Prší

Also known as Mau-Mau or Pony

DOCUMENTATION & TECHNICAL OVERVIEW
ONLINE VERSION LINK

16.05.2021

Managing the Server

1 Starting up the server

Server is contained within a headless server.jar file, which is to be opened from a console.
Upon opening, the server should be ready to send and receive all requests from local and remote clients (Default port 8002).

1.1 Server Properties file

Server loads and stores some of its properties into a **server.properties** file situated next to the executable. In case no file is present on server start, a new one is created automatically containing all necessary properties.

Example of server.properties file contents:

```
{"PORT":8002,"displayName":"Generic prsi
server","autoStartNewGame":false,"displayRawCommunication":true}
```

1.2 Connecting from outside local network

For the server to be accessible from outside of your network, you need to set up your router to **port-forward** (Default **8002**) to your computer's **IP address**.
If you are running another server on the same **PORT**, the game will throw an **"Address already in use: bind"** exception. To resolve this, close the conflicting application and restart the game server. The game will by default output your public IP address on start. Clients will connect to this IP when outside of your network.

1.3 Windows CMD Color Display Issues

On windows platforms prior to newer Windows 10 versions the CMD does not allow for colorized output. Run the jar file with **`-noColor`** argument or directly set **`enableColoredOutput`** in server.properties property to `false` to remove ANSI color codes or use a different console like PowerShell.
Some Windows 10 systems have colored output disabled by default. Running the included batch file **server.cmd** as admin **will override the registry setting**, enabling colored output on CMD consoles.

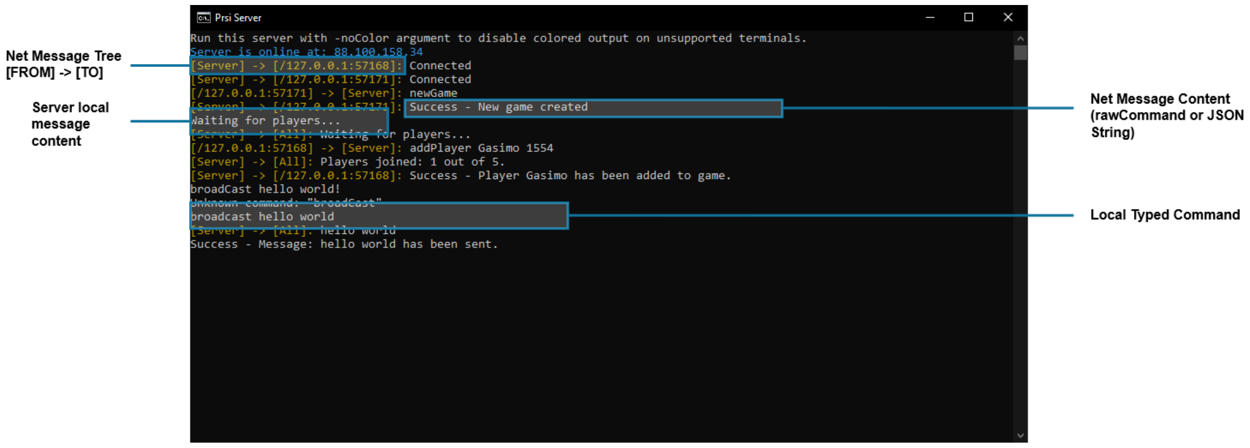
1.4 Using custom PORT number

In case you require a different PORT than default 8002, you can define your port in server.properties.

1.5 Debugging traffic

To review outgoing json traffic instead of just rawCommand, you can enable `displayRawCommunication` property in server.properties file

1.6 Interface


Prsi Server
Net Message Tree [FROM] -> [TO]
Server local message content
Net Message Content (rawCommand or JSON String)
Local Typed Command

```
Run this server with -noColor argument to disable colored output on unsupported terminals.
Server is online at: 88.100.158.34
[Server] -> [/127.0.0.1:57168]: Connected
[Server] -> [/127.0.0.1:57171]: Connected
[/127.0.0.1:57171] -> [Server]: newGame
[Server] -> [/127.0.0.1:57171]: Success - New game created
Waiting for players...
[Server] -> [All]: Waiting for players...
[/127.0.0.1:57168] -> [Server]: addPlayer Gasimo 1554
[Server] -> [All]: Players joined: 1 out of 5.
[Server] -> [/127.0.0.1:57168]: Success - Player Gasimo has been added to game.
broadCast hello world!
Unknown command: "broadCast"
broadcast hello world
[Server] -> [All]: hello world
Success - Message: hello world has been sent.
```

Terminal interface may differ based on your system.

