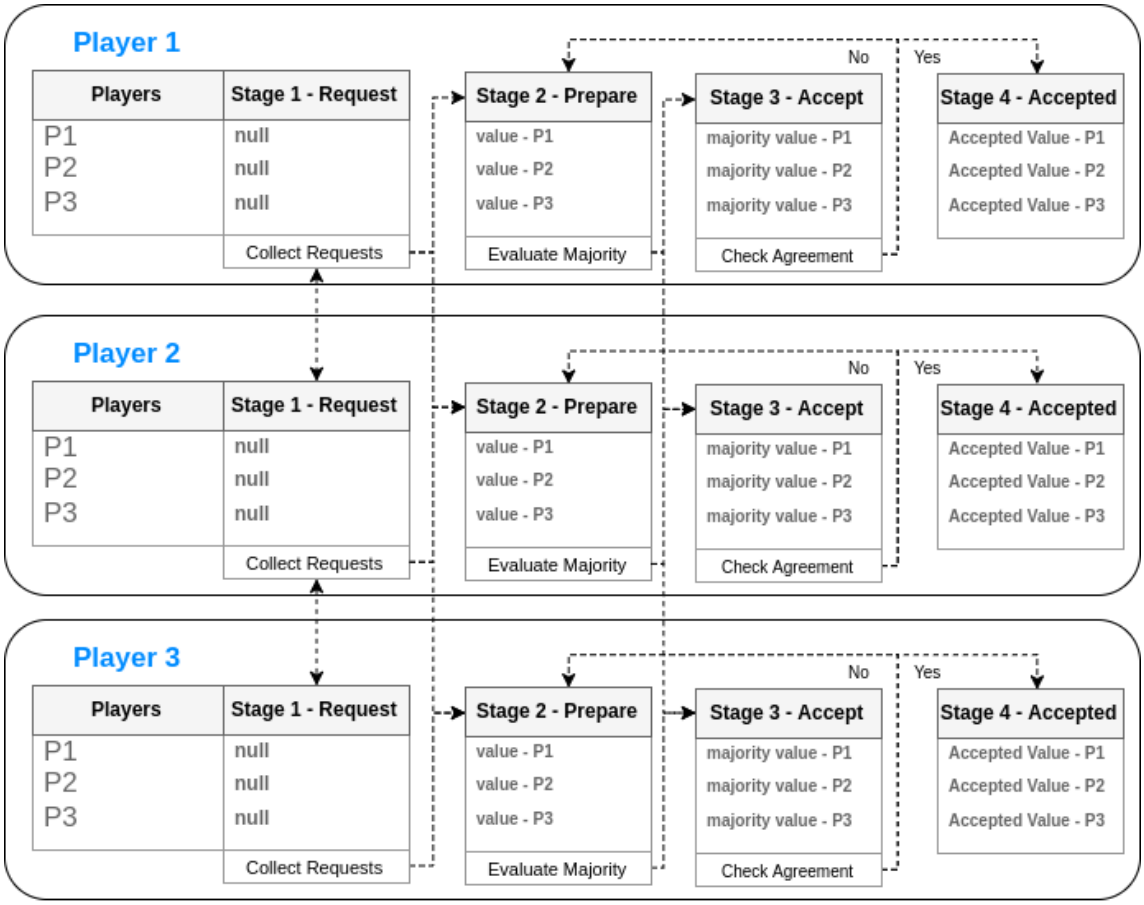

Figure 5: Paxos consensus with Byzantine Fault Tolerance

6.3.2 Tolerated faults

Here are a list of failures that are tolerated in the peer to peer card game that uses a three round Paxos consensus protocol. There are two categories of failures that are tolerated in this system: The participation failure which can be a crash failure or an omission failure and the Byzantine failure.

Participation failure

If a player crashes without sending the request to everyone (crash failure) or if a peer is not replying to a request(Omission failure), then two things happen that resolve this. Firstly, the peers that received the request before the crash will forward the request to the other machine. As a result the game will continue. Secondly, if a peer is not sending its vote or not sending its majority result, then that peer is removed from the list of participants.

Byzantine faults

This fault can occur in two ways: firstly when a peer(byzantine) replies an incorrect or inconsistent message on purpose in response to a message request, secondly a peer is sending an arbitrary message as a reply due to malfunctioning or unknown reasons (Arbitrary failure). Based on when in the game state this issues occur, they are resolved as follows.

