Distribully (Distributed pesten)

Ronald Kruizinga (S2527227) & Ruben Scheedler (S2550709)

March 22, 2016

Contents

1 Introduction

A well-known Dutch cardgame is the game of *Pesten*. The goal of this game is to reach an empty hand of cards. The player that has the turn tries to play a card on the stack. A card is allowed when either the suit or the number of the played card corresponds with the card of the top of the stack. If the player has no playable cards in his hand, he draws a card. Some cards come with a special effect on the game, when played on the stack, for example:

- 2: the next player must draw two cards.
- 7: the player may play again.
- 8: the next player is skipped.
- Ace: the player order is reversed.
- Jack: the player may define the suit of the stack for the next card.
- Joker: the next player must draw five cards.

On top of these effects, the effects of the cards *2* and *joker* can stack during the game. That is, when player A plays a *2*, the next player (player B), may play another *bullying* card (a *2* or a *joker*), increasing the draw count to respectively 4 or 7. If player C can not play a bully card, he must draw an amount of cards equal to the draw count as it stands.

We will create a digital, multiplayer, version of this game with a few changes. First of all, there will not be one stack to play cards on, but there will be a stack corresponding with each player. When a player has the turn, he may not only choose what card he plays, but also on which stack he plays it.

The second modification we make is that effects/rules (we will call effects *rules* from here on) are not always connected to the same card. So reversing the playing order may be bound to a *six* instead of the ace, for example.
Finally, we let every player choose his own rule mapping. That is, every player assigns the six available rules to cards of his choosing. This gives the effect that at the first stages of the game, you might not know what the effect of the card is that you played on a stack, because every stack has different rules. During the game, you learn what rules are associated with which cards on certain stacks and you can adapt your strategy towards it. This adds a whole new dimension to an already challenging game.

2 Architecture Overview

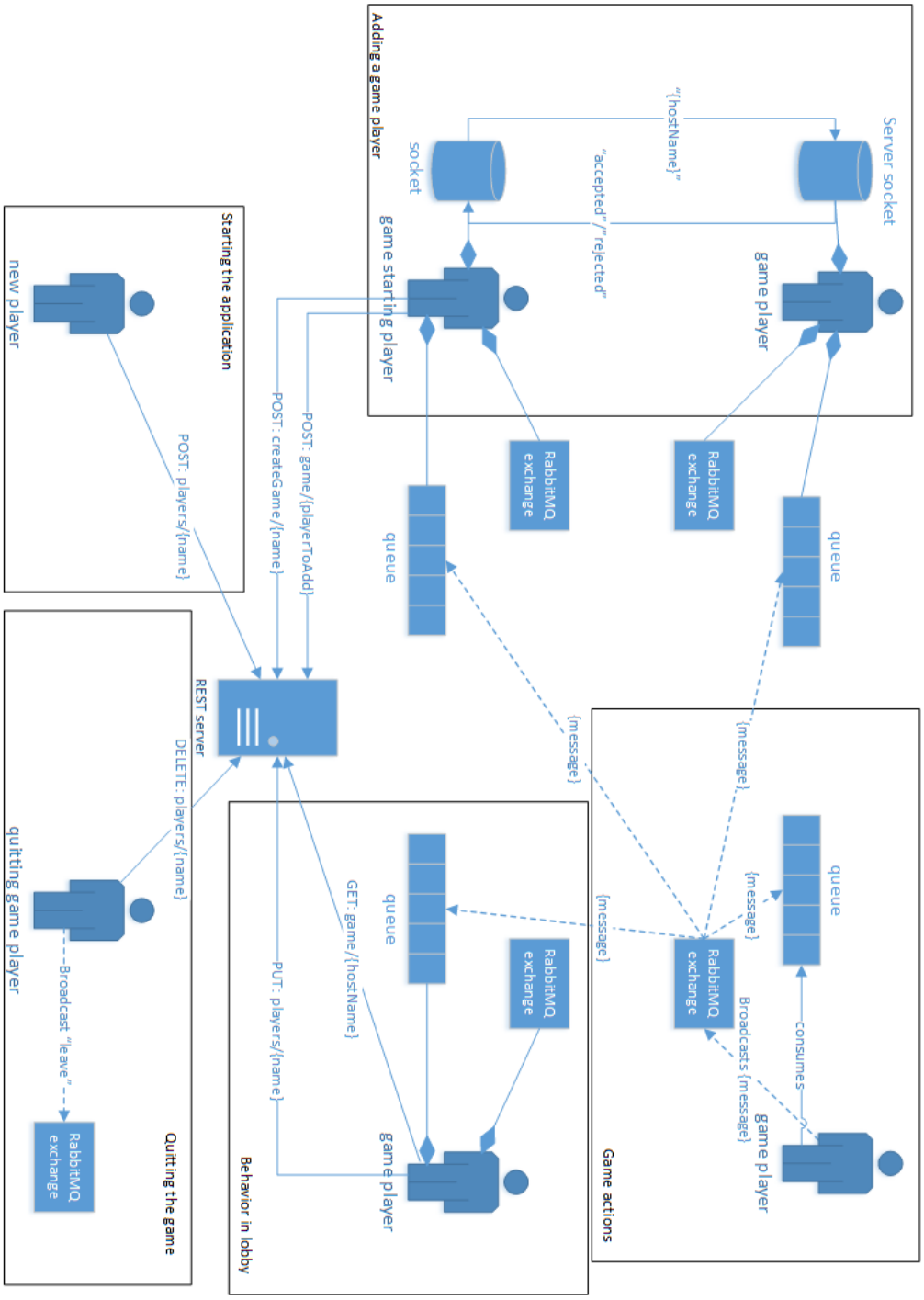

Figure 1: Overview of the architectural components used in the Distribully system.

