CS-6650 Final Project: Executive Summary

TEAM MEMBERS

Cole Garbo
Luna Szymanski
Rohan Subramanium
Abinashi Singh

VIDEO DEMO

https://youtu.be/SvluKSd3HM8

OVERVIEW

The purpose of this project was to demonstrate key distributed systems concepts in a practical and functional application, which the team decided to implement as a simple turn-based multiplayer game. This allowed the decision-making and coding process to be focused on the distributed algorithms and implementation and not the game logic itself, though the two could be tied together to create a more robust and scalable system. The design documents created for the initial project proposal provided a guideline for the program architecture that was expanded upon as needs and challenges became clear - for example, a change in implementation was needed to allow for multiple clients to listen for responses from the Coordinator itself - but without sacrificing functionality or needlessly expanding scope. The end result is an organized, documented, distributed game, which adheres to the initial architecture overview and delivers a stable and fault-tolerant product.

The final game architecture is divided into three packages: Client, Coordinator, and Server. Each package provides all of the classes, interfaces, and utility methods needed for the key class to function and communicate, allowing for a simple structure where the key class is created (ie, a Client), a remote interface provides all the RMI methods required (ie, ClientDriver), and a utilities class allows implementation (ie, ClientImpl). This structure is mirrored across all packages. RMI is used for communication across all packages, allowing innate multithreading for servers and clients, and a Coordinator handles safe operations between the Clients and Servers. Clients should be able to interact with the Coordinator asynchronously, never noticing any delay between the receipt and return of their messages, while the Coordinator ensures the Servers store mirrored replicas of sensitive data - the game database and synchronized voting mechanics.

GAME RULES

The implemented rules are based on the popular games “Apples to Apples” or “Cards Against Humanity.” A Client launches and waits for the game to begin, and will receive an announcement when the game starts. From there, a line of text is displayed with a blank (“_”) representing a word to be filled in. The Client enters their response, and then all player

responses are displayed on screen for voting. The Client enters the number of the response they like best, all votes are tallied and displayed for each player, and gameplay continues until one player reaches a score of 10. In the event of a tie, both players win.

KEY DISTRIBUTED CONCEPTS

1. The application uses Java RMI to handle communication across separate virtual machines, managed by a single coordinator. The Client, Coordinator, and Server packages all set up and connect to several registries provided to enable accessing a series of remote interfaces - again, one each for Client, Coordinator, and Server. Each remote interface is then implemented and backed by a series of utility methods to facilitate communication between remote classes and make appropriate gameplay decisions. Each class functions simultaneously as a producer and consumer of messages, waiting to receive gameplay updates and broadcasting responses.
2. Coordinator design patterning was implemented to manage application connections, communication, and resources, and the Coordinator represents the only centralized functionality. Connected Clients and Servers are all registered with the Coordinator and then grouped to allow for coordinated multicasting across all connections. The Coordinator can then manage commits to the Servers to ensure that all information is distributed across the replicas and there is no data lost, dropped, or duplicated.
3. Java RMI is naturally multithreaded, which made Client-Coordinator-Server communications simpler overall. However, this meant managing mutual exclusion more directly. The application has to be responsible for establishing synchronous access to shared resources, blocking off critical sections of code from running in multiple threads at one time. This was implemented around the application's voting mechanism, where multiple clients may call the operations to vote asynchronously, but the code to increment voting tallies within the server is synchronized so votes may not be recorded simultaneously and dropped or overwritten, ensuring there are no conflicts.
4. Fault tolerance and stability has been achieved using timeout mechanisms to ensure dropped clients are handled appropriately and moved on from. The game can be progressed with the remaining players if any individual response or vote is taking too long, ensuring each process can continue regardless of participants. This area could be improved in future iterations by implementing a "ping" mechanism from the Coordinator to all registered Clients and Servers, allowing any dropped Clients/Servers (those that miss a ping check) to be removed from their multicast groups and, in the case of Servers, restarted with a mirror of an existing and running Server. Similarly, an improvement for the Client would involve allowing a dropped Client to reconnect with the same credentials, and rejoin the existing game with their prior state and score.
5. Group communication is handled by the central Coordinator. Registered Clients are added to a client pool, and registered Servers are added to a server pool, representing groups which can be broadcast to simultaneously. Each turn begins with a broadcast to each Client from the coordinator; the Coordinator can then receive returned messages asynchronously, attributing them to the individual Client, and partitioning them out for

updates or storage to the corresponding Servers acting as backing stores. Communication to the Servers is also handled similarly, with the Coordinator sending out requests to store and then commit to the connected Servers as a group.

WORKFLOW / HOW TO RUN

To run the game, the first thing to be started must be the Coordinator. This coordinator starts up five Servers with all of the game cards stored in a list for retrieval each round. Once the Coordinator is running, Clients that start will bind its own RMI registry, lookup the Coordinator's RMI registry, and request the Coordinator to bind back to them. The Coordinator notifies all of the Servers of the new Client's ID and waits for more Clients to bind and fill up the minimum player requirement before starting. After the last Client binds, the Coordinator refuses any other Client registrations and starts the first round.

At the start of each round, the Coordinator requests a random card from one of the Server replicas. Once a card is chosen, the Coordinator tells all of the Servers to delete the card to avoid duplicate cards being drawn in a single game. The same card is broadcast to all of the clients using a thread for each in order to receive responses asynchronously. The count of responses is handled with mutual exclusion to ensure the count of responses is accurate for each round. After collecting all of the responses (or if any Client doesn't respond within 15 seconds), the Coordinator stores each response with the Servers, and broadcasts the list of all responses to all clients for voting, which is also handled with threads for concurrency control and asynchronous submission of votes. Votes are tallied and stored on the game server replicas, and referenced to see when a winner has been decided.

The following is a step by step walkthrough of a typical game:

1. Open Coordinator and have it bind to registry
    a. Coordinator opens 5 Server replicas to track the game
2. Open the correct number of Clients to bind to the Coordinator
    a. Clients will bind and request the Coordinator to bind back
3. Coordinator starts a round by choosing a random card from Server and broadcasting to all the Clients. The selected card is removed from all Servers
4. Clients have 15 seconds to respond with a String to fill in the blank
5. Coordinator collects responses and broadcasts a list of responses each with a response ID for the Clients to vote on
6. Clients submit votes to the Coordinator
    a. Coordinator tallies votes and updates all Servers
    b. If a client submits an invalid vote (not a response ID listed), no point is allocated for that vote
7. Coordinator checks for winner based on Servers' vote totals compared to win condition
    a. If no winner is found, repeat steps 3-7
    b. If a winner is found, report the winner to all Clients and close down all clients and the coordinator smoothly