Final Report: Co-Type

Final Report for the Distributed Systems Course 2025-2026

Simone Ceredi
simone.ceredi@studio.unibo.it

August 4, 2026

CO-Type is a distributed client server cooperative typing game. The main idea of the project is to build a robust and scalable system, with error recovery and state reconstruction from partial and potentially inconsistent data. The system is composed of a broker, which is the only endpoint known by the clients at startup, and multiple servers. There is no upper limit for the number of servers that can be used. The number of servers does not have to be defined, and new servers can be created at any moment. The game is designed to recover from client, server or broker crashes while having near zero downtime. If a client crashes during game, then the session is paused. If the broker crashes, then all the knowledge it had is reconstructed. If a server crashes, all the lobbies are transferred to a different server.

Disclaimer (if needed)

During the preparation of this work, the author used large language models to check for syntactic or semantic errors.

After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the final report/artifact.

1 Concept

The project is a distributed cooperative multiplayer terminal typing game. In the game players receive a code snippet to write and act as a distributed keyboard to type the code together. Each player can only type a subset of all keys. Unlike traditional game server architectures, in which, when a server crashes, the state is recovered from persistent logs, or via replication, in CO-Type there is no backup infrastructure: the game state is recovered via the partial, and potentially inconsistent knowledge known by the clients.

What it consists of: a terminal application with a simple user interface, supported by a single broker service, used for server discovery, and a arbitrary amount of servers. Both the broker and the servers run on a Kubernetes cluster.

Use case: after launching the client, a user picks a username and a lobby id, creating a new lobby or joining an existing one. Players share the lobby id via an external tool such as a messaging app. Inside the lobby, any player can assign the others a subset of keys they are allowed to type, and readies up once setup is complete; the game starts when all players are ready. During the game, players interact purely via keyboard, jointly typing a code snippet into a single shared buffer. Once the snippet is completed correctly, a statistics screen shows completion time and characters-per-second. No user data is persisted: there is no database and no local storage, only ephemeral in-memory session state held for the lifetime of the lobby and discarded once it ends.

Why is distribution needed:

- **geographic distribution:** the game session is managed directly by the server, having multiple ones allows for geographic distribution;
- **fault tolerance:** having multiple servers allows the system to quickly recover upon crashes;
- **horizontal scaling:** in order to manage thousands of sessions without the need for vertical scaling.

2 Requirements Elicitation and Analysis

The system has been designed to focus on availability rather than consistency, as happens with most games.

Functional requirements:

- *FR-1*: A user shall be able to create a lobby or join an existing one by entering a lobby id and username. *Acceptance criterion*: entering a fresh lobby id creates a new lobby and returns its id to the creator; entering an existing lobby id places the user in that lobby's player list.
- *FR-2*: Any player in a lobby shall be able to assign other players a subset of permitted characters they may contribute. *Acceptance criterion*: after an assignment action, the targeted player's client immediately reflects the new permitted character subset.
- *FR-3*: A player shall be able to mark themselves as ready. *Acceptance criterion*: toggling ready status updates that player's state as visible to all other lobby members.
- *FR-4*: The game shall start automatically once all players are ready. *Acceptance criterion*: when the last unready player marks themselves ready, all clients transition to the gameplay screen.
- *FR-5*: During gameplay, players shall jointly type a shared code snippet. *Acceptance criterion*: same as FR-6.

- *FR-6*: Players see the state of the game while playing. *Acceptance criterion*: the current shared buffer contents, and each player's assigned characters, are rendered on every connected client and update as the game progresses.
- *FR-7*: When the game ends the system shall display the game statistics. *Acceptance criterion*: upon correct completion of the snippet, all clients display total completion time and characters-per-second.
- *FR-8*: The system shall recover an in-progress lobby's state on game-server failure by migrating it to a healthy server, without relying on persistent storage. *Acceptance criterion*: when the hosting server process is killed mid-game, all connected clients reconnect to a newly assigned server, and the reconstructed buffer matches the last state known to the clients.
- *FR-9*: The broker shall route new lobby-creation requests to an available game server. *Acceptance criterion*: a lobby-creation request is always assigned to a server that is currently reachable and reporting healthy status to the broker.
- *FR-10*: The broker shall reconstruct its list of active lobbies after a broker restart. *Acceptance criterion*: after killing and restarting the broker process, querying it for active lobbies, after a spool up period, returns the same set of lobbies that were active immediately before the restart, without requiring any disk-persisted state.

Non-functional requirements:

- *NFR-1* **Fault tolerance**: The system shall continue operating correctly despite crashes of any single game server, the broker, or a client. *Acceptance criterion*: forcing crashes on game servers, the broker or clients shall still allow, after a reconciliation period, the game to be played, without manual intervention.
- *NFR-2* **Responsiveness**: Local input shall be reflected in the client's UI immediately via client-side prediction. *Acceptance criterion*: a keystroke is rendered in the local client's UI, independently of the latency to the server.