As can be seen in the overview above, we use a variety of technologies throughout our application. As will be described under *Design Decisions*, the application can be in various states. Some of these states require the start of communication with other components. We split up the communication in three types and used a different technology for each type. We will briefly describe each implementation, referencing the above overview.

2.1 Global Player Overview (REST server)

We use a REST based webserver to host a global player "database". There are two sets of interactions with the server. Requests using the */players* URI are related to the general playerlist. This includes getting the list, deleting yourself from and adding yourself to it or updating your availability, using the corresponding request methods.

Requests using the */game*, */endGame* and */createGame* are all related to players in a single game. */endGame* and */createGame* either delete or create a new game, identified by the name of the host. We had to use seperate paths for this, since */game* also had an implementation of POST and DELETE requests. */game* requests add or remove players to a single game, again identified by the hostname.

2.2 Inviting (Sockets)

For inviting users, we used a websocket implementation. When not hosting or playing a game, the user has a thread (*WaitForInviteThread*) running that waits for a socketconnection to be made to it. A player hosting a game can select an available player and invite him to join his game. A new thread (*InviteThread*) is then started, that opens a socketconnection to the invited player over which the name of the host is sent. If that player still has an open socket and he accepts the invite, he is added to the lobby.

2.3 Gameplay (RabbitMQ)

For the actual gameplay of the game, we use a RabbitMQ based implementation. In this implementation, every user has a single queue. This queue is attached to the exchange of every user, including himself. Every user also has a single exchange he can broadcast to. This way he is never dependent on the exchange of a different host/ip, as he can use his own. While in the lobby, the user is only connected to the host (in the diagram denoted as *game starting player*). As soon as the host starts the game, the user connects to all the other exchanges. We decided to implement it this way since until the game starts, the users in the game are tracked using the REST server, since only the host can add new players.
We differentiate between message on the queue using the routingkey of that message, as that is an easy identifier. All communications over the queue are done using JSON.

