Programming Life - Architectural Design

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

March 16, 2012

Contents

1 Introduction

1.1 Purpose of the System

The purpose of our system is to create a modeling environment for biology experts and people who work in the field of genetics. Within this modeling environment, you can easily model BioBricks. BioBricks are biological gates, which are created by gene manipulation. Our system should be able to create BioBricks with basic components, like AND and NOT gates. The system should also be able to import and export BioBricks, making it possible to create larger constructions with earlier designed BioBricks. This will allow the construction of more complex gates. From NOT and AND you can create OR and eventually implement flip-flops and other gates.

1.2 Design Goals

Our design goal for this project is to successfully implement the system mentioned above. We aim to make the system easy to use for people without too much knowledge of computers.

1.3 Definitions, acronyms and abbreviations

BioBrick is the key word in this project. A BioBrick is a logical gate representation of a manipulated cell. The idea behind this is: Genes can be manipulated to work as gates. By doing this we can create logical structures using cells. A BioBrick is a model for this. The term Bio comes from Biology. The term Brick comes from the idea that these BioBricks should be building blocks for a larger system. With Bricks you can build bigger structures.

Client-side is the part of the program which takes place on the computer of the client/user. The opposite is server-side. For our program, the client-side is a GUI implemented in JavaScript. These two parts will interact using AJAX. The handling of this will be done on both sides.

Server-side is the part of the program which takes place on the server. In our case it is implemented in Java. This part will do the heavier computations, so the client-side only has to display the results. The client will send requests to the server, the server will compute the result of the request and send it back to the client.

GUI is a Graphical User Interface. This is the part of the program which you actually see as a user. It is the screen which shows you the program and allows you to do things.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.

AJAX (Asynchronous JavaScript and XML) is a technique in which data can be exchanged between client and server side. This can be done asynchronously, meaning your main program does not have to wait for the results of the data transaction before it can continue. We will use this as a part of our communication between our JavaScript client-side and our Java server-side.

SBML is a Markup Language specially designed for Synthetic Biology. We will use this to simulate BioBricks. An external SBML solver application will be used for simulating the BioBricks.

1.4 References

`http://www.json.org/` The JavaScript Object Notation.

`http://www.sbml.org/` The Systems Biology Markup Language

1.5 Overview

In the next chapter we will discuss the exact architectural design of our application. We will go into detail about how is system is build up. There will be a short explanation of every sub-system. This will all be done in chapter 3.2. The mapping of how different parts of our application interact with each other will be given in chapter 3.3. In 3.4 there will be an explanation of how we manage data. Where and how will we save our BioBricks? Which format will we use? These are all questions we will answer. In chapter 3.5 we will tell something about how the application handles global recourses. There we will also explain why we chose not to have a complex access structure. Concurrency, how communication is handled in our system, will be discussed in chapter 3.6. Finally we will discuss the boundaries of our system. Where does our system end and what happens in case of errors or crashes?

2 Proposed software architecture

2.1 Overview

This chapter is about what we actually want to build. We want to create a modeling environment for biologists and genetics experts to model BioBricks. These BioBricks represent genetically modified cells or genes which act as logical gates.

