Chapter 5

Client-server architecture

The main features of existing simulation systems have been reviewed in Chapter 4, which compares their most important features. The importance of the client-server architecture realization in a new simulation system was shown in Chapter 1. The new developed system for the detailed simulations of neural networks applies the compartmental modelling approach; the theoretical background underlying this approach has been described in Section 3.2.3 of Chapter 3. A system has been designed with the client-server architecture, which provides the effective using of computer resources[52, 51, 40].

In this chapter, the main features and layout elements of the client-server architecture the NeuroSim simulation system based on are discussed. The implementation of the connection between the client and server on which the Genesis simulator is running is shown. The rules for transforming the model description files into the Genesis scripting language format are described. Finally, the neural model elements are introduced.

5.1 Roles of client and server

The detailed simulations of neural systems are made within the approach of compartmental modelling. The electrical circuit of each compartment is described by an ordinary differential equation, and the equations are combined into a coupled system. For computational modelling of the neural system, the equations arising from the model elements need to be solved in parallel. Considerable computational power is required for the following cases: models with a huge number of neurons, models consisting of neurons with complex branched structure and for the models consisting of neurons with complex rules for activated channels.

Thus, it becomes clear that it would be effective to divide the program into two parts: equation solver and simulation controller. While a high-power server performs extensive calculations, control and visual results presentation are left to a personal computer (see Fig. 5.1). The connection mechanism between a

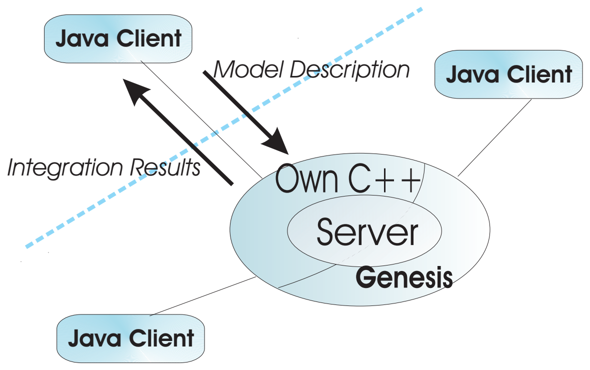

Figure 5.1: Client-server architecture of NeuroSim.

client and server is made with a Java tool – Remote Method Invocation (RMI), which supports distributing run-time objects across multiple computers.

The Java client program can be running either as an application on the user's computer or as an applet from the Internet. The user-friendly graphical interface allows one to define the elements of the experimental model. The description of the objects corresponding to elements of a neural simulation model, including the properties and their values, are automatically generated in standard XML format. All elements: neurons, neuron connections, ion channels, and inserting mechanisms were developed as objects with a serialization facility that allows one to have complete access to their methods and properties. Serialization of elements allows one to save the defined models in the form of serializable objects.

After defining an experimental model on the client side, the data and model description are transferred to the server, which is accessible through a network.

In the first stage of the project, the server for solving the system of differential equations was realized on the basis of the Genesis simulator. In the second stage, our own C++ server for numerical integration was implemented, as described in Chapter 7.

At the remote computer, the XML files containing the model and data necessary for numerical integration are converted to the necessary for server format. Files in Genesis scripting language or C++ server input files are generated using transformers developed for parsing of XML documents.

Following that, the simulation of the model from prepared input files is done on the server side. The data of simulation results are uploaded onto the client and can be used for analysis: membrane voltages and channel conductance curves can be viewed by the user; at the same time, an animation of neuron activations and the pulse's transformations through the network connections can be launched.

Summing up, the client-server architecture enables researchers using the simulation system to reduce the time needed to define models to a few minutes. At the same time, the model can be complex and require extensive calculations, the simulation of which only becomes possible by placing the mathematical operations on a powerful server or a supercomputer.

