University of Nevada, Reno

HTTP-based Solutions to The Extension of the NCS Brain Simulator

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in Computer Science & Engineering

by

Nathan Michael Jordan

Dr. Frederick C. Harris, Jr., Thesis Advisor

May, 2014

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

NATHAN MICHAEL JORDAN

Entitled

HTTP-Based Solutions To The Extension Of The NCS Brain Simulator

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Dr. Frederick C. Harris Jr., Advisor

Dr. Sergiu M. Dascalu, Committee Member

Dr. Yantao Shen, Graduate School Representative

David W. Zeh, Ph.D., Dean, Graduate School

May, 2014

Abstract

The NCS brain simulator is a tool that allows neuroscientists to create biologically-realistic simulations of neurons and neural networks. NCS, while powerful, is written in C++/CUDA, and by itself would not be easy or desirable for a neuroscientist with hardly any programming experience to create a model in this kind of low-level environment. To facilitate the use of less-technical means of interacting with the simulator, some kind of intermediary is needed to interact with the brain simulator and turn high-level concepts neuroscientists are familiar with into a lower-level format that the simulator can use. To do this, we have created a service that interacts with external programs and services using a RESTful HTTP interface. This service handles the operation of the NCS simulator, access control, session management, data storage and retrieval, and other functions related to the operation of the brain simulator. Implementing the service as a RESTful interface allows for the easy integration of other technologies such as Javascript and Python. Using this interface, we have created a suite of browser-based tools, as well as a Python client library to work with the simulator. These tools provide a higher-level method of building, reporting, and visualizing simulations, which are critical for neuroscientists to be able to effectively use the simulator.

Acknowledgments

I'd like to thank my committee members Dr. Yantao Shen, Dr. Sergiu Dascalu, and Dr. Fred Harris Jr. I would like to give additional thanks to Dr. Fred Harris Jr., who has allowed me to work in the Brain Computation Lab for the last three years and has provided a great deal of assistance in my pursuit to complete my thesis and graduate study at the University of Nevada. I would also like to thank my fellow grad students Roger Hoang, Devyani Tanna, and Torbjorn Loken for a wealth of insight and moral support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Experiments in the field of neuroscience have traditionally been conducted using invasive physical methods including *in vivo* (using a living organism) and *in vitro* (in an artificial laboratory environment) methods. These methods, while effective in small-scale experiments, become impractical for performing experiments on a larger scale and are often times quite expensive. To combat these problems, neuroscientists have turned to *in silico* methods to perform experiments, a field known as computational neuroscience. To create neurological experiments in a computer environment, a number of neural simulators that allow researchers to model biologically realistic neurons and neural networks have been created[21]. These neural simulators allow researchers to build, run, and output the results of a simulation. While each simulator has the same goal in mind, they each have a different set of features and limitations. Researchers must choose which simulator provides the most appropriate set of functionality for their experiments [6].

The University of Nevada has created its own neural simulator called NCS, or Neo-Cortical Simulator. NCS tends to focus on larger scale simulations that involve large numbers of neurons and synapses. Performance was also a key consideration when designing NCS, and as a result the simulator was designed to be as fast as possible without adding features that would create additional computational overhead. By ensuring that the simulator is as fast as possible, NCS can run a million-cell model in realtime [17].

Although very fast and scalable, the NCS simulator by itself does not provide a

convenient way for researchers to perform common tasks surrounding the simulation in an intuitive way. NCS provides a thin and feature-limited Python layer that allows it to be used with Python-based programs. This thin layer exposes the minimal features that NCS provides, such as adding neurons and synapses to the simulation. To make the NCS simulator a more useful tool for neuroscientists, additional higher-level features must be added that augment the minimalist approach to designing NCS. These features should provide neuroscientists the ability to easily create models, configure simulations, manage the data generated by simulations, and visualize their results.