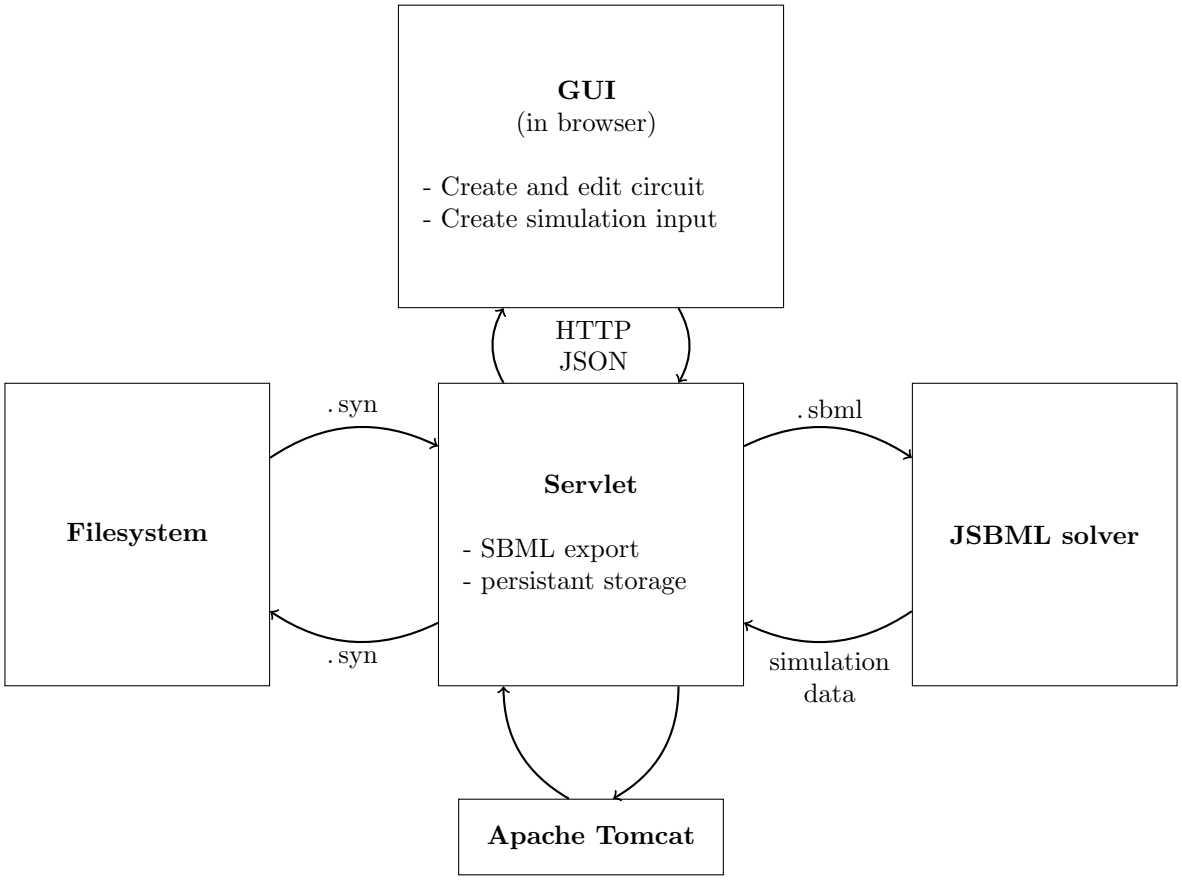
Our program will consist of two major parts: A server and a client. The server will be implemented in Java as a servlet on top of Apache Tomcat and will handle the big computations and storage of data. The client will mainly display results. This is where the user will model the BioBricks and see the simulations of the outcome. The client will be implemented in JavaScript. The client will send requests to the server. The server, in turn, will respond to the requests. The type of response will depend on the request. Some example requests are: Storing data, checks if models are valid and importing BioBricks.

There will be two major formats in which we store and transfer data. SBML is a synthetic biology XML variant. The main advantage of using SBML is that we can easily export and import BioBricks, since Java already supports this. JSON is a format in which we can easily transfer data from our Java server to our JavaScript client and vice versa.

2.2 Subsystem Decomposition

The main decomposition in our system is the devision between the client and the server. The server is essentially a servlet on top of Tomcat, responsible for file operations and the more heavy calculations, while the client is mainly just a GUI. A schematic view of this decomposition can be found in figure 1. In rest of this section we'll discuss each interaction in more detail.

Figure 1: The system decomposition



2.2.1 Interface (API) of each sub-system

Persistent storage

User generated circuits will be stored in simple text files. Since the communication between the client and the server is in JSON already, it is convenient to use that format for persitent storage aswell. The format will include:

- The circuit as a directed graph,
- Protein assignments for each edge,
- Position information for each elemental gate,
- Grouping information to support user generated 'compound gates'[1].