2 Creating and starting a game session

Before the players can join, we need to **create or load** a game session using the "**`newGame`**" command. This will create a new **game lobby** so that the players can join.

2.1 Adding players

Players should always join by calling the **"`addPlayer`"** command **from their respective client instances**. Controlling more players from a single instance is possible but not supported as some breaking changes to the client may surface alongside development.
Be advised that calling **"`addPlayer`"** from the server-side console will create a player with sessionID 0. **We cannot play as this player using the server console directly**. If you wish to assign a client, call the same command on a client and sessionID will be assigned automatically.

2.2 Skipping waiting for players

The game will automatically start when 5 players join in. We can skip waiting for players by executing the **"`forceStart`"** command.

2.3 What happens afterwards

The game will **shuffle** all cards in deck and **give 4 cards to each player**. Then the remaining **top card** of the deck will be picked and transferred to the "top". The game will then assign all necessary gameplay variables and proceed to the **main game loop**. **More technical information will be available in the Technical Overview.**

Server commands

Command-Breaking Characters

Do not use any of the following characters in your commands as they are used for internal serialization and network malformation detection during merged requests: `[]`

[Following commands are executed server-side]

Command	**Arguments**	**Description**
`newGame`		Creates a new game session
`gameStatus`		Prints game status
`forceStart`		**Skips waiting for players** and force starts the game lobby under the condition that at least 2 players are connected and a game session is present.
`addPlayer`	`playerName, playerSecret` **playerName:** The name of the player **playerSecret:** Player password used for authentication	**Adds a player** to the game session as long as the game is in a state which allows it. Can be also used to re-login and reassign **sessionID** in case a player disconnected mid-game. This command should be only called **once by each client instance**.
`getPlayer`		*Deprecated*
`makeTurn`	`Int option`	Called by a player on turn to select valid turn action(s).
`broadcast`	`Message` **Message:** The message to be sent to clients	Sends a message to all connected clients.
`echo`	`Message` **Message:** The message to be printed to console	Used to **print** messages into the console. When called by the server on a client, this will be printed on a client.

Using the Client

1 Starting up and connecting to game lobby

Currently the client is only operating on text I/O. An additional GUI interface may be added at a later date. Keep in mind the current text-controls are there for **prototyping** rather than active playing and will be replaced in the future with more user-friendly messages. Open the **client.cmd** or **client.jar** using the console to begin. Before executing any commands, please read the Commands section.

1.1 Connecting to a game server

The game server manages and distributes the entire game loop. Therefore it's important to have one set up as defined in **Server Control Section 1**. To connect to a local server, execute the **"`connect`"** command. To connect to a server outside of your local network, add an IP address next to the command (example: "**`connect 127.0.0.1`**").
Client will welcome you with an "Connected." If you are having issues connecting in, make sure the server is operating on the same port as the client (default 8002) and port forwarding is properly set up.

1.2 Adding yourself into the game

Use the "**`addPlayer`**" command appended by your **username** and **password** to add yourself into a game session (example: "addPlayer Prokop password1234").

1.3 Connecting to an existing game

In case you disconnected mid-game, you can still connect back and control your player by executing the "**`addPlayer`**" command with your name and password you used prior to disconnecting. The server will automatically assign the player your network **sessionID** and you will be in control again.