The first is when an arbitrary message or a byzantine message is put in the votes at stage 2. This is resolved on the third stage: a vote with a majority is selected from all the votes collected thus implying an arbitrary vote (or a Byzantine vote) will not get a majority because not all the peers are voting arbitrarily or are byzantine. If this by any chance happens, meaning a peer computes a majority result which is byzantine value, then it is resolved by the next approach.

The forth stage is introduced to exactly make sure no faulty peer is participating in the voting procedure. It makes sure all the peers reach to the same majority vote results. It restarts the voting if there is a difference in the majority vote result. This termination condition of everyone agreeing on the same majority value ensures the byzantine fault tolerance in the system.

6.4 Concurrency

The architecture of this project is build around peer-to-peer communication involving a multitude of calls from and to any machine while simultaneously working on the same object from within an environment. On every such system the problem of concurrency will inevitably arise and if not corrected, result in breakages and wrong results. The architecture of this project recognizes only two situations in which a concurrency issue is possible. The way this issue is fixed is to shift focus on where a specific object needs to be changed. In the Device Discovery Scenario 6.4.1 the object will not be locally changed but instead a new version of the object will be send to all subsequent servers so that they can replace their old value with the current value and thus evading potential localized concurrency issues. The second is when votes from other servers in a random order and asynchronous order as described in 6.4.2.

6.4.1 Device Discovery Scenario

One scenario when this is observable comes when a client crashes for any reason and comes back into the game immediately after. The first thing a client does upon start is to assign it self a unique UUID value and use it to identify itself in a game. It will then search for other clients in order to create a game. At some point the client will discover the previous game and join it. He will then immediately receive the list of other machines in this game, and in there , he may or may not exist as a previous player, depending on whether he has been removed from the game or not.

6.4.2 Vote Application Scenario

Every server stores a temporary vote object of the current vote circulating between the servers. The servers agree on values and apply them on their respective local temporary vote object. Since the calls are asynchronous they will arrive in unknown times and they will write new values or change existing ones in a random manner. The way this is managed is to have a separate temporary object that only exists for the duration of the current receiving call, and when this call is finished this separate temporary object will copy its values into the actual temporary vote object. The other calls arriving in the same time will get either the old value of the temporary vote object or an updated one from another arrived call and again create a separate copy of the object and work on it. This way no two machines will work on the same object at the same time.

7 Result

Device Discovery

The initial device discovery works with no errors. All devices discover each-other and synchronize accordingly. Faults have been detected when devices crash and then get back into the game, but these faults are corrected at the first stage and the game resumes to a correct stage.

Voting

What happens with voting issues Due to the nature of the a distributed system it is anticipated that voting issues may occur. It will be necessary to note however, that the paxos consensus with a combination of byzantine fault tolerance should not allow any faulty votes to be processed. The elimination of issues and the cause of such result to sharing that cause to all other participants and the issue will be collectively eliminated.

Leader Selection

The initial leader selection works based on the voting paradigm the game revolves around. The initial leader is used only to jump-start the game itself. After a turn has passed and all the evaluation and voting for cards has been completed, the current leader will call the next leader for a notification to start the next turn, thus passing the leadership role.

Performance

Due to the constant message passing it is assumed that there will be a drawback with regard to performance due to the latency between messages. Single machine performance poses no issues as the amount of computation in a single machine is insignificant. Therefore the only performance issues arise from the network speed itself. This is a drawback not only with the current architecture but with every single distributed system which is based on a networked architecture. Therefore it is safe to say that this game will work without any delay provided the network is powerful enough.

References

D. Alioth. The computer language bench marks.

M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg. Byzantine fault tolerance, from theory to reality. In *SafeComp*, volume 3, pages 235–248. Springer, 2003.

L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

J. Liang and J. Mei. Fully distributed consensus algorithms for double integrators under heterogeneous topologies. In *Control Conference (CCC), 2017 36th Chinese*, pages 8760–8764. IEEE, 2017.

J. Liu, S. Mou, and A. S. Morse. Asynchronous distributed algorithms for solving linear algebraic equations. *IEEE Transactions on Automatic Control*, 2017.

A. Loveless, C. Fidi, and S. Wernitznigg. A proposed byzantine fault-tolerant voting architecture using time-triggered ethernet. Technical report, SAE Technical Paper, 2017.

N. A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

M. Malekpour et al. Achieving agreement in three rounds with bounded-byzantine faults. 2017.

Z. Yang and W. U. Bajwa. Byrdie: Byzantine-resilient distributed coordinate descent for decentralized learning. *arXiv preprint arXiv:1708.08155*, 2017.