To accomplish this task, we have created a suite of tools that surround the NCS simulator in the form of a web-based interface and higher-level Python interface that gives a much more intuitive medium for interacting with the brain simulator. The web-based interface provides the ability for neuroscientists to create simulations without programming experience, and the Python interface provides high-level access using an easy-to-use Python library. This thesis covers the development and implementation of these tools, as well as the architectural and design choices that were made when creating them. The rest of this thesis is formatted as follows: Chapter 2 discusses background topics surrounding these tools; Chapter 3 details the design and implementation of the software; Chapter 4 presents conclusions and discusses possible future work that could be implemented in the software.

Chapter 2

Background

2.1 Other Simulators

There are a few neural simulators other than NCS that should be discussed in order to justify such a solution. The first of which is NEURON[6], which is developed at Yale and Duke. Unlike NCS, NEURON tends to focus on relatively small simulations, as it uses accurate, albeit computationally-expensive formulas to model neural interactions. NEURON was created in C/C++/FORTRAN and uses a scripting language called Hoc to describe models and create the simulation. NEURON currently supports simulating using a single machine or multiple machines using MPI to distribute the workload. Simulations can be run from the graphical user interface (which generates Hoc code), a shell session using Hoc directly or Python script using the neuron Python package. The NEURON GUI consists of a multi-windowed environment in which users can create models, execute simulations and view the results[6]. The NEURON user interface is shown in Figure 2.1 and example Hoc code is shown in Figure 2.2[6, 8].

Another simulator, called GENESIS (GEneral NEural SImulation System)[6], is similar to NEURON in that it mostly models smaller neural networks including interactions within a single neuron[4]. GENESIS uses a proprietary scripting language as well as formatted data files to create and run simulations. GENESIS can use MPI or PVM to parallelize its computations across multiple machines. Like NEURON, it can run simulations via a command-line interface known as G-Shell, a Python wrapper library, or a graphical user interface named XODUS[6]. An example of the XODUS