2 Playing the game

2.1 The basics

Once the game starts, you will be informed about the status of the game. That being which cards do you have, how many cards do other players possess and who is on turn. In case you are on turn, you will be presented with a list of legal actions you can take in a given scenario. These actions are:
`PICK / PICK (Number of Cards), SKIP, PLACE or CHANGE_COLOR`
To select an action, use the **`makeTurn`** command appended by the index number of the action you want to make (e.g. `makeTurn 2`).
Once a player wins (by getting rid of all their cards), the game ends unless there are at least **2 players remaining**. The game will end once **only one player with cards remains**.

NOTE: This game does not implement the rule of returning players back in game using any card. The rule may be added as a selectable feature in the future.

Technical Overview

1 Overview

1.1 Goal

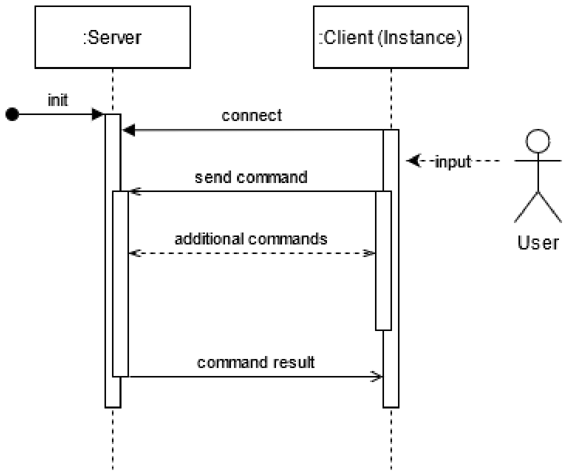
Create a playable game of Prší using Java with networking functions (Server, Client) with the ability to be modularly extended, maintained and re-used in other applications (Game Engines, Websites, Deep Learning etc).

1.2 Server Components

Server is responsible for all **game logic** and is made of 4 critical points: **Networking**, **Command Interpretation**, **Game Management** and an **Initialization & Reference** container. Deep dive into these components is done in 2 In-Depth Look at the components part of the overview.

1.3 Server Command lifecycle

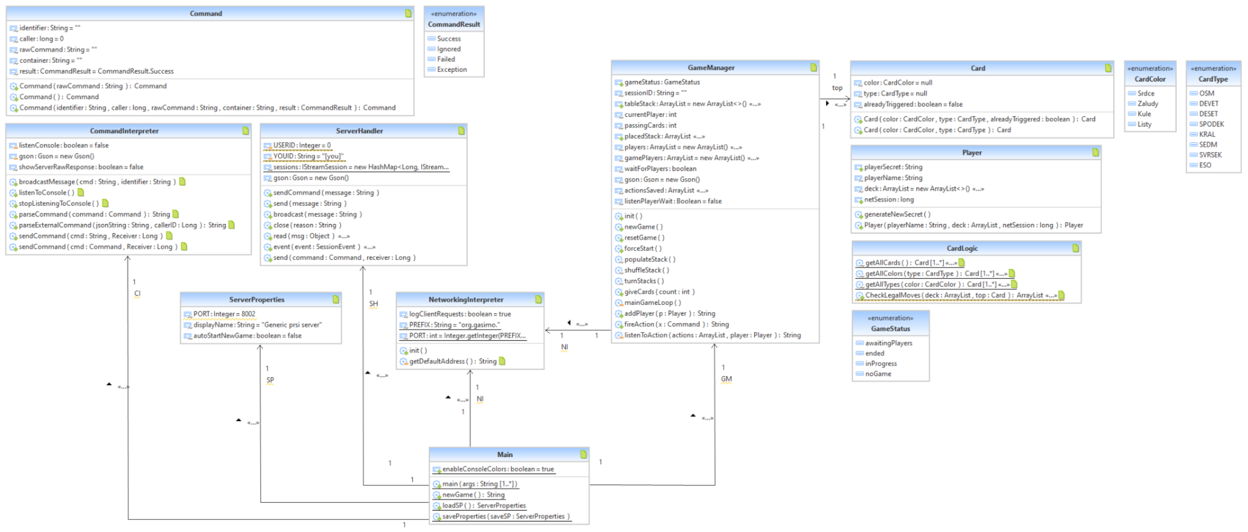
Server logic operates on a simple system. It executes commands and multiple threads and retrieves results. An example of an ongoing lifecycle is shown on the diagram on the left. **Client connects to a server and then proceeds to send a command.**
For this command to be completed, the server must ask the client to execute a few commands as well (*additional commands* arrow) Once these get executed, our command has been finally finished and the result is returned. More in depth look at command structure can be found in Command Framework and Implementation.



