weKittens: Exploding Kittens

Milan Lagae

Institution/University Name: Vrije Universiteit Brussel
Faculty: Sciences and Bioengineering Sciences
Course: Programming Distributed and Replicated Systems

Contents

1 Intro 1
2 Implementation 2
2.1 UI 2
2.2 Model 2
2.3 AmbientTalk 2
2.4 Lobbies 2
2.5 Game 2
2.6 Event Serialization 2
3 Design Choices 3
3.1 Lobby 3
3.2 Game 3
3.2.1 Disconnects 3
3.2.2 Reconnect 3
3.2.3 Exploding Kitten 3
3.2.4 Nope Card 3
3.2.5 Dead & Leaving 3
3.3 AT 3
3.3.1 Nope Card Time-Out 3
3.3.2 Offline Time-Out 4
4 Test Scenarios 4
4.1 Mock Network 4
4.2 Threads 4
4.3 Lobby 5
4.3.1 playerLimitLobby 5
4.4 Game 5
4.5 Video 5
5 Manual 5
5.1 Game 5
5.2 Tests 5
5.2.1 Java 5

1 Intro

This report will discuss an implementation for the assignment “weKittens: Exploding Kittens” for the course: Programming Distributed & Replicated Systems“.

First, the implementation itself will be discussed in section §2. Following the implementation, design choices will be discussed in section §3. Test scenarios in section §4. And to close, running the game, in section §5.

2 Implementation

This subsection will discuss the implementation of the application. At first, a general overview of the application will be given, using the image in Figure 1, as a guide.

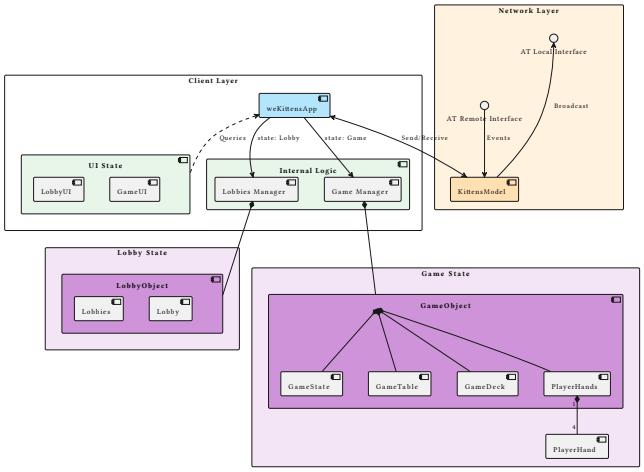

Figure 1: weKittens Component Diagram

There are more classes responsible for the workings of the application, but the more important ones are included in Figure 1.

2.1 UI

Depending on the state of the application, the interface of the application is different. For the lobby state, the lobbyUI will be displayed, for game state, it is gameUI. The current active UI object will query the state of the currently active logic object (lobbies or game).

It is the job of the weKittensApp to refresh the interface when a state change has occurred, so the interface displays it accordingly.

2.2 Model

The purpose of the KittensModel class is to be the bridge between the AmbientTalk logic & the Java logic. The class acts as the network controller.

All the messages sent over the AT interface, must first pass through the KittensModel class. All received messages from the AT remoteInterface are received by the respective method depending on the network state (lobby/game).

The received lobby/game-events are de-structured, based on the specified enum value (LobbyEventType/ GameEventType), and passed to the matching method. The method than applies the changes to the lobby or game object.

2.3 AmbientTalk

The localInterface receives the messages from the KittensModel class. A game event can be either send to a single player or broadcasted to all players currently in the game.

The lobby & game object have different methods for broadcasting & sending events. Broadcasting lobby events happen network wide, while game events are restricted to players present in the game. Single events are send to the player in question.

During the transition from lobby to a game, the list of id's of the player's are set in the AT object by the method: setGamePlayers.

2.4 Lobbies

The lobbies object maintains a list of all lobbies that are currently present on the network. A player will create a lobby, which will be reflected in the lobbies list of the other players active on the network.

If a player joins, the joining player get's a lobby object that is set to match the joined lobby. Other players on the network will receive the updated lobby information. More information about joining a lobby in §3.1. Each player is also able to leave an already joined lobby.

2.5 Game

The Game object contains the state & logic of the Exploding Kittens game. Included in this state are the following most important fields: GameDeck, GameTable, PlayerHands.

The GameDeck represents the list of cards from which players can draw a card. The state of the table is represented by the GameTable. It displays which card a user draws from the deck pile; which card a user played on the discard pile and the number of cards a user has remaining in its hand.