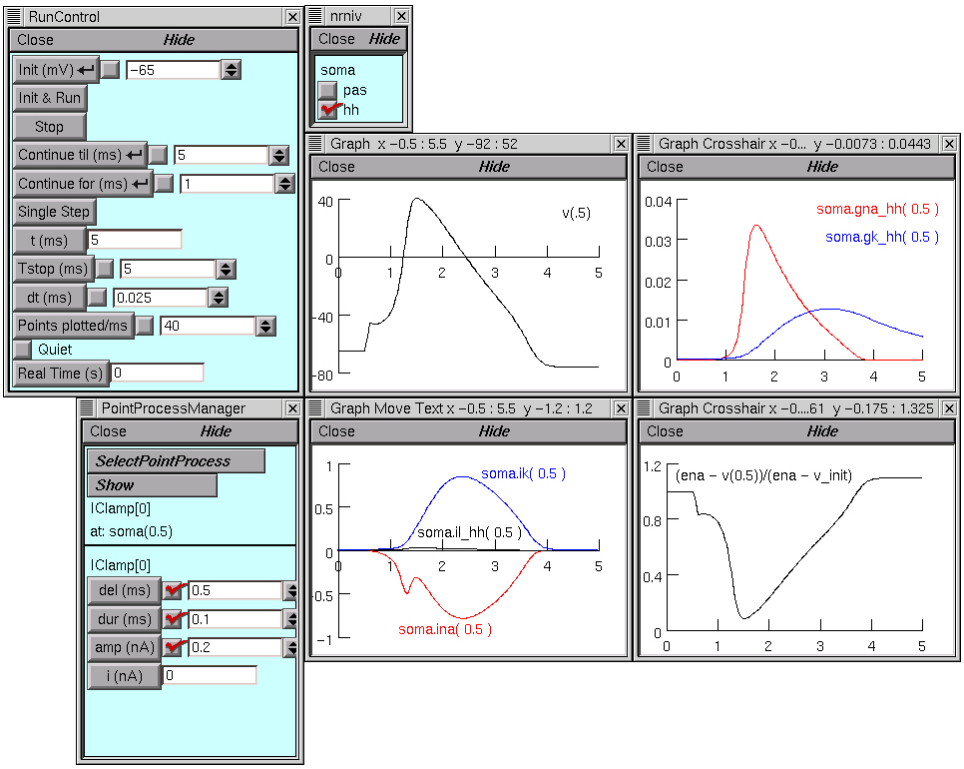

Figure 2.1: The NEURON GUI showing the results of a simulation that has been run using the simulator[6].

```
create soma, dend, dendx, axon
connect dend(0), soma(1)
connect dendx(0), soma(1)
connect axon(0), soma(0)

access soma
pt3dclear()
pt3dadd(0, 0, 0, 8)
pt3dadd(6, 0, 0, 18)
pt3dadd(10, 0, 0, 20)
pt3dadd(14, 0, 0, 18)
pt3dadd(20, 0, 0, 10)

insert hh
Ra = ra

dend {
  nseg = 21 // may need more
  diam = DEND_DIAM
  L = DEND_LENGTH

  insert hh
  gnabar_hh /= 5
  dend_gnabar_0 = gnabar_hh
  gkbar_hh /= 5
  Ra = ra
}
```

Figure 2.2: A code snippet from a Hoc script creating a neuron[8].

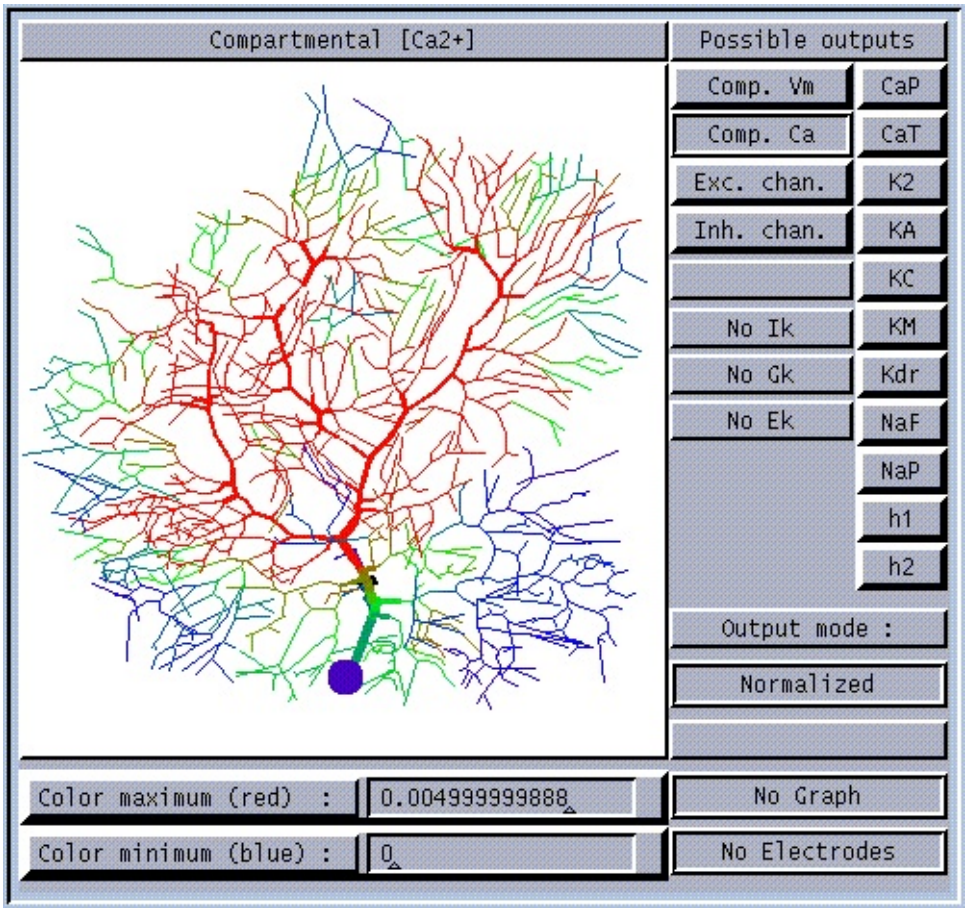

Figure 2.3: The GENESIS GUI (called XODUS) showing a visual representation of a Purkinje neuron model[6].

user interface can be found in Figure 2.3 and a sample of the GENESIS scripting language can be found in Figure 2.4[4, 6].