5.2 Genesis simulator as a server

In the first phase of our project, the well-known simulator Genesis was used as the server for numerical integration. Developed on Unix platforms, Genesis presents certain difficulties for the novice user in acquiring programming skills for the Genesis scripting language, in addition to the possible difficulties of working on a Unix platform. Experienced users need to spend time to master the simulation process. Therefore, the NeuroSim environment based on a client-server architecture provides a good alternative to using the Genesis package. The client side, which was developed as platform independent, provides an intuitively clear graphical front-end requiring practically no special programming skills to set up the simulation process.

5.2.1 Client-server connection

As mentioned above, the communication process between the client and server includes the transmission of the model description data from the client to the server and delivery of the calculation results back to the client. This data exchange process is realized using Java RMI, which directly supports communication among Java applications on multiple machines. This defines Java for developing a client graphical interface that could be conveniently integrated with its communication parts. Java RMI allows simple combination with the Genesis simulator, thus utilizing Genesis for numerical calculations. Moreover, Java provides platform independence for the client.

In the following, I consider the implementation of NeuroSim for connection with the server based on the Genesis simulator. The main building elements of NeuroSim as an RMI application are outlined.

Main elements of an RMI application The main idea of RMI is that the client requests an object from the server. The **"remote object"** is implemented using an interface extended from *java.rmi.Remote*. Once the client has the remote object, it invokes the object's methods as though the object were local. However, the methods are invoked on the server side with results returned.

RMI requires a common interface implementation for "remote object", which will be used by both the client and the server. All methods should throw a *java.rmi.RemoteException*. The code for implementation of the remote interface *StartGenInterface* is following:

```
public interface
StartGenInterface extends java.rmi.Remote{ public String message
             (String message} throws RemoteException;
                                            }
```

The “remote object” *StartGenImpl* will be used by the server. The class for creating a remote object implements the remote interface:

```
public class StartGenImpl extends UnicastRemoteObject implements
                                        StartGenInterface{...}
```

Then I created the RMI client, which look up the object on the server host, using *Naming.lookup.* It will typecast the object to the “remote object” type and use it just like a local object. RMI clients must know the host that provides the remote services. The URL is specified by *rmi://host/path*:

```
StartGenInterface remObject =
(StartGenInterface)Naming.lookup("rmi://"+host+"/StartGenInterface");
```

The RMI server creates an object of `StartGenInterface`:

```
StartGenImpl locObject=new StartGenImpl();
Naming.rebind("rmi:///StartGenInterface",locObject);
```

The next step is the proper creation of the client stub and the server skeleton that support the method calls. The server class and server skeleton class, as well as the remote object interface and implementation, are required for RMI server operation. Similarly, the client files, interface files, and the client skeleton class have to be available on the client machine. For the applet, they have to be available on the applet’s host machine.

The client will look up the remote object available on the server host from the specified URL.

RMI client and server for Genesis simulations The connection between client and server can be established from the menu of the client’s graphical user interface. Using methods from the Java RMI package, the program will try to make a connection to the server with a specified “host” and will then search the “remote object” implemented the interface *StartGenInterface* according to the following code lines:

```
try { StartGenInterface c;
c=(StartGenInterface)Naming.lookup("rmi://"+host+/StartGenInterface");
       ....
       } catch (RemoteException re) {
                System.out.println("RemoteException");
       } catch (NotBoundException nbe) {
                System.out.println("NotBoundException");
       }
```

The interface **StartGenInterface** contains methods from “remote object”, and the class **StartGenImpl** of a “remote object” contains their implementation. These are:

- *makeXMLFile(String n)* creates an XML file with model description, step size, and method of integration

- *TranDat(String arg1,String arg2)* performs transformation of the input XML file into a Genesis file
- *StartGenesis()* is responsible for starting the simulation with Genesis using prepared files in the Genesis scripting language
- *Vector getDat(String s)* returns resulting data in vector format