SBML export

Since SBML is an available standard and differential equation solvers exist for this format, we'll use it to run the simulation. The circuit will be represented in SBML and imported in the solver. The solver will respond with calculated concentrations of different species[2] as a function of time.

HTTP interface

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.
[3]

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. The server will respond to the API calls with a standard JSON object, as proposed in figure 2. The response always contains a success field. When the server cannot respond with success, the reason will be given in the message field. In case of a response with data, the client can find it in the data field of the servers response.

[1] As discussed in requirement 1, see RAD.
[2] SBML-term for different signals in the simulation
[3] **SBMLToModel** seems a logical counterpart, but because SBML is such a low level format, not all the information required to rebuild the model can be extracted. Therefore we will not implement it

Figure 2: Some examples of JSON responses in a generic wrapper

Generic response:

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

Response of getFiles():

```
{
  "success": true,
  "message": "",
  "data": {
    "files": [
      "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": {}
}
```

2.3 Hardware/Software Mapping

Our architecture is composed of two major elements, the client implemented as a web application and the server, implemented as a Java servlet on top of Apache Tomcat. The browser has to be Chrome 15+ or Firefox 7+, the supporting system hardware and operating system is not relevant. The server can be run on the same machine, which will be the case during the development stage. When the server and the client are not on the same machine, a reliable network connection has to be available.

2.4 Persistent Data Management

The application will persistently store data serverside on the disk. A database seems overkill for the rudimentary data storage that is required, and the already existent and mature text formats are good candidates to save our data in. The application will store data using the following file:

.syn (JSON)

A `text/json` file in our made up extension that embodies the entire modeled circuit/BioBrick, and possibly later also simulation output data series, all written down in JavaScript Object Notation. The object structure has to comply with that of the application to be read.

2.5 Global Resource Handling and Access Control

The application will be controlled through a web interface. Upon reflection we decided that it would be harder to ensure that only a single user can interact at one point in time, than to give some rudimentary support for multiple users. This however does not mean that we implement access control. Anyone with a browser can connect to the server and without authorization will be able to access the functionality. The decision on this was made because the focus should be on implementing the required functionality, and at first we only aim to support in the order of 5 users interacting at the same time. We agreed that security would be a nice-to-have, but to ambitious to implement with the current goals in mind.

This means that in theory anyone who has access to the network where the server runs could operate the application, so users of the application should be aware that they do not work on and/or save sensitive information.

2.6 Concurrency

As is described in the previous section, our system will implicitly support multi-user operation, so the server must be able to handle requests from different browsers. This is possible because the client can model locally without continuous synchronisation with the model on the server, and the server does not distinguish between calls from different browsers. To handle the concurrency problems of a multi-user environment, we launch certain subsystems in separate threads. Because user interaction is a dominant part of our application, concurrency problems may not hinder the workflow or be very problematic. This implies multi-threading, and the following subsystems will have to run on a separate thread with concurrency handling:

1. **Main controller**
   The main controller will have to be able to handle calls from multiple subsystems at the same time. For example the user must still be able to interact with the server when the server is taking a long time in simulating a heavy biological system.
2. **Web server (HTTP request handling and page serving)**
   These two subsystem must be able to terminate when hung in the processing of a HTTP request or serving HTTP to a client. Therefore they must be isolated in a separate threads. This will be handled by the Tomcat webserver, which has well developed support for handling multiple connections at the same time.
3. **Simulation math**
   This subsystem must also be able to operate independently of the rest of the server systems so in case lengthy simulations are performed, the server can still interact with the client and the simulation can be aborted.
4. **GUI**
   This is essentially the browser, and in this sense should also be considered a separate thread. This has the benefits as described in the system decomposition. Because we only support recent versions of the major browsers concurrency problems with the GUI are already covered.

2.7 Boundary Conditions

In a deployment context, the server will run somewhere on a server indefinately. At any point, the user can start a browser on a random device and start to use it. During the development stage, the server will run locally, and has to be started by the user before he can start using it through someting like `http://localhost:8080/`, depending on Tomcat's configuration. The user's work will be saved to the server at regular time intervals to prevent loss of work in case of a browser crash.