university of groningen / faculty of mathematics and natural sciences

Sport Performance Management

Technical Documentation

Joost Ouwerling

s2176378 j.t.ouwerling@student.rug.nl

Erik Bijl

s2581582 a.f.bijl@student.rug.nl

March 22, 2016

Net Computing 2015 - 2016
University of Groningen

All code and documentation is available at
`https://github.com/joostouwerling/netcomputing/`

1 Introduction

Professional sports is a matter of details. The difference between winning or losing can be extremely small. In order to maximise sport performance, data is necessary. A coach or manager wishes to have a lot of data how a athlete performs during activities. Data what could be collected can be speed, covered distance, positions and other location-based data.

A good abstraction of how a sports(wo)man performs is illustrated in a heat-map. A heat-map is a graphical representation where individual values contained in a matrix are represented as colors. A heat-map provides a great abstraction of where an athlete was during a game. Positions during matches can be highlighted and intensity can be increased according to their attendance on a position.

Our idea is to design a program which creates a heat-map during a match, in order to afterwards analyse these heatmaps. A player has the app on his/her phone which sends his locations to a program. A program receives this locations and extracts heatmaps from the locations. A coach can access these heatmaps and analyse the game of his players.

2 Architecture

There system consists of four components, which are all defined in their own packages. Figure 1 illustrates how the components are connected together.

Player

Players represent the sports(wo)men who have their location monitored during a match. In an ideal world, they would carry devices that send the location periodically to a server. We simulated this by creating an Android app that has the following functionalities:

- The user can select who he is, by providing a list of known players on the Webservice.
- The app shows a list of available matches, fetched from the Webservice, which show the match details. The user can click on a match to go to the detail view.
- When the user is in a match detail view, monitoring can be started. This will send the location of the player periodically to a DataServer provided by the match details.

DataServer

DataServers do two things. First of all, they listen for incoming location packets and store them in their local database. Secondly, they listen for requests, made by Clients (through the Webservice) for all locations, for a given player and match, and respond by sending the locations to the callback provided in the request.

Webservice

The Webservice provides endpoints for both Players and Clients to communicate to. It also has a connection with all DataServers to request locations when a Client asks for it. The functionalities which are implemented:

- Serves lists of all players and matches, retrieved from a central database and used by Clients and Players.