All these functions are invoked remotely on the server. Model description and simulation parameters in XML format are transferred to the server using the method *makeXMLFile(String n)*. After that, the server performs the transformation into the Genesis scripting format using the Extensible Style Sheet Language for Transformation (XSLT). The benefit of this method is that an XSLT engine converts the XML document according to formatting rules that are described in the XML style sheet (XSL) document. Thus, XSL conversion rules can be redefined for transformations into various formats. XSLT is not a standard part of Java Standard Edition, therefore one needs to download the appropriate classes and include them in the server. Then one needs to establish an object of the type *Transformerfactory*, which allows the creation of different transformers for different style sheet templates. These operations are performed in the method *TranDat()*:

```
TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer(new
                                                StreamSource(arg1));
transformer.transform(new StreamSource(input), new
                       StreamResult(new FileOutputStream(arg2)));
```

After transformation, three types of Genesis files will be created: initialization files of model elements with their properties, files with simulation parameters, and files with commands to start the simulation.

The command starting Genesis is realized using a method of the *java.lang.Runtime* class that executes the specified string command in a particular process. The Java code

```
 Runtime r = Runtime.getRuntime();
 try{ Process p = r.exec("genesis MultiCell");}
      catch(IOException e){System.exit(11);}
```

will execute the Genesis simulator with the input file “MultiCell.g”. This file contains the list of files to be included for simulation.

The output data will be stored in ASCII files, using the appropriate Genesis object. From these files, data will be transferred to the client with the “remote object” method; after that, the files will be deleted.

The **StartGenServer** class was realized as an RMI server class and thus, should be active when the client sends the request to the server host.

5.2.2 XSL transformation rules

Rules for transforming the XML data file into the Genesis scripting format are described in the XSL format document. It includes the description of elements,

```
<pnn>
  <neurons>
    <neuron>
    ...
    <comp>
    <name>soma</name>
       <property>...
       </property>
       ...
    </comp>
    ...
    </neuron>
    ...
  </neurons>
  <connections>
    <connection>
    ...
    </connection>
  </connections>
<pnn>
```

Listing 5.1: Elements of an XML neural description file

called **templates**, which can be recognized during analysis of the source file. If a pattern in the source file is matched, transformer will add the code to the output document according to predefined rules.

The element `<xsl:template match="name">` in the XSL file contains converting rules that are applied when a node `"name"` is matched. One uses the element `<xsl:apply-templates select="name">` when the rules for a template with a node `"name"` should be applied.

I will demonstrate the use of important XSL elements by the example of XML element transformations in a neural model description file. Listing 5.1 shows the parts of an XML file containing elements that will be explained in more detail. Listing 5.2 shows the XSL code in which the transformation rules are presented. As one can see, parsing starts with analysis of the elements `<pnn/neurons>` and `<pnn/connections>` that correspond to descriptions of neurons and their connections. Templates for these elements are applied with the XSL elements `<xsl:apply-templates select="pnn/neurons">` and `<xsl:apply-templates select="pnn/connections">`. Thus, the template `<xsl:template match="pnn/neurons">` contains the rule that every XML element `<neuron>` will be selected. It activates the element `<xsl:for-each select="neuron">`. The number of the cell will be added to the output file that provides the element `<xsl:value-of select="id"/>` used for selecting the value of the element `<id>`.

For every `<neuron>` element, the template `<xsl:template match="comp">`

```
 ... <xsl:template match="/">...
<xsl:apply-templates select="pnn/neurons">
<xsl:apply-templates select="pnn/connections"/>
</xsl:template>
<xsl:template match="pnn/neurons">
<xsl:for-each select="neuron">
str cellId =<xsl:value-of select="id"/>
<xsl:apply-templates select="comp"/>
</xsl:for-each></xsl:template>
<xsl:template match="comp">
<xsl:choose> <xsl:when test="name[.='dend']"> ...
</xsl:when> <xsl:otherwise>...
</xsl:otherwise></xsl:choose>
<xsl:apply-templates select="property">...
</xsl:template> <xsl:template match="property">
...</xsl:template>
<xsl:template match="pnn/connections">...
</xsl:template></xsl:template>
```

Listing 5.2: XSL document transformation rules applying to files of neural element descriptions.

will be applied. It contains the rules that are based on the "logic" XSL elements. Different code will be generated for each compartment corresponding to a soma and a dendrite. The logical tests are realized the with the XSL elements `<xsl:choose>`, `<xsl:when>`, and `<xsl:otherwise>`, that allows to test different logical conditions.