1.4 Compiling

This project uses Maven dependencies, make sure to open it with a supporting IDE or with all dependencies already implemented.

2 In-Depth look at the components



Class Diagram of important server classes

2.1 Initialization and Reference

Initialization of the server and its subsystems is handled and referenced within Main class in this order:

1. Create CommandInterpreter **(CI)**
2. Create ServerHandler **(SH)**
3. Check, Load or Create Server Properties **(SP)**
4. Create and initialize GameManager **(GM)**
5. Create NetworkingInterpreter **(NI)**
6. Listen to the console on a separate thread using `CI.listenToConsole();`.
7. Initialize **NI** -> Begin network session

2.2 Networking

Networking actions are handled in **ServerHandler (SH)** and **NetworkingManager (NI)** using **snf4j** library.

SessionID
Every client who connects is provided an **unique sessionID** which is later assigned to its corresponding players. This provides the ability to not require password for every player command and the ability to message a specific player directly (e.g. turn actions and other player-exclusive information.) By having sessionID separable from the base player, we can re-assign and move sessionIDs between players in case a client's session timed-out and got its sessionID regenerated.

NetworkingInterpreter
Takes care of main networking operation logic and provides data for **ServerHander**.

ServerHandler
Manages networking events like **send** (to), **broadcast** (to all) or **receive** (from any). All received bytes are transcoded into String and relayed to **CommandInterpreter**.

2.3 Command Framework and Interpretation

Commands
Every action between server and client is parsed using Gson into an JSON Command[] Array. Commands are intermediate objects we parse from and into JSON during networking communication and execute upon receiving.
Command objects include fields like **SessionID** (assigned dynamically), **rawCommand** (text form of the command to be executed, list of server commands available here), **Container** (Place we can place additional **abstract** json objects into to make deserialization easier) and similar as documented in JavaDoc.

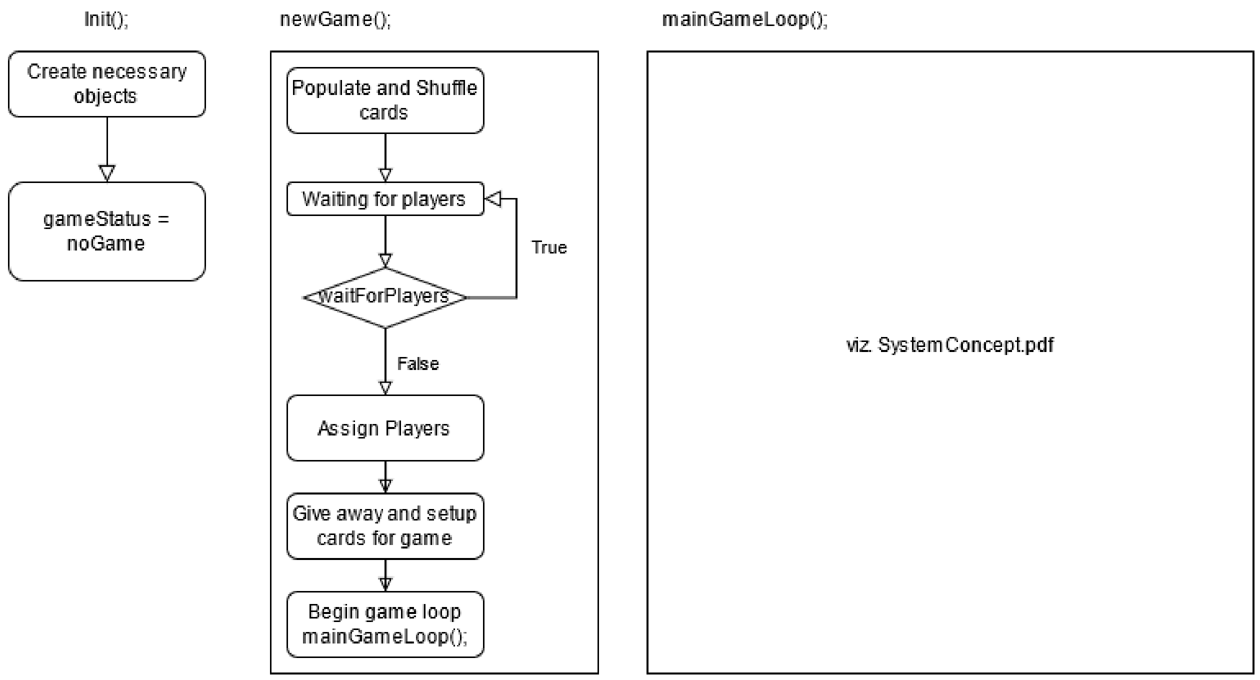
CommandInterpreter
Command Interpreter **manages** and oversees **deserialization** and **execution** of all commands. Based on the rawCommand attribute we switch between cases for each command and return the execution result to the caller.
Due to the nature of networking, it can occur that we may receive **merged** and **malformed strings**. My solution is that each time we receive a json Command[] string, we do a **checksum** checking whether all square brackets **"[ ]"** are properly **enclosed**. We know we are at the end of one command object by enclosing the first occurrence of "[". If another one follows, we know to expect another Command object, and that the request was merged presumably due to **slow connection or instability**. However this approach invalidates the use of singular square brackets in json string objects (e.g. player name).