Unlike the NEURON and GENESIS simulators, the NEST (NEural Simulation Tool)[6] simulator tends to focus on larger neural networks rather than smaller ones. NEST uses a command-line interface as well as a Python wrapper to create models and run simulations. Unlike its counterparts, NEST lacks an out-of-the-box user interface component. Results of the simulation must be viewed using 3rd party Python libraries like matplotlib[6]. An example of the NEST scripting language can be found in Figure 2.5[14].

The last simulator (and perhaps most-similar to NCS) that will be discussed is

```
create cell /n
create segment /n/soma

model_parameter_add /n/soma Vm_init -0.068
model_parameter_add /n/soma CM 0.0164
model_parameter_add /n/soma RM 1.500
model_parameter_add /n/soma RA 2.500
model_parameter_add /n/soma ELEAK -0.080

model_parameter_add /n/soma LENGTH 4.47e-5
model_parameter_add /n/soma DIA 2e-5

runtime_parameter_add /n/soma INJECT 2e-9
output_add /n/soma Vm

run /n 0.05

quit
```

Figure 2.4: A code sample from a GENESIS script creating a neuron, injecting current into it, and outputting the voltage[4].

```
/iaf_neuron Create /n1 Set
/iaf_neuron Create /n2 Set
/iaf_neuron Create /n3 Set
n1 n2 Connect
n1 n3 Connect
```

Figure 2.5: A small code snippet of a NEST script creating three neurons and connecting them in a chain[14].

BRIAN. BRIAN is a neural simulator developed naively in Python and does not use a command-line intermediary: models are written exclusively in Python [15]. BRIAN uses the scipy and numpy libraries to speed computations by taking advantage of native C code execution. It does not, however, provide the ability to parallelize the simulation across multiple machines. An example of a simple BRIAN simulation script can be found in Figure 2.6[15].

A comparison of these simulators and their features can be found in Table 2.1[17].

```
from brian import *
eqs = '''
dv/dt = (ge+gi-(v+49*mV))/(20*ms) : volt
dge/dt = -ge/(5*ms) : volt
dgi/dt = -gi/(10*ms) : volt
'''
P = NeuronGroup(4000, eqs, threshold=-50*mV, reset=-60*mV)
P.v = -60*mV
Pe = P.subgroup(3200)
Pi = P.subgroup(800)
Ce = Connection(Pe, P, 'ge', weight=1.62*mV, sparseness=0.02)
Ci = Connection(Pi, P, 'gi', weight=-9*mV, sparseness=0.02)
M = SpikeMonitor(P)
run(1*second)
raster_plot(M)
show()
```

Figure 2.6: An example of a BRIAN script creating a group of 4000 neurons and creating a raster plot from the result of the simulation[15].

Table 2.1: This table provides a comparison between the major neural simulators including NCS based on the features they provide users (modified from [17]).

Simulator	Platform	Language	Coding Style	GUI	Realtime Visualization	Neuron Models	Parallel Support	Python Interface
BRIAN	Linux Mac Windows	Python	Python	No	No	LIF HH IZH	None	Yes
GENESIS	Linux Mac Windows	C	C	Yes	No	HH	MPI PVM	No
NEURON	Linux Mac Windows	C C++ FORTRAN	HOC Python	Yes	No	HH LIF	MPI	Yes
NCS	Linux	C++ CUDA Python	Python	Yes	Under Development	HH LIF IZH	MPI GPU (ZeroMQ planned)	Yes
NEST	Linux Mac Windows	C++ Python	Python SLI	No	No	HH LIF IZH AdEx MAT2	MPI	Yes

2.2 NCS

NCS is a spiking neuron simulator that is written in C++ and CUDA. The simulator, similar to the BRIAN simulator discussed earlier, its currently focused on simulating large-scale neural networks, on the order of hundreds of thousands to millions of neurons and billions of synapses[17, 22]. To handle these large-scale simulations, NCS is designed to scale to handle the sizable computational throughput needed. It is designed from the ground up to distribute the workload across multiple processors on multiple machines. In addition to using multiple CPU cores on each node, NCS also will detect and use NVIDIA GPUs if they are present on the machine to take advantage of the SIMD capabilities of these cards as well as their floating-point performance[22].