Keeping record of which cards each player has in its hand is maintained by the class PlayerHands. For each player in the game, a separate PlayerHand object is stored. Containing the list of cards the user currently holds and the state of the player (dead/disconnected).

2.6 Event Serialization

Building on the event system displayed during the practicums, each lobby/game event is a record that is serialized when passed over the AT network.

Each individual lobby & game event has a matching record object. Every defined record must implement the serializable class. For each event, the event

itself and a matching enum value (LobbyEventType / GameEventType) will be send.

On the receiving side the AT remote interface sends the event, to the KittensModel, which will cast the Record to its correct type based on the Enum value. This structure of passing events is similar for the lobby & game state of the application.

3 Design Choices

3.1 Lobby

The current implementation of the lobby system is a simple CRU (Create-Read-Update) system, no support for deleting a lobby at the moment.

On the creation of a lobby, the creator is set as the coordinator of the lobby. Each request for joining a lobby must be accepted by that specific player. This ensures a shared state of which players are included in the lobby. The game can also only be started by the initiating player.

Multiple games can be played on the network, by creating separate lobbies using the lobby system.

3.2 Game

This subsection will discuss design choices pertaining to the game.

3.2.1 Disconnects

Depending on the state of the game, if a player is detected to be offline and it is currently that player's turn, the game is paused for all players, user input is blocked. In the other case, the players's are allowed to continue playing, until it is that offline player's turn.

3.2.2 Reconnect

When a player reconnects after disconnecting from the network, the current player will send the updates value of the game to the reconnecting player.

On receiving the GameEvent, the player will replace the current game values with the new values. Currently the implementation is disabled, since the order of events is not guaranteed to be 'total', it is possible for the reconnect event to arrive before game event's send earlier.

Possible solution to this problem, is implementing a sort of order on the game events, or making the operations idempotent on the game state.

3.2.3 Exploding Kitten

If a player draws an exploding exploding kitten card and no defuse card is present in the players hand, the player is considered dead.

For the implementation, it was chosen to discard all the cards in the player's hand. The game than continues without the player in the game order, but is allowed to spectate the game. If the player wishes, he is able to exit the application.

3.2.4 Nope Card

When any card is played, the variable waitingForAck is set to true and for each player a timer is started, for more info on the AT implementation see §3.3.1.

Each time a response is received from a player, either passing or playing a nope card, the list of online player is retrieved. The id of the player is set to true, in the cardPlayedAck hash map. And the timer for the particular player is terminated.

Then the value of the card value is checked. If the card is not null, and the type of the played card is a nope card, the action of playing a nope card is activated, and last played card power is activated. The timers for all players are also cancelled.

Two boolean values are created: one for checking if all players have acknowledged, or if all online players have acknowledged. If that is the case, the power of the last card is activated.

3.2.5 Dead & Leaving

After a player is dead, he is still able to follow the game, but all his cards are placed on the discard pile, and he is unable to draw any more cards.

In both cases, the player is removed from the game order.

For the actions, where there is expected player action or application responses, these are handled by taking into account the 'online' player list.

When playing a favor/cat card, if the card is allowed to be played (accepted by all participating players), the player may select the player he wishes to request a card from. The list of selectable players is kept up to date by the players are that are actively participating and online.

3.3 AT

This subsection will discuss design choices made pertaining to the distributed part of the application in Ambienttalk.

3.3.1 Nope Card Time-Out

The description of the nope-card time-out system, can be seen in Figure 2.