Constraints:

- **No persistence**: The system shall not require any database or disk-backed storage to operate or recover. *Acceptance criterion*: the deliverable operates correctly with no database or external persistence layer present in the deployed containers.

Implementation choices:
Golang, gRPC, test-driven development and trunk based development as branching strategy.

2.1 Relevant Distributed System Features

- **Fault tolerance:** the central thesis of the project is managing crashes and errors without the need for persistent storage. The system shall be able to recover from crashes of any of the three components (client, broker and server).
- **Scalability:** splitting the broker from the game servers allows the use of a very light discovery system, able to handle a large amount of clients, as it's only needed to retrieve routing information from an in-memory storage. While the game servers, even if they are also lightweight due to the nature of the game, can scale horizontally indefinitely, allowing the handling of large amounts of games.
- **Transparency:** there are two main kinds of transparency relevant in the system:
  - **Failure transparency:** no user intervention shall be needed upon any kind of crash, apart from if a user crashes from the game and has to reconnect.
  - **Location transparency:** even if there are different servers that can manage a game, the users do not have to care about which one has been chosen to host the game.

3 Design

The system has been designed following a client-server architecture, with a particular focus on fault tolerance and scalability.

3.1 Architecture

The system uses a **client-server** architecture. A high level representation can be found in fig. 1.

The system is composed of three main entities, a broker, multiple servers and multiple clients. The architecture is structured so that there is a single fixed endpoint, the *broker*, that can be contacted by both *clients* and *servers*. The *broker* works as a discovery service, *servers* register to it, and *clients* ask for an available server to use for the game.

This structure allows the use of multiple servers, granting error recovery capabilities to the system, other than scaling. The scaling is achieved by tracking how many games are running on each server, so that when a client wants to create a new lobby, then it is instructed to the server with the lowest workload.

The components communicate with each other via gRPC, when communicating with the broker, both clients and servers, use a request-response interaction; while when managing a game, a gRPC stream is used by the server to send updates about the game state to each client.

This architecture allows the system to recover from a variety of errors and crashes that may happen during operation. The main ones are crashes of the broker, of a server and of a client. If the broker crashes, then all servers notice the crash, as they keep heart-beating the service, and upon the restart of the broker, they reconnect and communicate

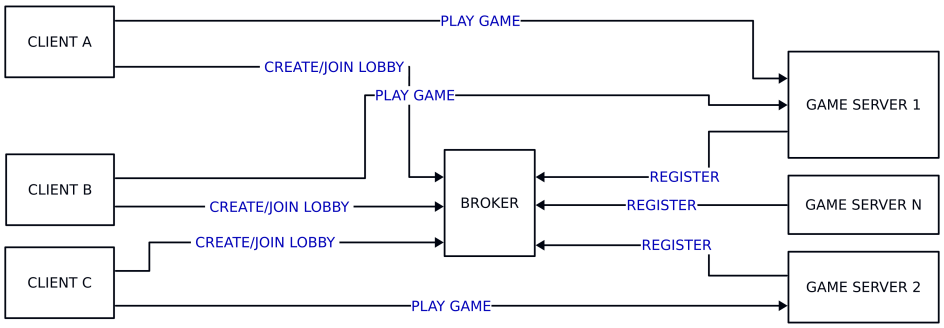

Figure 1: Co-Type architecture.

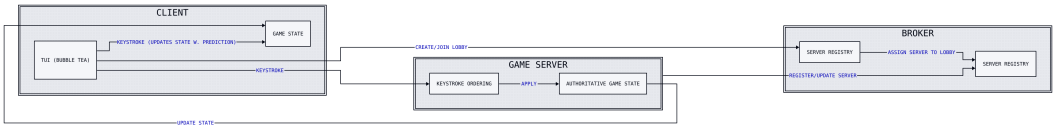

Figure 2: Co-Type infrastructure.

which lobbies are being managed. If a client crashes during a game, then the game is paused up until the client joins back. If a server crashes during a session, then the process is more complicated:

1. clients ask the broker to resume a game, and receive an available server;
2. each client contacts the server communicating the state of the game known by the client upon the crash;
3. after all clients have connected to the server, the server chooses the most up to date state of the game and resumes the session.

The processes used for error recovery are explained in section 3.5.

3.2 Infrastructure

The system infrastructure is composed of three elements that may be located anywhere geographically. A high level description of how they interact can be found in fig. 2.

The three components are broker, clients and servers, each one deployable independently of the others and with a distinct task:

- **Broker:** acts as a service discovery, but it's not limited to that. It allows servers to register to it by sharing their location (i.e. how a client can contact them) and keeps track of which and how many game sessions they are managing. When a client wants to create a lobby it finds the most suitable server to host it, and returns the location to the client. If the client want instead to join a lobby it finds the server hosting it, and returns the location information. It also has a central part in the server crash recovery process as described in section 3.5. It is deployed via a Kubernetes deployment with a single replica, as it is a stateful workload.
- **Clients:** each client is a TUI that allows the user to interact with the game, and is runnable via a single binary.
- **Servers:** alongside the broker allows the system to scale and manage errors. They manage game sessions, from the lobby state up until the end of a game and are also stateful entities. Upon startup a server contacts the broker to register itself in the server pool, so that it can be chosen to host games.

