Programming Life - Architectural Design

Group 5/E:
Felix Akkermans
Niels Doekemeijer
Thomas van Helden
Albert ten Napel
Jan Pieter Waagmeester

March 14, 2012

Contents

1 Introduction

1.1 Purpose of the System

1.2 Design Goals

1.3 Definitions, acronyms and abbreviations

1.4 References

1.5 Overview

2 Proposed software architecture

2.1 Overview

2.2 Subsystem Decomposition

(which sub-systems and dependencies are there between the sub-systems?)

2.2.1 Interface (API) of each sub-system

We propose the following API for the server-side of our application. Every model is represented by an object which stores the needed information. It is basically a directed graph with building blocks as nodes.

listFiles(): list of filenames - Returns a list of available models.
If there are no available files, then it returns an empty list.

getFile(fileName): model - Returns the BioBrick model stored in fileName.
If the file does not exist or if the file is corrupt, it will give an error.

putFile(fileName, model) - Stores model in a file called fileName.
If something goes wrong while saving, it will give an error.

listProteins(): list of proteins - Returns a list of proteins, including their meta-data.
With absence of information, it will give an error.

listModels(): list of models - Returns a list with a model for all available files.
Returns an empty list when there are no available files.

modelToSBML(model, fileName) - Exports model to an SBML file and saves it to a file called fileName.
Returns an error if model is not a valid model.
Returns an error if saving failed.

simulate(fileName, inputValues): outputValues - Simulates the SMBL model saved in fileName with given input values and returns the matching outputvalues.
Returns an error if the simulation of the file failed.

validate(model): boolean - Validates the model and returns true if model has everything defined to simulate it, and returns false otherwise.

For the client-side of our application, we will not follow a predefined API. As the client will only implement the view part of the model, view controller (MVC) system, it will only reuse the API of the server and not implement an API of its own. Messages between server and client will be passed using JSON objects.

2.3 Hardware/Software Mapping

Our architecture is composed of two major elements, the client implemented as a web application and the server, implemented in Java. The browser has to be a recent version of one of the major browser[1], the supporting system is not relevant. The server can be run on the same machine, which is likely to be the same machine during the development stage.

[1]We'll test in Google Chrome and Firefox

Figure 1: Examples of JSON responses

Generic response:

```
{
  "success": boolean,
  "message": string,
  "data": object
}
```

Response of getFiles():

```
{
  "success": true,
  "message": "",
  "data": [
    "example1.syn",
    "d-flipflop.syn",
    "xor-gate.syn"
  ]
}
```

Response for getFile(fileName) for a non-existing file:

```
{
  "success": false,
  "message": "Requested file not found.",
  "data": {}
}
```

2.4 Persistent Data Management

(file/ database, database design)

2.5 Global Resource Handling and Access Control for the different actors

2.6 Concurrency

(which processes run in parallel, how do they communicate, how are deadlocks prevented?)

2.7 Boundary Conditions

(how is the system started and stopped, what happens in case of a system crash)