Running on top of the C++ code is a thin Python layer that provides lower-level access to the simulator. This layer doesn't attempt to provide advanced capabilities in an effort to keep the C++ code as streamlined and as simple as possible without having to add additional overhead for higher-level functionality. For example, NCS doesn't explicitly provide any form of grouping or organization for the neurons it simulates. Rather, it keeps track of groups of neurons, individual neurons, and the connections between them. This may work for smaller simulations, but at the scale of millions of neurons and billions or trillions of synapses, this approach to organization would leave most users lost in their own design. Similarly, a thin python layer without programming conveniences like classes removes a level of abstraction that would be expected from a normal programmatic API. In addition, neuroscientists might like to avoid coding altogether and focus on the biology.

To provide an intuitive and modern way for neuroscientists to interact with the NCS simulator, a more capable and user-friendly interface needs to be created. Additionally, to make the simulator more extensible as a cluster-based neural simulator running large-scale models, a better method is needed to organize, create, run, manage, store, and report on simulations than running the simulation via a shell on the cluster and directly manipulating the filesystem. These characteristics of NCS and other neural

simulators do not help them create collaborative environments where experiments can be shared, recreated, and verified. A better approach is needed to handling the many tasks that surround these simulators.

2.3 Client-Server Model

To solve these shortcomings, we have created a client-server based approach to the design of NCS. In this model of computing, a server running on a machine accepts requests from clients who wish to use the servers resources. A diagram of the client-server model can be found in Figure 2.7[24]. In the context of NCS, the server will run on the master node of an NCS cluster and directly interface with the low-level Python interface. The server is responsible for multiple functions. Firstly, the server should be able to manage and keep track of the simulator. This includes calling NCS to run a simulation, ensuring that clients don't attempt to run a simulation while another is in progress, and managing the output of the simulator including resulting data, errors, warnings, and other messages as well as allowing clients to get access to this information. Using this computation model as opposed to the methods used by existing simulators provides a number of advantages. Firstly, the simulator can be used by multiple remote clients. This allows many different researchers to use the same cluster, and allows simulations to be run from non-console clients, such as a web browser. Secondly, it removes the need for direct shell access to the machine, which removes potential security risks associated with providing a remote shell. Lastly, it provides a centralized management system in which simulations can be logged, stored and shared in a consistent format, rather than the typical approach of allowing each user to create their own organizational methods. This makes it easier to share results, duplicate experiments, and incorporate models from other researchers into your own experiments.

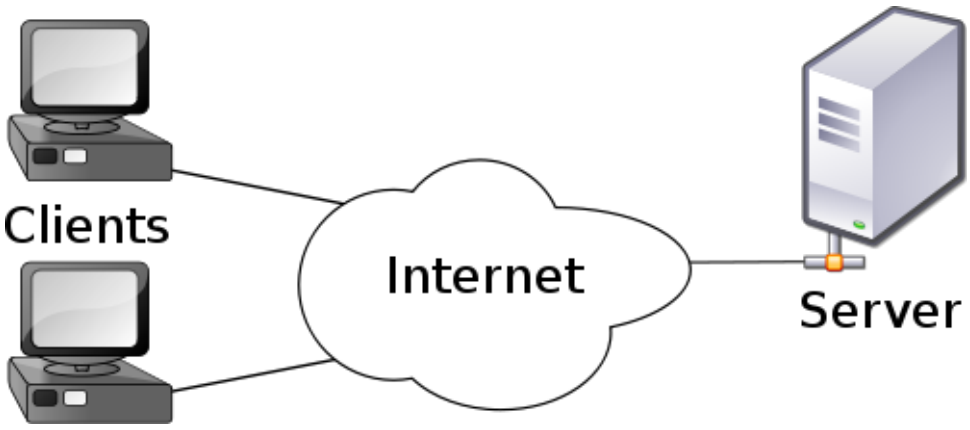

Figure 2.7: The client server model allows for multiple clients to request resources from the server[24].