3.3 Modeling

Domain model The domain has been kept simple, in order to maintain focus on the error management and recovery. The main domain entities modeled in the system are:

- **lobby:** the domain entity used to manage games, identified by an *id*, it stores a list of **players**, alongside **game data** if the game has started and a *status*, which can be:
  - *waiting for players:* when a game still has not started;
  - *playing:* during the game phase;
  - *paused:* if the game is paused;
  - *game ended:* when the game session has ended.
- **players:** identified by a *name* is used to store the state of the player inside a lobby, whether it is *ready*, what *characters* he is able and/or unable to write along with if he can *delete* characters;
- **game data:** keeps track of the status of a game, including:
  - *snippet:* the code snippet that players have to write;
  - *correct characters:* how many characters have been written correctly;
  - *wrong characters:* how many wrong characters have been written, if any;
  - *revision:* an incremental value that is used to keep track of the latest changes when state reconstruction is needed;
  - *start time:* when the game has started;
  - *elapsed time:* server-side calculated elapsed time from game start to game finish, keeping track of pauses.

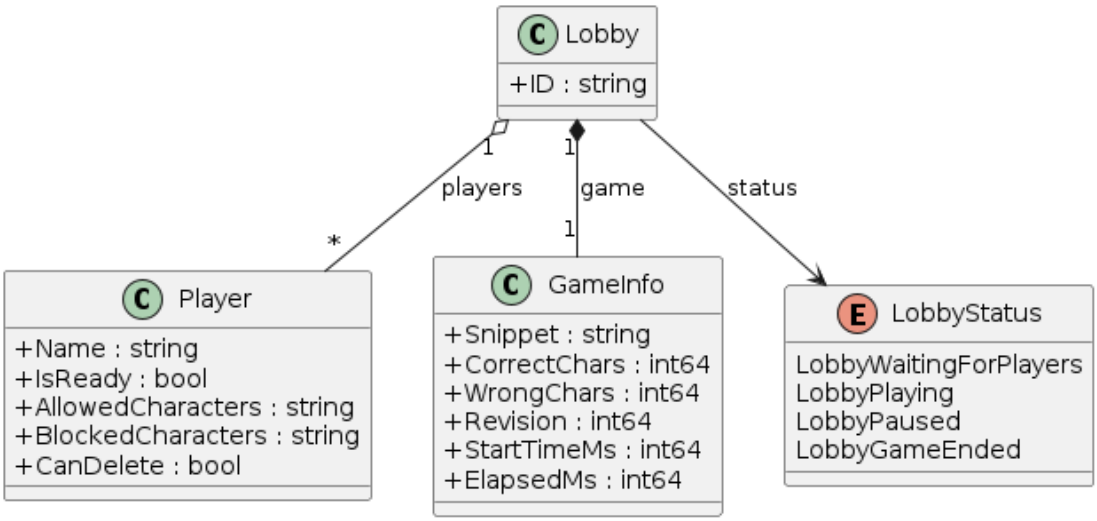

Figure 3: Co-Type domain model.

Domain model mapping Clients and servers need to keep track of all the entities of the domain model, as the game server needs to track a game lobby and its players, while the clients need that information in case of a crash of the server and the subsequent game state reconstruction. The broker instead is instead only interested in the identifiers of lobbies and players, and only needs to manage the entire state of a lobby when a server crashes and it is used to reconstruct the game state.

Messages Messages between the broker and the servers are always *request-response*, while between the client and the server, other than *request-response* also *server streaming* is used. This second interaction method is useful when a game starts, as it allows the server to broadcast updates to all the clients every time the game state changes.

3.4 Interaction

Components communicate via gRPC, the most interesting interactions regard lobby creation and join, actual gameplay and what happens when a server crashes.

Lobby creation and join When a user wants to create a lobby(fig. 4):

- the client communicates with the broker that he wants to create a lobby, the broker then either says that a lobby with that name already exists, or returns the URL of the server that has the lowest load;
- knowing the URL the client connects to the server saying that he wants to create the lobby;

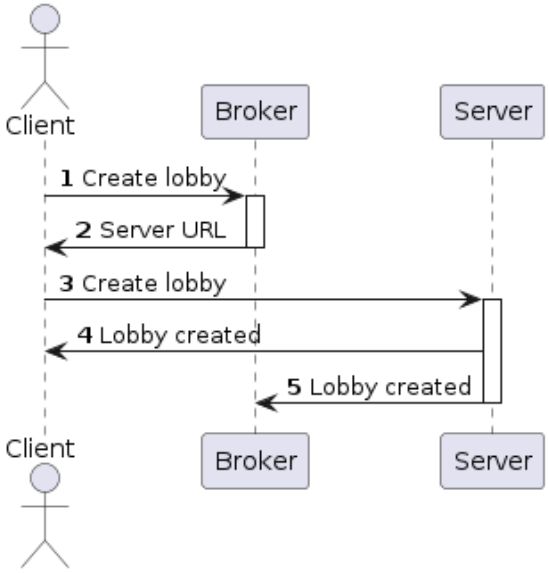

Figure 4: Lobby creation interactions.

