DistributedBattleship Proposal (Draft)

Aleksandra Budkina, Graham Brown, Harryson Hu, Larissa Feng, Sharon Yang

Introduction

Distributed Battleship is a game of battleship between two groups of players where each turn the players must arrive at a consensus via PAXOS. It is possible for a group to be replaced by a 'dumb' artificial player which will make random or preprogrammed moves. The core difficulty of the system lies in the implementation of the PAXOS algorithm, alongside handling client failure gracefully.

Problem

Playing a game collaboratively is difficult as:
- Each move for each team requires a consensus
- Players will join and leave throughout the game
- Game state must be shared
- Detecting end game state

Approach/Solution

Each team is represented as a group of players
- Players within a group share a game state
- Groups will arrive at a consensus through Paxos
- Once a decision has been reach, the move will be broadcast to other team.
- There exists a server which handles new client joining and client failure
- The game starts once there are two player groups.

Assumptions & Constraints

- The network server that handles player registration and leader selection never fails.
- Each player has a unique ip port pair
- Messages don't get corrupted between entities
- Players don't send any malicious messages and can't 'hack' the server to join 2 teams

Topology of network

Participants are: server, player nodes. Nodes are individually connected to the server. Connection between nodes is a peer-to-peer (many-to-many) connection within the teams and a star (one-to-many) connection between the teams where the Leader of one team is a central point of connection for another team. Computer-player node is a single node which communicates via one-to-one connection with server, and one-to-many with the team of user-players.

Start of a game

- The first player to join: Connects to the server, receives from it an empty list of other players to connect, Leader status because it is the first player, and TEAM # 1 if the game mode is ‘Against People’; receives a PC IP:Port address, Leader status because it is the first player, and TEAM # 1 if the game mode is ‘Against Computer’.

- The second player to join: Connects to the server, receives from it an list of another connected player, who the Leader is, and TEAM # 2 if the game mode is ‘Against People’; receives a PC IP:Port address, list of other users IP:Port, and TEAM # 1 if the game mode is ‘Against Computer’

- Subsequent players to join: The server joins player to the smaller team if the game mode is ‘Against People’; receives TEAM # 1 if the game mode is ‘Against Computer’. The player receives all IPs of team members that it needs to connect to.

Server behavior

- A new player will register itself with the server. The server will provide the player with the IPs of the other players on the same team for the new player to connect to.
- If a new player is a Leader, it will receive the addresses of the opposite team for sending update statement after a move was accepted.
- When the server detects that a leader of a group has failed, the server will pick a new leader from the group and send the new leader a list of all IPs that the leader needs to communicate with in its role.

Semantics

All connections are via RPC

Failures/disconnect:

- One player disconnects:
    - Once server detects that player has disconnected, pings all players of its team and the Leader of the opposite team that this occurs.
- Last player on a team disconnects:
    - The server will broadcast that last player went offline. Remaining team wins by default
- Team Leader disconnects:
    - The server will know about the disconnection of the leader. The server will then pick another member of the group as the Leader and notify it about a new status. Also, server will send to a new Leader the addresses of the opposite team to connect.
    - The new leader checks game state and rebroadcasts the latest consensus to the opposite team.
    - Options: Next most senior player becomes the leader. When players discover when leader is disconnected (see case of one player disconnects), they ping the server for the next player or (tbd) they

coordinate amongst themselves (in this case, each player will get a player number from the server)
    - Extension: Have a scoring system where users who have made more decisions that agree with the consensus get more points/karma, and the player with the highest karma becomes the next leader
- When a decision is unable to be reached by consensus:
    - Player doesn't decide on move: The timer is set on each player. If player doesn't choose to move, the system will send NOOP command to other team members. In case all team members sent NOOP, another team wins.
    - Player disconnected: see handling of disconnections.

End of the game:

- When a game ends successfully (i.e. one team sinks all the other team's ships):
- For a game to end unsuccessfully, it is due to all of one team (or both) disconnecting, in which case, refer above.
- All players disconnect, server detects and clears all state, idles.

Future additional complexity and optimization

- Handling Computer Player disconnection and reconnection
- Choosing a Leader to unify the final decision faster (based on the # of previously correct answers)

Testing Plan

Each module will have a set of local unit tests as well as a script to automatically start up a server with a number of clients, who then can connect and play.

Package Structure and Description

```
/go/src/
        /battleship
                battleship.go
                battleship_tests.go
        /paxos_network
                paxos_client.go
                proposer.go
                acceptor.go
                learner.go
        /rpc_network
                rpc_client.go
                rpc_client_tests.go
        /rpc_server
                server.go
                server_tests.go
```

The battleship application is responsible for running the game, and holds the game state. It can make a choice regarding the current game state (where to fire, win/lose, etc.) and will submit the new game state via the paxos_network. After submission, it will wait for a new game state.

The paxos_network is responsible for coming to a consensus regarding some choice. It does not have any concept of what the choice means (aka which tile you are shooting to) beyond that two choices differ and must be decided between. The paxos_network contains the local paxos group (our team) and the foreign paxos group (their team). The paxos_network communicates with those groups via the rpc_network.

The rpc_network is responsible for handling communication between a group of clients. It contains a collection of rpc clients to which we can send and receive data. The rpc_network handles failures and retries if needed. Heartbeats will be sent on the rpc_network invisibly to upper layers.

The rpc_server is responsible for detecting new clients joining the rpc network, and for detecting clients failing/leaving the network. It operates solely upon the rpc_network, and has no concept of paxos or the battleship game. When the set of rpc_clients changes on a join or disconnect, it will notify the paxos_network.

SWOT Analysis

Strengths • 4/5 group members have worked together previously • Larissa -> devops • Alex -> Project management & questions • Sharon -> math/algo • Harryson -> Clean code • Graham -> Structure and Naming • Simple user interface	**Weaknesses** • Difficult for everyone to meet at the same time in-person • All group members are relatively new to Go • None of us have worked with Paxos before
Opportunities • Learning Paxos in lecture soon • Create a fun game to play, which implies a simple demo • Azure provides lots of resources to set up game players + servers to test with • We are able to use 3rd party libraries during project implementation	**Threats** • Paxos is a Complex Algo to implement • Work from other courses/commitments • Showing paxos will be hidden from the game clients, and so we have to have a way to reveal the paxos coordination.