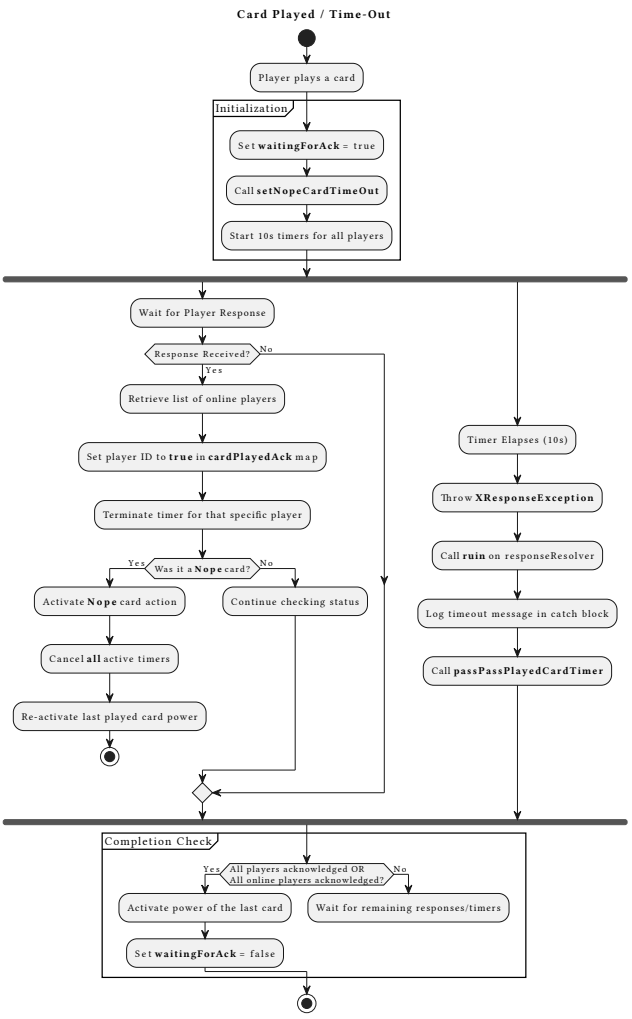

Figure 2: Card Played, nope card time-out

When a card is played, the AT method setNopeCardTimeOut is called with the list of players currently in the game. For each player a future is created, when the future is resolved, the accompanying offline timer is cancelled. When the future is ruined, the passPassPlayedCardTimer method on the Kittensmodel class is called, passing the played card in that players name.

The response timer is created, with a timing of 10 seconds (testing purposes). If the timer has elapsed, an exception is created named: XReponseException, with subtype: Exeception. The exeception is passed as a value to the ruin method on the reponseResolver. with a message indicating which player did not response in time. The message is than logged in the catch block of the future.

3.3.2 Offline Time-Out

When a player is detected to be offline, a future is created. At the same time, a timer is started. When the timer has elapsed, the future is ruined by creating an exception and calling the ruin(e) method on the resolver.

In case the user appears online again, the resolve() message is send to the resolver. This will resolve the future, and no future action will be taken. When the ruin message is send to the resolver, the catch block on the future is triggered.

The message in the exceptions is retrieved and logged. The id of the user is passed to the kickPlayer method defined on the model. The method on its part will handle kicking the player if he is part of the game.

4 Test Scenarios

4.1 Mock Network

For easier testing of the application, a mock network implementation was created to mimic behavior of the AT network as best as possible.

The mock implementation can be found in the: mock directory inside the test directory. The implementation consists of the following files: MockNetwork, MockInterface, MockLocalInterface, MockRemoteInterface.

When starting a test scenario, a mock network is created. Each application receives a MockInterface, consisting of a MockLocalInterface and MockRemoteInterface. The MockLocalInterface implements the atLocalInterface interface class, as best as possible. The MockRemoteInterface is responsible for passing the received messages to the KittensModel class as the AT implementation does.

The MockNetwork mimics the discovery of an actor when one is added to the network, the interface status is also propagated across the network.

Since the AT implementation expects events to be serialized across the network, the values passed through the mock network must also mimic this behavior. Copying the values passed over the network is handle by the deepCopyRecord function.

4.2 Threads

To prevent threading issues when executing actions, such as clicking buttons, selecting rows in a table, the code must be passed to awt.EventQueue. This ensures all actions are processed in the correct order and no other thread than the awt one performs UI actions.

Most calls to the awt.EventQueue are wrapped inside of a CompletableFuture, containing a delay depending on the action performed before. This allows the returning future to be delayed by calling join on the result. The main thread, running the network and application(s) can continue to work.

Once the time inside of the future has passed, the asserts are performed on the application values.

4.3 Lobby

The following test scenarios are declared for the lobbies:

1. createLobby
2. disconnectFromLobby
3. startGame
4. leaveLobby
5. playerLimitLobby

4.3.1 playerLimitLobby

This test scenario ensures the lobby system does not allow for more than 4 players to be present inside the lobby, and resulting game.

4.4 Game

The following test scenarios are currently included in game test files:

- TwoPlayerTests
  1. drawCards
  2. playerLeavesWins
- ThreePlayerTests
  1. playerLeaves
- FourPlayerTests
  1. drawCards
  2. playerLeaves

4.5 Video

Included in the zip folder, is a video detailing the workings of the application, according to the implementation described above and requirements in the assignment description.

5 Manual

The following subsection will describe on how to run the application in the different player configuration and how to execute the accompanying tests.

5.1 Game

The game can be started by creating 2 or more run configurations of the main.at file. Using the Jet-brains included compound functionality, a specific number of applications can be started.

5.2 Tests

Majority of the tests are created in Java using the Junit testing framework.

5.2.1 Java

Running the tests can be done by running any individual test. While running all tests simultaneously is a tested approach, they occasionally exhibit unexpected behavior.