- the server creates a lobby containing the user, and also tells the broker that he is managing that lobby, so that the broker can route other players to the correct server when joining the lobby.

When a user wants to join a lobby(fig. 5):

- the client communicates with the broker that he wants to join a lobby, the broker then either says that a lobby with that name does not exists, or returns the URL of the server that is managing the lobby;
- knowing the URL the client connects to the server saying that he wants to join the lobby.

Gameplay interaction During gameplay different clients interact with the server and receive updates on the state of the game via *gRPC server streaming*(fig. 6):

- a client ("Client A" in fig. 6) executes an action, updates its UI, and sends the action to the server;
- the server updates the state of the game, then sends the new state to each client;
- upon receiving the new state each client updates its own ("Client A" initial state prediction may have been wrong).

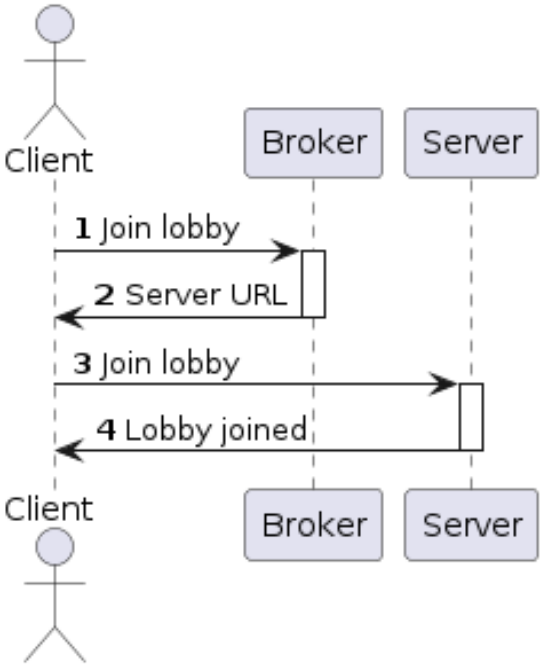

Figure 5: Lobby joining interactions.

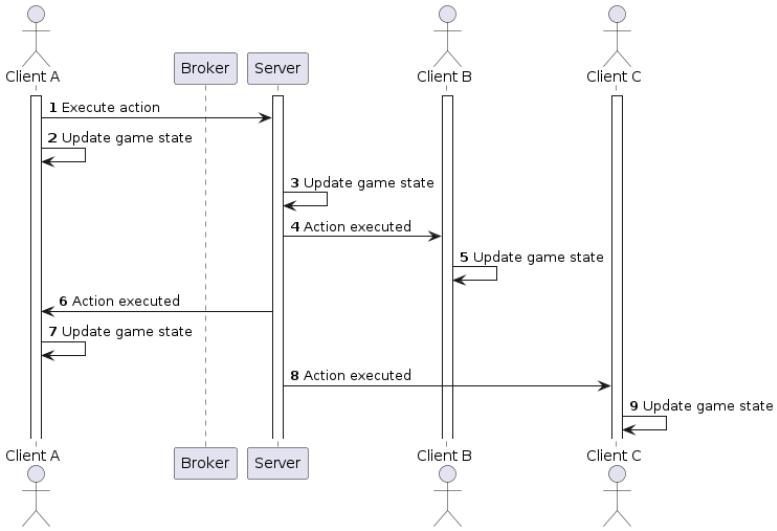

Figure 6: Gameplay interactions.