As a result of this transformation, the Genesis script file is automatically created by a transformer on the server side, instead of preparing it manually for every newly arising modelling task.

5.2.3 Elements of neural models

A simulation model is constructed using the Java client graphical interface. The electrical properties of individual neurons are described with Hodgkin-Huxley type voltage-dependent ionic channels. The neuron connections can be made by a chemical synapse. The standard inserting mechanisms, like spike generator and synaptically activated channel, can be included into the model. Electrical excitation can be implemented with different types of cell stimuli: pulsed stimuli, randomized spike generation, or constant injection.

The executable Genesis file for simulation is automatically created during the parsing of the XML client's data file and consists of predefined functions of initialization of neural elements and assignation of their fields.

Here we give a short description of the standard Genesis elements. An object `compartment` is commonly used in Genesis simulations and uses the fields V_m, E_m, R_m, C_m, R_a, and *inject*, corresponding to the labels in Fig. 3.11. Morphological parameters should be taken into account for construction of realistic compartments. These parameters are specific resistance RM, specific membrane capacitance CM, specific axial capacitance RA, compartment's length and diameter.

Compartments are connected in Genesis with two messages AXIAL and RAXIAL, which are used for calculation of axial currents in both directions. Voltage-gated channels with Hodgkin-Huxley dynamics are implemented with the object `hh_channel`, after setting parameters of gate variable dynamics (see Eq. 3.6). The elements presenting a channel and compartment are linked with messages CHANNEL and VOLTAGE for calculation of the channel current.

Synaptic input can be generated in three ways: as synaptic activation in response to presynaptic action potential generation, as random synaptic activation, and as pulsed synaptic generation. All of these methods are carried out using the Genesis object `synchan`, which simulates a time-dependent synaptically activated ionic channel. Similar to a voltage-gated channel, a `synchan` element should be linked with the postsynaptic compartment using the messages CHANNEL and VOLTAGE.

Connections between two cells are performed with the synaptic channel that is activated with the spike initiation in the presynaptic cell. The presynaptic compartment should contain an object `spikegen` performing threshold spike discrimination. The `spikegen` notifies spike initiation via an INPUT message from the presynaptic compartment. The `spikegen` and `synchan` objects interact through the SPIKE message.

A SPIKE message may only be sent from certain objects, e.g. the `spikegen` or `randomspike` objects. The `randomspike` object is used for implementing random synaptic responses, which generate a series of random spikes.

The pulsed synaptic input is produced with the object `pulsegen`. It generates a series of pulsed events and is linked with a `spikegen` object via INPUT and SPIKE messages.

The `pulsegen` object can also be used to apply a current to a compartment in pulsed form. In this case, the object `diffamp` plays the role of a subsidiary element that produces an output from two inputs.

5.3 Summary

The basic features of the client-server architecture NeuroSim is based on have been presented in this chapter. The client-server architecture is an effective solution for the mathematical modelling of neural systems where extensive computational resources are needed. This architectural technique was not realized in the existing simulation packages whose summarized characteristics can be found in Chapter 4.

The roles of the client and server, as well as the effective working of a whole simulation system have been shown. The chapter contains details of the connection implementation by way of the example of the Java client and the server based on the Genesis simulator. The Java RMI tool used for distributing the working client and server components of the NeuroSim system have been outlined; the implementation of the main components for RMI connection in our system has been described. The chapter contains the transformation rules for the XSLT engine, which are used to parse model description files into the Genesis format for simulation. The last section of the chapter includes a description of the neural model elements.