2.4 HTTP & REST

2.4.1 HTTP

As a result of the client-server architecture chosen for NCS, a method is needed to transport information from the client to the server and vice versa. The HTTP protocol is a time-tested protocol used to transport textual and binary data over the Internet and various other computer networks. This makes HTTP an ideal candidate for handling the transfer of information between the client and server.

A HTTP transaction begins with a request from the client to the server. Each request can be one of four types: GET, POST, PUT, and DELETE. Each of these methods requests the server to perform a specific action on a target. The target of the action is specified as the *path*. The path is specified like a UNIX filesystem path, in the form of a series of strings separated by forward slashes. The effect of actions GET, POST, PUT, and DELETE on a path are defined by the HTTP server itself, but generally follow a certain format[13]. The GET method instructs the server that the client is requesting information about the path included in the request. For example, when visiting a website, a GET request to the path of / would ask the server to provide information pertaining to the index of the site, whereas a GET request to the

path /blog would request information about the websites blog (provided such a path exists on the server). A POST request to /blog might instruct the server to create a new blog post with the data provided with the request (such as a title, author, and content).

Typically, when browsers interact with websites the data returned from the server is HTML, which the browser uses to render the page in the browser window. However, HTML is far from the only type of information that an HTTP response can contain. HTTP responses can contain plain text, binary data (including images), XML documents, JSON objects, or any other user-defined format. Following the standardization of HTTP/1.1 in 1999, which added the PUT and DELETE methods, HTTP became capable of performing all CRUD (Create, Read, Update, Delete) operations for persistent storage[13]. The CRUD options of HTTP/1.1 form the basis of the REST architectural style[12].

2.4.2 REST

REST (REpresentational State Transfer) is an architectural pattern that defines a method of interaction between a client and server. The key component behind the REST architecture is the concept of a *resource*. A REST resource, in the context of HTTP, is a URI, or Uniform Resource Identifier. A URI for a web application is typically the location of the resource on the web server in the form of a URL or Uniform Resource Location[12]. For example, the URI for the users resource might be http://example.com/users and a URI for a student who's student ID is 1033 might be http://example.com/student/1033. These URI's uniquely identify their respective resources on the servers they are hosted on as well as the Internet as a whole.

While the URI specifies a way to uniquely identify a resource, a method is needed to represent the resource[12]. This is typically done using one of the formats mentioned in the previous section such as HTML, XML, or JSON. For example, a student resource might be represented in JSON format shown in Figure 2.8 or in an XML format such as in Figure 2.9. These formats provide a consistent method of interaction between

client and server.

To interact with a RESTful interface in the context of HTTP, the client sends a request to a particular URI on the server with one of the HTTP methods mentioned previously. Sending a GET request tells the server that the client wishes to receive the representation of the resource it requested. If the client wished to receive the representation of the aforementioned student with ID 1033, it would send a GET request to the URI http://example.com/student/1033 and the server would respond with the information shown in Figure 2.8. To add a student, the client would send a POST request to the server containing the representation of the student (in the same format as Figure 2.8) directed at the URI corresponding to the new student's ID, say 1555, at http://example.com/student/1555. To update the student with ID 1555, the client would send a PUT request to the URI http://example.com/student/1555 with the updated user representation. Similarly, to remove the student, the client would send send a DELETE request to the URI http://example.com/student/1555. This architectural style makes a HTTP API standardized and predictable, making it easy for clients to interact with the server.

2.5 The JSON Format

There are several ways to transmit information over HTTP, some of which are discussed in the HTTP section of this chapter. The JSON (JavaScript Object Notation) format, as its name suggests, is derived from the Javascript syntax for an object. JSON, which was defined in 2001 by Douglas Crockford, is a textual format that takes a minimal approach to representing data[5]. Unlike XML, JSON doesn't use opening and closing tags, but uses a list of properties instead, making the files smaller[20]. Additionally, JSON has several datatypes, which is a feature not present in XML, making it easily parsed into programming languages such as Python, Ruby, and Javascript, which usually require a single function call to turn a string of JSON into a language-native object. JSON also has a binary format called BSON that can be used to transfer and store information in a binary format that is more space conscious and holds a superset

```
{
    "name": "Bob",
    "grade": "sophomore",
    "age": 24,
    "phone": "1+(775)555-5555"
}
```