3 Design Decisions

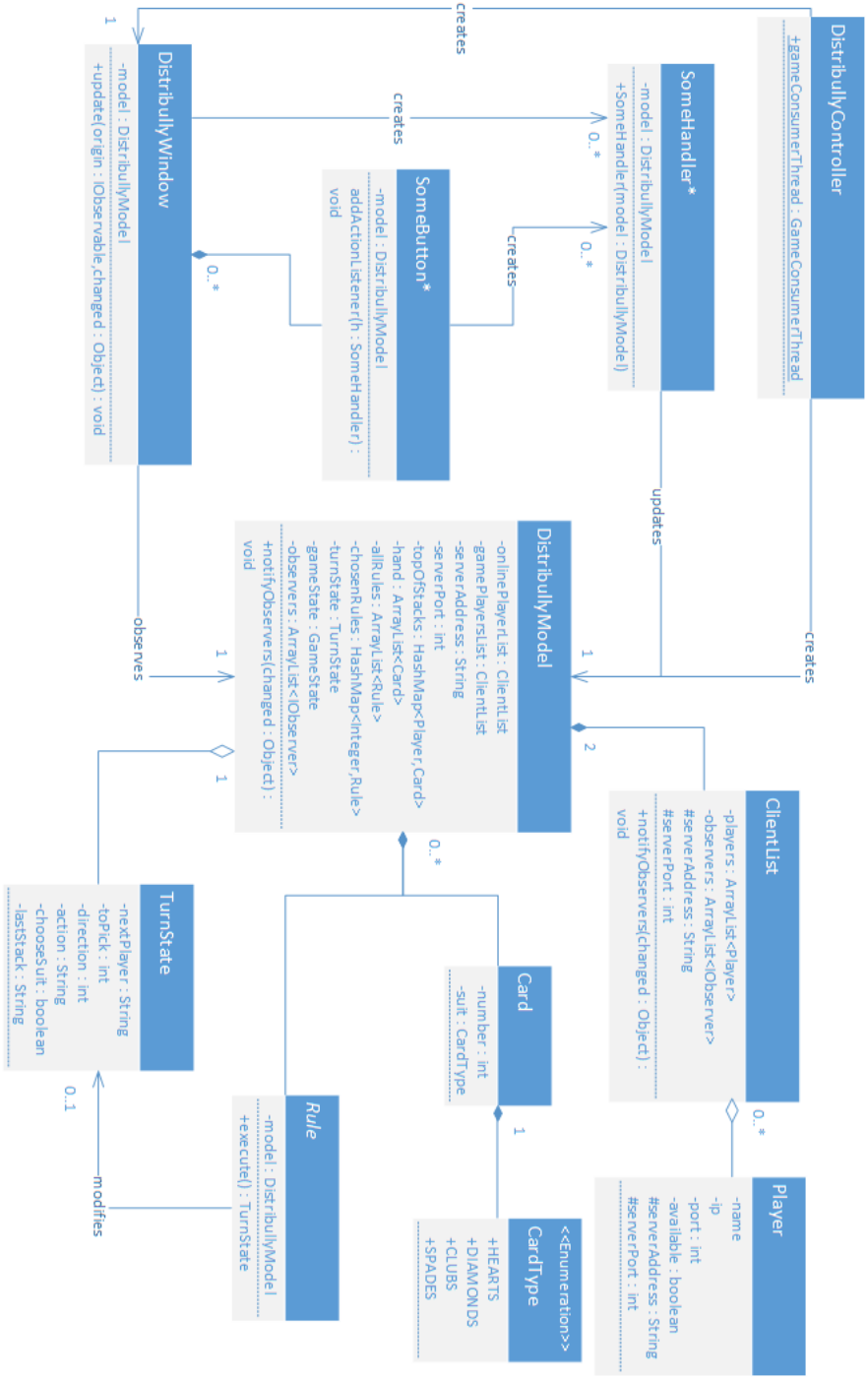

Figure 2: Overview of the classes that were used in the Distribully system, with their interactions.

* = Not an actual class, but rather a representation of a group of classes with similar responsibility.

3.1 MVC

We implemented this application using the MVC (Model View Controller) pattern. This is visible in our class diagram as well: *DistribullyController, DistribullyModel* and *DistribullyWindow* form the basis of the design, representing the three major components. The model contains the state of the application (and the game). It contains a list of all online players (which are registered at the REST server), a list of the game players (which are players of the game you are in), the hand of cards you are holding, etcetera. Another important part of the model is the *gameState*. This field determines what state the game is in. For

example: not playing, choosing rules, or playing.

The view of our program is *DistribullyWindow*. It contains a menu and a main content panel. The view observes the model and when it changes, the menu and the current content of the window are updated. For the menu, that means certain buttons are toggled according to the game state, for the window it means changing what panel to show. For most game states, a custom content panel was defined, that shows the relevant information from the model to the user. For example, a list of online players, dropdowns to assign rules, or the hand of the player.

3.2 Event Handling and Threads

The user interaction uses a similar flow throughout the application. The content panel contains buttons (denoted as *SomeButton*) in the class diagram. Each button, when clicked, generates an event and this event triggers a certain handler (denoted as *SomeHandler* in the class diagram). The handler performs certain actions, like creating messages and/or updates the model, which triggers an update of the view. We choose to implement the buttons and handlers in seperate classes, so that they become reusable and so that classes have clearly defined responsibilities.

During most game states, the user is waiting for an action of another player in the network. This waiting behavior is implemented in seperated threads, because communication mostly takes place asynchronously. The most important thread is denoted in the class diagram as well: *GameConsumerThread*. This thread is reponsible for consuming messages that are placed in the users queue and acting upon them. However, there are also threads for sending invites to players using sockets (*InviteThread*), awaiting an invitation over sockets (*WaitForInviteThread*) and keeping the lobby up to date(*LobbyThread*, which repeatedly sends a GET requests to the REST server to refresh the game player list). These threads were made part of the *DistribullyController*, so that all handler classes can start and stop the appropriate threads.

3.3 The Turnstate

When a game has actually begun, the *TurnState* component of the model becomes essential. It contains the state of the turn, that rotates through the players. Someone has the turn when the turnstate's *nextPlayer* field contains his username. It also shows the current draw amount (related to stacking of bullying cards) and whether the user is allowed to determine a suit for a stack (*chooseSuit*) and the last stack that was played on (*topStack*). When someone plays a card, a message is broadcasted, containing a *stackOwner* field. The player whose name is in that field sees a card was played on his stack, and the right *Rule* is executed. This rule modifies the turn state, for example by increasing *drawAmount* by 2 and the new turn state is broadcasted.

4 Possible Improvements

As in any distributed system, a lot of communication takes place throughout the network, using messages. Many messages are sent asynchronously and combined with multithreading, this yields problems. Waiting for the right moment to act was difficult in our system and it is very well possible that there are certain race conditions that we did not account for. The drawing/rendering of visual components also remains a difficulty that could be improved, given more time. Our Lobby often renders everything twice even though all components get removed. Decreasing the amount of repaints would improve the performance of the application, although the observer pattern enabled us to do this quite well already.

5 Possible Extensions

In the section above we mentioned race conditions and related to that would be the implementation of turn timers. Turn timers yield the advantage that the game can move on when a user lost his internet connection or when he just left his computer during a game. What also would be interesting to implement is displaying the amount of cards that each player has left. Finally, additional rules are, from a developer perspective, relatively easy to implement. All that is required is a modification of the turn state and registration in the model list of rules.