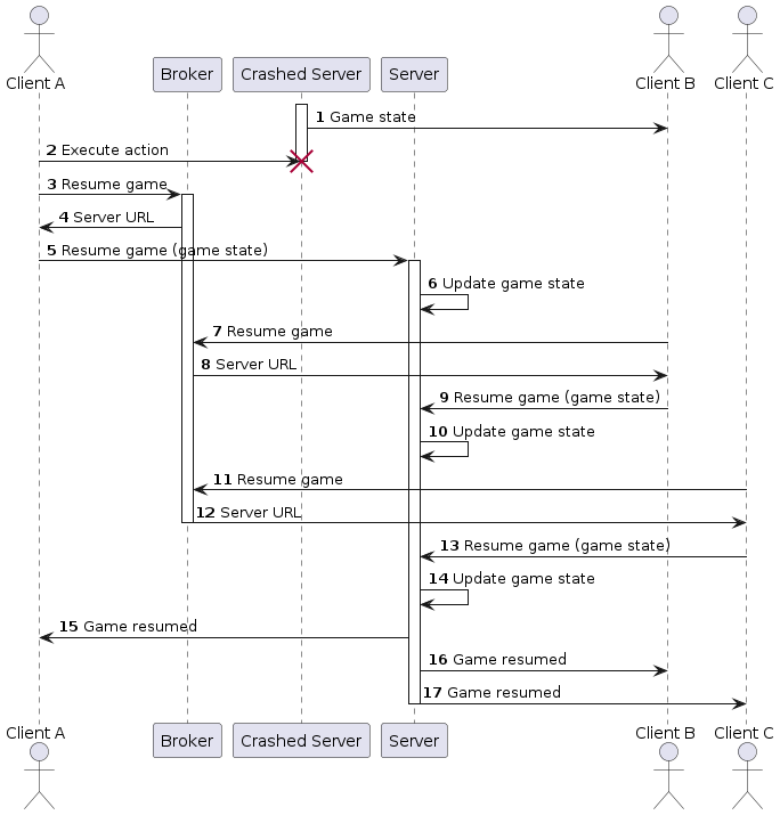

Figure 7: Server crashing and game state recovery.

Server crashes If a server crashes ("Crashed server" in fig. 7) then a game state recovery process takes place, before resuming gameplay:

- a client ("Client A" in fig. 7) is the first to notice the server crashing and communicates that to the broker, along with the users that were in the lobby, the broker sends the URL of a server;
- the broker responds with a server assigned to that lobby;
- meanwhile other clients may ask to resume the game to the broker, and are routed to the same server;
- each client sends the state he knows to the server;
- the server keeps track of which players still need to join, and if a newer state *revision* is received updates the state;
- when all players have joined the game resumes.

Other interactions Other interactions happen in the system:

- upon starting the server communicates its URL to the broker;
- every 10 seconds an heartbeat is sent from each server to the broker, other than as a way to look for broker crashes, the heartbeat message is also used to update the load of the system;
- if the server notices the broker crashing, then it tries sending the URL along with the load to it;
- when every player in a lobby exists it, or the game finishes, the server sends a message to inform about the session termination to the broker.

3.5 Fault-Tolerance

Fault tolerance in the system is achieved using **sharding**. Multiple server services are deployed and each lobby is managed by one and only one of them at any time. If one of the servers crash then clients notice, thanks to an open gRPC connection with the server used for server streaming, and move the game to a different server as shown in section 3.4.

Of the broker service there is instead always only one, while differently from the servers, designing it to be stateless and use persistent storage solution, and replication, would have been the simplest solution, the idea of the project was trying to have a fault tolerant system without using it. The health status of the broker is checked every 10 seconds by each server, which uses the heartbeat message to also update its information on the broker. If the broker crashes then every server notices and re-instantiates the connection.

If it is instead a client that crashes during game, then the server notices, via the gRPC server streaming connection, and pauses the game. When the client joins back then the game is resumed.

4 Implementation

Every component of the system has been implemented using **Go 1.26**, using **gRPC** for the communication between them. Go has been chosen for the great support for concurrency, both via *gorutines* and *channels*, since *go-grpc* manages *gorutines* implicitly, first class level support for concurrency helps avoiding mistakes.

Layered architecture Each component has been implemented following the layered architecture. This choice is the standard in Go services as it enhances testability of the system, without complicating the architecture unnecessarily. To improve code reuse across all services *go workspaces* have been used, allowing the domain, along with some utility functions to be located in a shared library.

Test Driven Design Each component has been implemented following a test driven design approach, with unit tests for each component. Following this approach allows for the verification of the correctness of the system both during development and refactoring. All tests are run automatically via *pre-commit* hooks, along with formatters and linters, to ensure that the code is always in a working state. All the checks also happen on *GitHub Actions* on every push to the repository, and on every pull request. *GitHub Actions* also manage the release of the system, with the delivery of docker images for each component to GitHub Container Registry.

4.1 Technological details

Networking All three components use *go-grpc* which has been chosen as it allows having a more strict structure in the interaction between the systems. This makes for simpler refactorings and better help by the compiler during development. While, for a game, using UDP would have made more sense, the choice to use gRPC has been made to learn what it offers with regard to other solutions seen during the course. Moreover, bi-directional and server streaming streaming initially seemed to fit the idea of the project, even if only server streaming has been used, as bi-directional streaming was not the correct choice. Using UDP with binary encoding would have made for the correct approach, and typically is the choice when creating games.

Concurrency Differently from how it could have been implemented using UDP directly, by using *go-grpc*, each request gets its own gorutine. For this reason it is necessary to manage concurrent access to shared resources, the repositories in this case. Each repository is designed with this implication in mind, and along with the actual entries that need to be stored in memory, it also uses a read write mutex. This allows either multiple readers or a single writer to access the data at any moment, making sure that no data is changed when another gorutine has access to it.

TUI The user interface has been designed to be extremely simple, as it was not the main focus of the project. Creating a GUI from zero would have required too much time dedicated to it, while with a CLI it would not have been possible to implement this kind of game. A TUI is for this reason the best compromise.

BubbleTea has been chosen as it is a mature project with pre made widgets that can be used while allowing for good customization. It uses an event loop to manage events, called messages, asynchronously. Every operation that may take time to complete, such as network requests, are wrapped inside a message so that they do not block the responsiveness of the UI. The UI is drawn via *immediate mode*, a string is returned by a `View` function and is exactly what is drawn on console.