Example of an parsed Command[] string:

```
[{"identifier":"","caller":0,"rawCommand":"echo Hello
World","container":"","result":"Success"}]
```

2.4 Game Management

The main game loop is contained within the GameManager object. The game goes through multiple stages as described below:



Simplified summary of mainGameLoop:

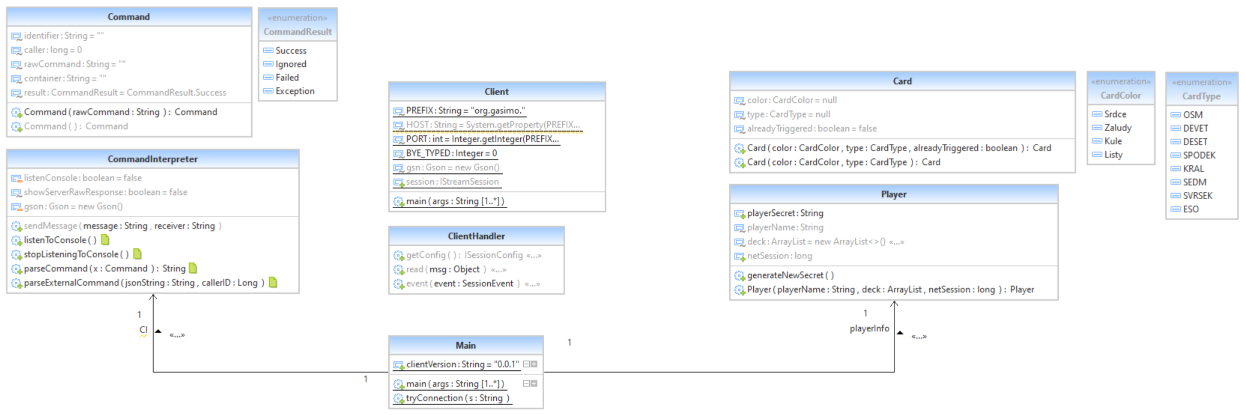
1. Game selects a player who is on turn (by taking index currentPlayer from gamePlayers list)
2. In case we are running out of cards, we must take cards from the placedStack and push them back into tableStack. In case this yields no satisfying result, we will have to force add an skip action into actions.
3. Game checks the current game and players deck to calculate all legal actions (enum ActionType) and writes them into an actions String Array.
4. Game sends info about the current game status to all players (including their deck cards and enemy card count and status).
5. Game sends a list of available Turn actions to a player (client) with a reqTurn (Require Turn) command prefix and awaits a valid response.
6. Upon receiving and confirming a valid response, the game executes the desired action. In case the player used a color-changing card, the game will ask the player again (using turnActions) to provide which color he wishes to change to. Upon receiving and confirmation we change the color.
7. If our player has no cards left, then the player won. We remove the player from the list of active players (gamePlayers) and shift the currentPlayer value accordingly. Then we notify the players about this. If there are at least 2 players left, the game will still continue with the rest of the players.
8. We select the next player by adding +1 to the currentPlayer. In case we have reached the boundary index, we reset the currentPlayer value to 0.
9. We go back to point 1.