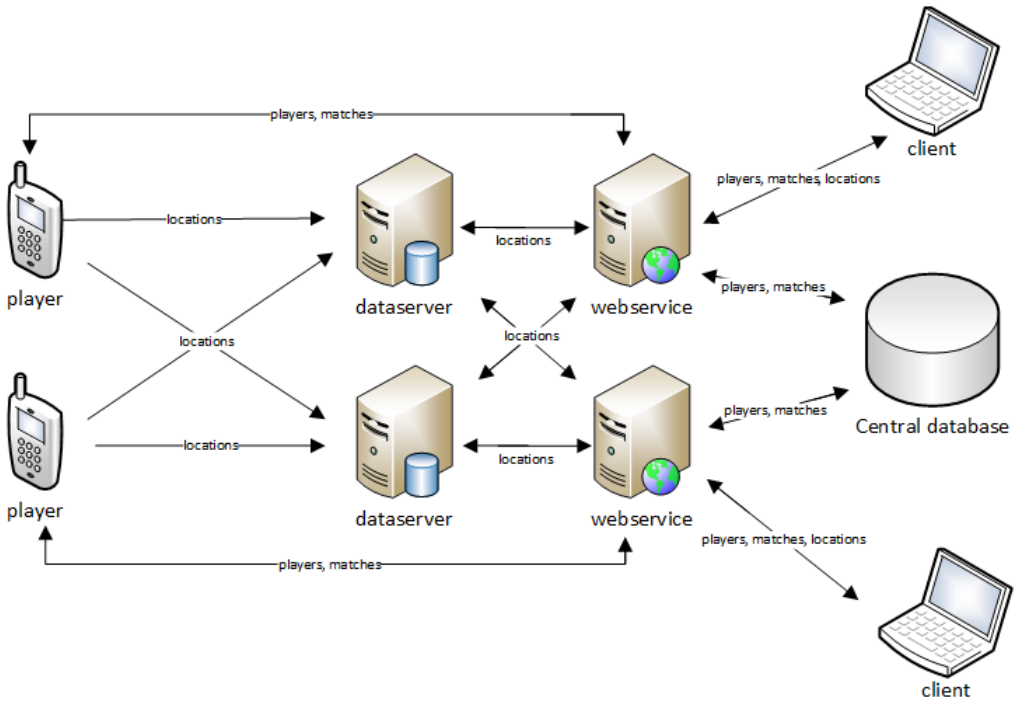

Figure 1: An overview of the architecture.

- Endpoints for creating new players and matches, which are consequently stored in a central database. These endpoints are used by Clients.
- Endpoint for Clients to request all locations for a given match and player. When such a request is received, the Webservice asks all DataServers if they have locations for this match/player. The Webservice sends the received locations back to the Client.

Client

The Client is mainly used by coaches and offers the following functionalities:

- Create new matches and players by providing forms and sending the input to the Webservice.
- Request all locations for a given match and player, by showing a list of all matches and players in the Webservice, which the client can select. The locations are requested from the Webservice (which in turn communicates with the DataServers).

3 Implementation

The main package for this application is `eu.sportperformancemanagement`. Every part of the architecture is defined in its own package. Since the client is purely web-based, it is not part of a Java package. There is also a `common` package, which contains code that is shared over multiple parts. This section includes some UML diagrams, but they are not always complete (e.g. getters and setters are not there). This would result in much superfluous information which is not necessary to understand the application from a technical point of view.

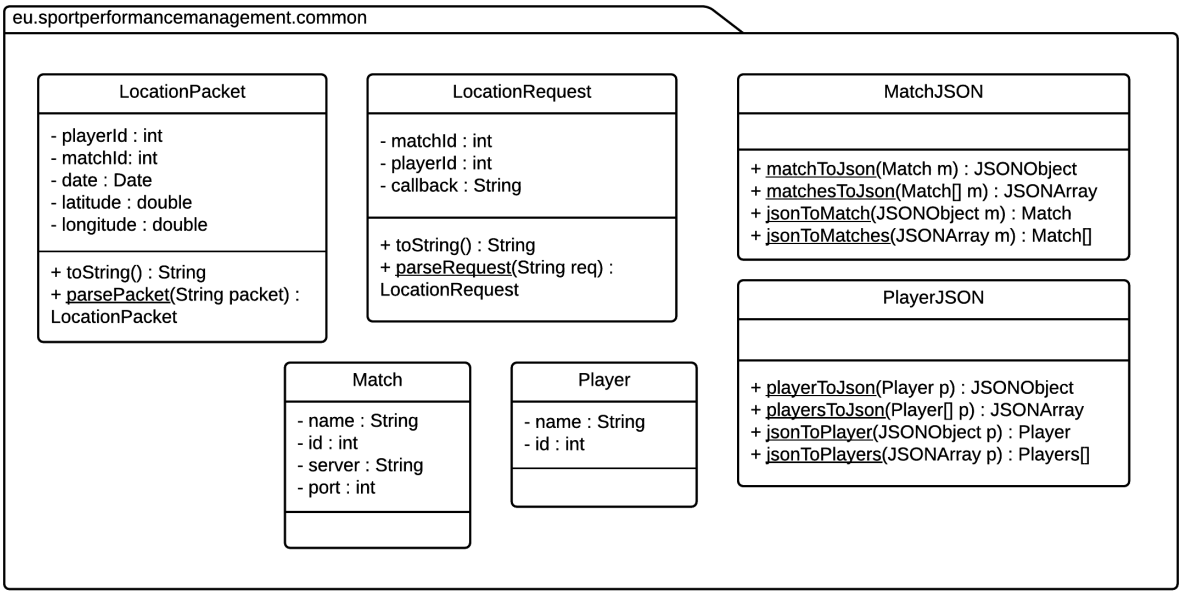

Figure 2: The UML diagram for the common package.