5 Validation

The system is validated via automatic unit tests, and via manual acceptance tests.

5.1 Automatic Testing

Unit tests have been used to test the domain, services and repositories of the system, most importantly testing:

- lobby reconstruction, starting from the data each client has, tests are in place to ensure that the resulting game is created from the most recent snapshot;
- data races via the go compiler `-race` flag, to make sure that two gorutines do not access the same data concurrently, with one of the two having write access;
- server selection in the broker has also been tested to ensure that the correct server is chosen when a new lobby has to be created;
- the UI has been tested via unit tests, since each *BubbleTea* component simply returns a string when rendering, to ensure that the UI is consistent with the state of the game it has been tested.

5.2 Acceptance test

While error recovery inside each component has been tested via unit tests, the interaction between them is tested with *manual* tests, as creating the infrastructure to ensure that the correct server crashes and to check if everything still works would have taken too much development time.

Manual tests that have been performed during the development of the system are:

- lobby creation and join, ensuring that the lobby system works correctly;
- general gameplay, during the whole development process gameplay sessions have been played out ensuring that all features work as intended during normal working conditions;
- client crashes, ensuring that the game gets paused and only once the player has returned it is resumed;
- broker crashes, ensuring that servers notice the crash of the broker and send, once the broker is back up and running, their URL and which lobbies they are managing;
- server crashes, to ensure that the game transfer to another server happens without losing any data.

6 Deployment

In order to deploy the system some tools are required, the instructions that follow expect the following tools to be installed:

- make (optional)

- Go 1.26+[1];
- kind[2] (optional minikube, or any other cluster can be used);
- kubectl[3];
- Helm[4].

The following process expects kind to be used, but inside the repository an Helm chart is still provided so the system can be installed on any cluster.

6.1 Clone the repository

```
git clone https://github.com/sceredi/co-type.git
cd co-type
```

6.2 Create the kind cluster

```
kind create cluster --name cotype --config k8s/cluster.yaml
```
If you already own a cluster, skip to section 6.4.

6.3 Deployment

Using kind the deployment process is automated via `make`.
```
make deploy
```
Which checks if the kind cluster exists then deploys the system via Helm.

6.4 Manual deployment

If you want to deploy the system without using kind, if you are able to get the node ip, such as via `minikube ip --profile [profileName]` you can deploy the cluster via:
```
helm upgrade --install cotype-release ./k8s
--namespace cotype
--create-namespace
--set global.nodeIp="[node-ip]"
--set global.logLevel=INFO
```
If you instead are using an hosted kubernetes engine the process is a bit more involved, as it requires changing the chart so that instead of using `NodePort` services use `LoadBalancer`s. Particularly one for the broker and one for each server, then you also need to update the default values so that servers know their url.

[1] `https://go.dev/doc/install`
[2] `https://github.com/kubernetes-sigs/kind`
[3] `https://kubernetes.io/docs/tasks/tools/#kubectl`
[4] `https://helm.sh/docs/intro/install/`

6.5 Client configuration

After using one of the possible deployment methods, before running the clients it is necessary to configure them so that they know how to communicate with the broker.

If you ran `make deploy` you can now run `make addresses` and see the ip and port to use, something like:

```
Broker GatewayService :  172.19.0.3:30051
```

In this case you can simply run:

```
echo 'DISCOVERY_ADDR="172.19.0.3"\nDISCOVERY_PORT=30051\n' > client/.env
```

If you are instead something different from kind you have to use the "node ip" or URL that your kubernetes engine uses.

6.6 Running the clients

You can now run one or multiple (with multiple terminals) clients via:

```
cd client
go run cmd/app/main.go
```

7 User Guide

With the system deployed and the client configured you can now test the following scenarios.

7.1 Normal gameplay

- launch two clients;
- on the first create a lobby (fig. 8);
- on the second join the lobby;
- if you wish you can configure what each player can do (fig. 9);
- make both players be ready by pressing `r` on each TUI;
- complete the snippet, you can see that the cursor advances on both clients (fig. 10);
- both clients now show the game statistics (fig. 11).

7.2 Client crash

- Create and start a game as described before, during gameplay use `ctrl+c` to make one player crash;
- you can now see from the running client that the game has been paused;

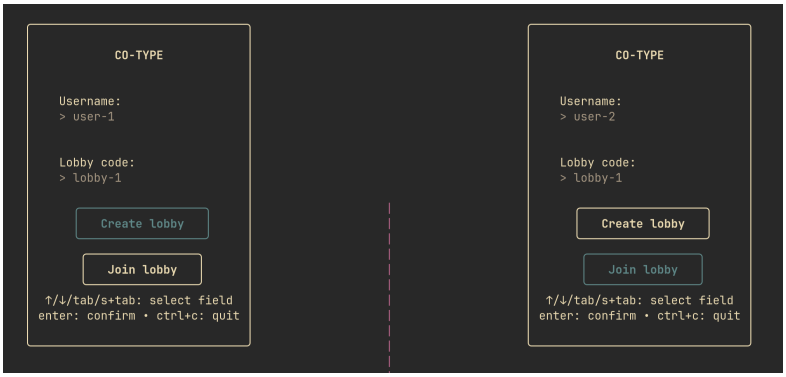

Figure 8: Side by side view of two players wanting to create and join the same lobby.