3 Fulfillment of Assignment and Possible Improvements

3.1 Assignment: Graphical Input

Based on the assignment, the client was meant to have a graphical interface built on top of it. **However due to time constraints I have decided to invest more time into making the server foundation as stable and modular as possible rather than in the client interface,** resulting in fully text-based controls for rapid prototyping. The user interface can be easily integrated with or built upon using any standard Java UI library. It can also act as a foundation for future Prsi API.

Client Important Classes:



Command
identifier : String = ""
caller : long = 0
rawCommand : String = ""
container : String = ""
result : CommandResult = CommandResult.Success
Command (rawCommand : String) : Command
Command ( ) : Command
«enumeration»
CommandResult
Success
Ignored
Failed
Exception
CommandInterpreter
listenConsole : boolean = false
showServerRawResponse : boolean = false
gson : Gson = new Gson()
sendMessage (message : String , receiver : String)
listenToConsole ( )
stopListeningToConsole ( )
parseCommand (x : Command) : String
parseExternalCommand (jsonString : String , callerID : Long)
Client
PREFIX : String = "org.gasimo."
HOST : String = System.getProperty(PREFIX...
PORT : int = Integer.getInteger(PREFIX...
BYE_TYPED : Integer = 0
gsn : Gson = new Gson()
session : IStreamSession
main (args : String [1..*])
ClientHandler
getConfig ( ) : ISessionConfig «...»
read (msg : Object) «...»
event (event : SessionEvent) «...»
Card
color : CardColor = null
type : CardType = null
alreadyTriggered : boolean = false
Card (color : CardColor , type : CardType , alreadyTriggered : boolean) : Card
Card (color : CardColor , type : CardType) : Card
«enumeration»
CardColor
Srdce
Zaludy
Kule
Listy
«enumeration»
CardType
OSM
DEVET
DESET
SPODEK
KRAL
SEDM
SVRSEK
ESO
Player
playerSecret : String
playerName : String
deck : ArrayList = new ArrayList<>() «...»
netSession : long
generateNewSecret ( )
Player (playerName : String , deck : ArrayList , netSession : long) : Player
1
CI
«...»
1
playerInfo
«...»
Main
clientVersion : String = "0.0.1"
main (args : String [1..*])
tryConnection (s : String)
1
1

3.2 Extension: Computer AI Player

Doing Computer Game AI using the Monte Carlo method would be probably the most efficient but boring solution. Much more interesting would be to use Deep Learning technology to take advantage of the headless client nature to train a player model, possibly using TensorFlow or an ML alternative.

3.3 Assignment: Saving / Loading Game State

Due to lack of time, I have been unable to properly implement Saving and Loading of game states. If I were to finish it, I would have a **GameData** object to save all game-essential data to. This object would be then saved onto some save file on drive and loaded on server request. Players would join the existing game the same way as they rejoin disconnected sessions.

3.4 Improvement: Code Cleanup and Unification

- Some commands in the framework add json strings next to rawCommand. It would be much cleaner and more united to have all the additional json or message data in the command container field.
- The argument count requirement could be automated instead of hard-coding responses.
- More commands should take advantage of the Local/Remote execution distinction to prevent clients from executing broadcast, forceStart or newGame commands against other players' will.

- Migrate more commands and messages to take advantage of JSON and client info commands. Messages about players' card count or player on turn are results of raw messages (broadcast) rather than game status (rawCommand info field, container jsonData).