3.1 eu.sportperformancemanagement.common

The most important classes of the common package are shown in the UML diagram of figure 2. `LocationPacket` and `LocationRequest` objects are only used to communicate within the system, so they provide an easy way to marshall their information with almost no overhead. `Match` and `Player` classes both have a class which can make and parse JSON of (arrays of) these objects.

3.2 eu.sportperformancemanagement.player

As said in the Architecture section, the sports(wo)men are modelled using an Android app. There are three activities (i.e. "pages"): showing a list of matches, setting the player and the match detail activity. The last activity is used to start monitoring locations for a certain match. The match detail activity uses two classes to monitor locations and send location updates to the DataServers. `LocationMonitor` listens for locations, using the Google API client provided in the Android framework, and sends the location to `LocationSender`, which makes a `LocationPacket` and sends it over UDP to the DataServer connected to this match (figure 3).

3.3 eu.sportperformancemanagement.dataserver

The DataServer has two important classes. First of all, `LocationListener` listens for incoming `LocationPackets` on a UDP socket and uses `LocationDAO` to store them in their local database. `QueueListener` listens on a Rabbit MQ channel for incoming `LocationRequests`, indicated by a specific routing key as defined in `SpmConstants` in the common package. When this request comes in, it will use `LocationDAO` to fetch all locations for the given player and match and send them using a http POST request to the callback.

3.4 eu.sportperformancemanagement.webservice

The Webservice is mainly a Jersey REST webservice. There are three resources available. `MatchesResource` and `PlayersResource` both provide a GET and POST endpoint on respectively `/matches` and `/players` where the user can get a list of all matches/players and create a new match/player. To handle communication with the database, a `MatchDAO` and `PlayerDAO` class is used. The third resource is `LocationResource`. When a GET request is made to `/locations/<match_id>/<player_id>`, a `LocationRequest` is created with the given match and player id. A unique callback is also generated and added to the request. This `LocationRequest` is sent to a RabbitMQ fanout exchange using `RequestEmitter`. The DataServers who receive this request make a http POST to the callback with all locations they have for this match and player.

3.5 Client

The client consists of a set of webpages. The first webpage shows a form where the coach can select a match and a player. Then a request is sent to the Webservice. On return, the webpages makes a heatmap using the Google Maps Api: `https://developers.google.com/maps/documentation/javascript/examples/layer-heatmap`. Furthermore, there are two pages where the coach can create new players and matches using POST requests to the webservice.

3.6 Communication

All communication between components is depicted in a data flow diagram in figure 3.

Player - DataServer
: The player sends their location data to the DataServer over UDP sockets. Since a lot of locations can be sent, we choose for the faster UDP protocol. The data that is sent is a `LocationPacket` object from the common package. The `toString()` result is send, and this is parsed at the DataServer using `parsePacket`.

Player - Webservice
: Player uses the REST service defined at the Webservice to load the list of matches and players. For this, the REST endpoints `/matches` and `/locations` can be accessed with a GET request and can return the following results

- *200 - OK* with a JSON array of matches / players, made by `MatchJSON` (or Player).
- *500 - Internal Server Error* when loading the resources from the database failed.

Client - Webservice
: The client uses the same REST endpoints as defined in Player - Webservice for loading lists of players and matches. In addition, it uses the endpoints for creating new matches and players. These can be accessed by making a POST request to the same endpoints. For creating a match, the `name`, `server` and `port` field need to be provided. For players, only `name` is necessary. Both endpoints can return the following:

- *201 - Created* with a JSON representation of the created object.