- open the client back, use the same name of the player you just made crash and join back the lobby;
- the game now resumes.

7.3 Broker crash

- Create a lobby with a player;
- make the broker crash by running `kubectl get pods -n cotype`, then `kubectl delete pod broker-xxxxxxxxxx-xxxxx -n cotype`;
- wait 15-20 seconds so that the pod has time to restart and servers to register on the broker;
- you can now join the lobby that was created before the crash from another client.

7.4 Server crash

- Create and start a game as described before, during gameplay use `kubectl delete pod gameserver-X -n cotype` to delete one of the servers, repeat the command for servers 0 to 2 so that all have crashed, meaning that surely also the one hosting the game has;
- switch back to the clients and you can expect in a few seconds to be able to continue gameplay as the game has been moved to a new server.

8 Self-evaluation

8.1 Simone Ceredi

Role Being an individual project I have designed, developed and tested every component of the system.

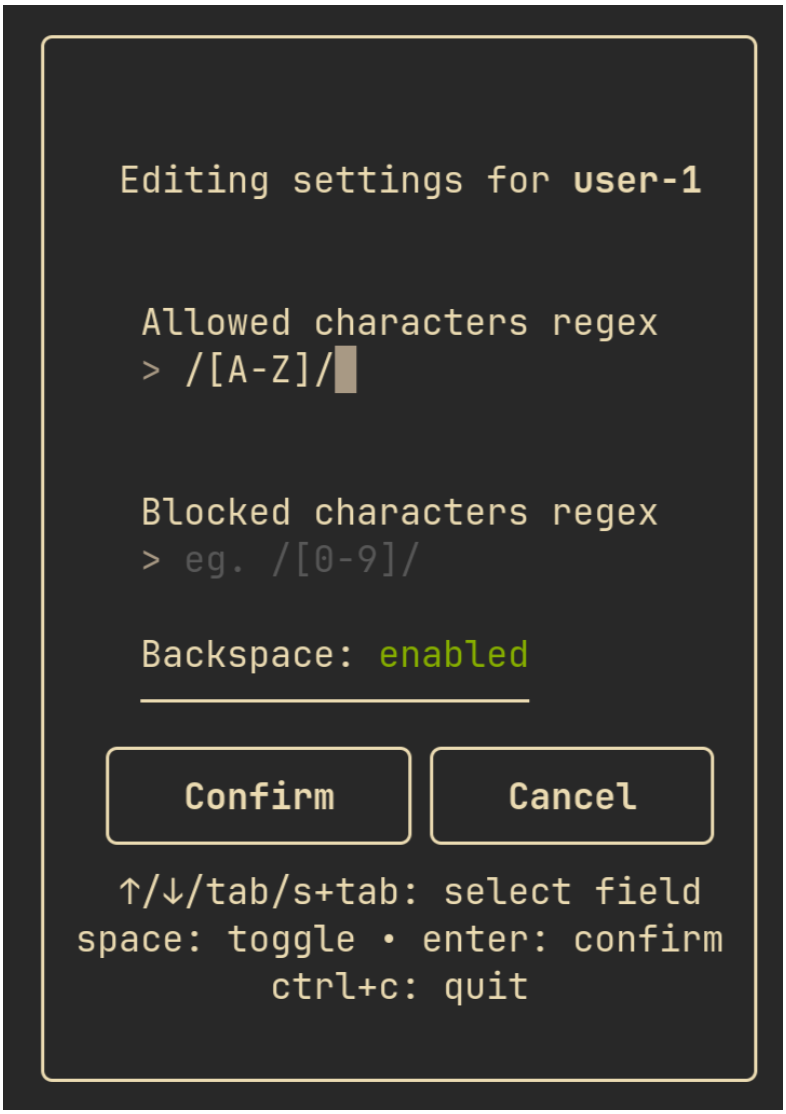

Figure 9: Settings view.