- *409 - Conflict* when a resource with the given `name` already exists.
- *500 - Internal Server Error* when inserting the resources in the database failed.

In addition, the Client can request locations of a certain `matchId` and `playerId` at `/locations/matchId/playerId`. This can return the following:

- *200 - OK* with a JSON array of locations.
- *500 - Internal Server Error* when the locations could not be loaded.

Webservice - DataServer

When the Webservice receives a request for locations of a certain match and player, it will ask all DataServers if they can provide the locations they know. This is done using RabbitMQ. The Webservice sends a `LocationRequest` to the exchange, which uses a fanout with specific routing key to send the request to all DataServers, which are listening on the queue. When the DataServer receives a message with the specific routing key, they parse the `LocationRequest`, fetch the locations from the database and POST's them to the callback. The Webservice waits a predefined number of milliseconds before returning all locations it received on the POST callback.

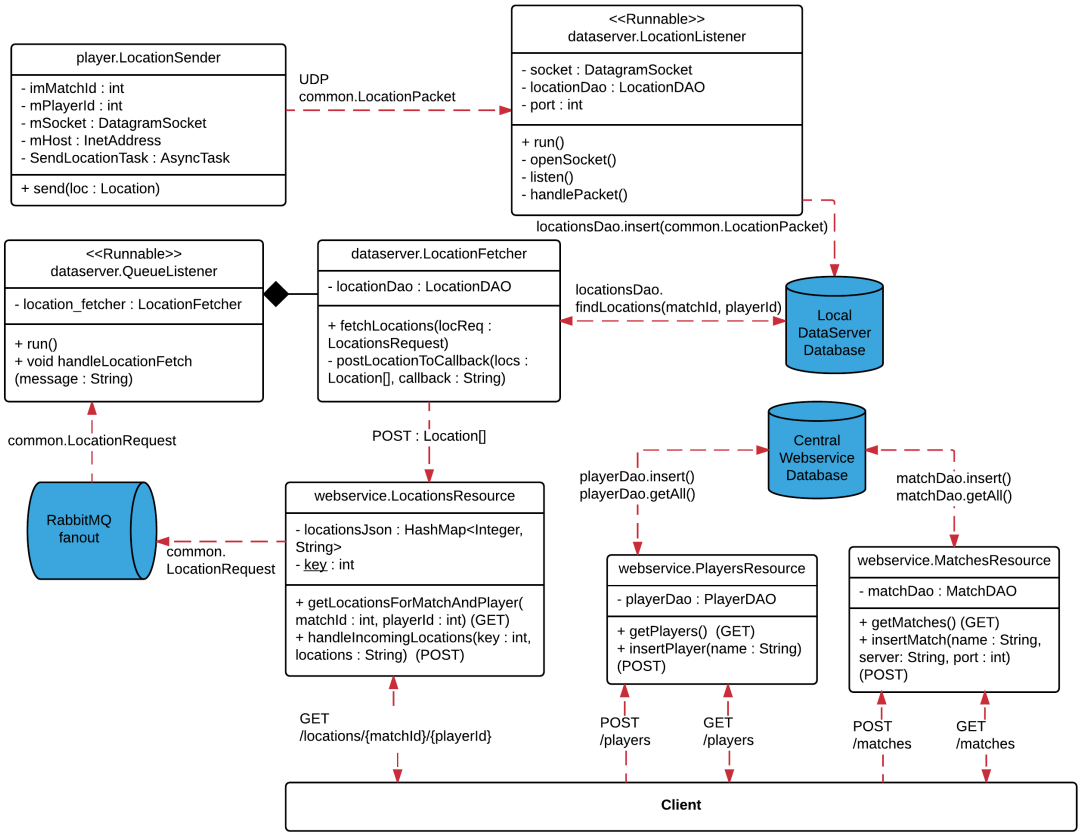

Figure 3: The data flow between different elements of the application. Note that the Player also communicates with the webserver to get players and matches.