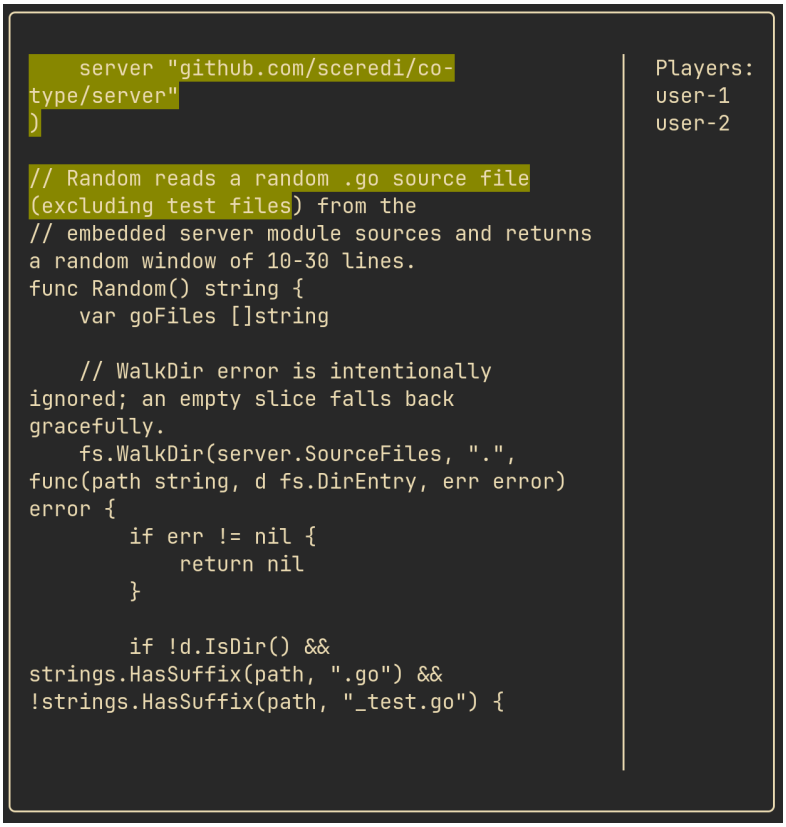

Figure 10: Co-Type gameplay view.

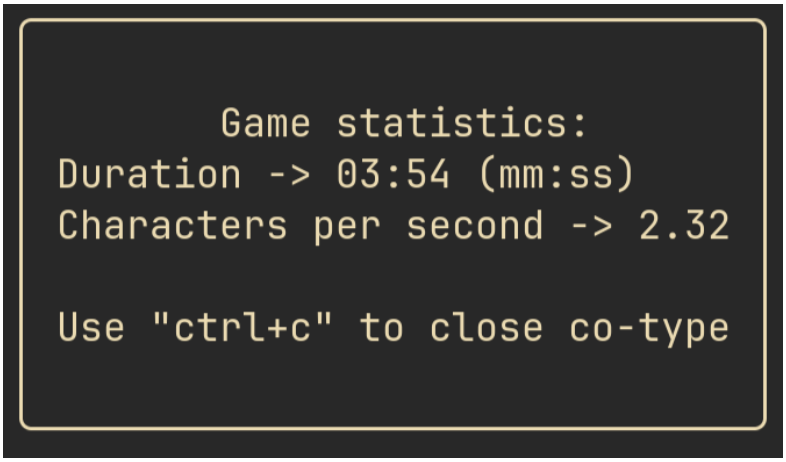

Figure 11: Co-Type game statistics view.

Deviation from the initial proposal I did deviate from the initial proposal mainly on game aspects, rather than architectural as the project was taking more time than planned to develop. The main change regarding the game is dropping the separate host/player roles in the lobby, which would have been an interesting topic to develop. Regarding more architectural aspects I did not use bi-directional gRPC streaming as I found that it would have made the code harder to understand and less maintainable. The idea was to use bi-directional streaming during the gameplay phase with players sending their keystrokes and receiving game-updates. I instead opted to use request-response for keystrokes and server streaming for updates coming from other clients.

Strengths

- **Fault-Tolerance:** the algorithms developed to manage crashes of all components without need to restart sessions and lose data and without the use of permanent storage has been an interesting challenge.
- **Kubernetes:** during the development process I discovered that using docker compose would have made the deployment of the system repetitive, as there is no equivalent to Kubernetes's StatefulSets, I have for this reason decided to study Kubernetes a bit to be able to deploy the system, discovering also Helm and other interesting tools in the meanwhile.
- **Monorepository structure:** every microservice project I have developed up to this point I have always chosen to use a multi repository code structure, but this time I wanted to try and use a monorepo and leverage go workspaces, and I have found some nice advantages having a commonly shared library, and some interesting challenges with the deployment process in the CI/CD pipeline.

Weaknesses

- **Verbosity:** I have always admired go for its simplicity and after learning it for a bit I wanted to try and develop a more interesting project with it that would leverage the language features. I have found myself having to write quite a lot of code to do simple things, prolonging the duration of the project and not allowing me to polish the project as much as I wanted.
- **Deployment:** my inexperience with Kubernetes at the moment I developed the Helm chart has led me to create a system that is only suited to my particular use case, with kind as a Kubernetes engine, unknowingly making the system not deployable on services such as AKS, EKS, GKE and so on.

9 Future works

Future developments would initially focus on polishing the server crash recovery algorithm, as depending on how servers crash, the clients may be connected to the new server but unable to resume gameplay. Other improvements would be:

- improving the Helm chart to make deployment easier on any type of cluster;
- improving the game with an actual GUI;
- auto generating the code snippets that are written during gameplay rather than reusing the server code itself.