Figure 2.8: This figure shows how a user might be defined using the JSON format. The student contains four attributes. Name, grade, and phone are string attributes, whereas age is a numerical attribute.

```
<student>
    <name>Bob</name>
    <grade>sophomore</grade>
    <age>24</name>
    <phone>1+(775)555-5555</phone>
</student>
```

Figure 2.9: This figure shows how a user might be defined using the XML format. Like the JSON example in Figure 2.8, this student object contains four elements. XML doesn't provide the ability to dictate the type of elements, so all elements contain strings.

of the normal JSON data types[9].

There are six data types available in the JSON format: number, string, boolean, array, object, and null[5]. All strings in JSON are enclosed within double-quotes and are UTF-8 encoded. A JSON object is an associative array of name/value pairs, enclosed within curly braces. The names must be strings followed by a colon and then the value. The key/value pairs are delimited by commas. Numbers are decimal numbers that can be in E notation or fractional. Arrays are lists of values within square brackets delimited by commas. Boolean values are simply true and false, and null is represented as the text null. An example JSON document using all types is shown in Figure 2.10.

```
{
    "company_name": "Planetary Express",
    "phone": "1+(800)555-5555",
    "manager": "Hubert Farnsworth",
    "company_net_worth": 0.0,
    "personnel": [
        "Phillip J. Fry",
        "Turanga Leela",
        "Hermes Conrad",
        "Amy Wong",
        "John Zoidberg",
        "Scruffy",
        "Bender Rodriguez"
    ],
    "total_deliveries": 100,
    "website": null,
    "is_profitable": false
}
```

Figure 2.10: This figure shows all valid JSON types. company_name is a string, personnel is an array, total_deliveries is a numerical value, website is a null value, and is_profitable is a boolean value.

2.6 WebSockets

HTTP is a request-response protocol. It does not provide a method of streaming data between the client and server in a full-duplex fashion like that of TCP. To solve this problem, a new protocol was created that allows full-duplex communication via HTTP called WebSockets. WebSockets allow Javascript running on the client's browser to initiate a connection with a server in which both the client and the server can send data to one another simultaneously[11]. This is helpful in many different applications where it would not be desirable to continuously request data from the server (polling) when there may not be any data that the client needs. When using WebSockets, the client can simply wait until the server has data to send and receive it asynchronously without constantly polling the server for it. This greatly reduces the network footprint of real-time HTTP-based applications and provides a much more intuitive interface for the exchange of data between client and server when such situation arise. For example, a web-based instant messaging client could use WebSockets to wait for incoming

messages from the server and receive them as soon as they are ready, as opposed to constantly polling the server many times per second to check for new messages.

2.7 MongoDB

In recent years, a number of new database systems have been developed that do not rely on the traditional relational-model, which often use SQL as their database language. These databases are colloquially known as NoSQL databases in reference to the fact that they do not use SQL as a data management language[19]. MongoDB is one of these databases. Instead us using tables and columns like in a traditional database it uses collections and documents[9]. Documents are JSON objects that are stored in a BSON format on disk. A collection is a list of documents, usually with a similar format (but this is not required). This type of database is advantageous because it doesn't require a strict schema with each attribute in its own column like in a SQL database[9]. This allows more flexibility in the way you want to store and retrieve your data. For example, storing the data shown in Figure 2.10 would require a few different tables in a well-designed SQL database. MongoDB becomes even more advantageous when the data your application works with is already in a JSON format. MongoDB uses Javascript as the language to manage the database data, which provides methods for CRUD operations as well as a few aggregate functions such as summation and average. More complicated queries require the use of a MapReduce programming model[9]. Database ORMs exist in many languages for MongoDB, so often times the user will never need to use Javascript to interact with the database and can use the language of their choice. Python contains a a library called MongoKit that performs this task[23].

2.8 Client-Side Web Development

2.8.1 Bootstrap

Over the years, the Internet and the World Wide Web have seen a massive expansion in size, ubiquity and utility. As the World Wide Web evolves, the way websites are built and designed also continues to evolve. Originally, websites were designed by hand and often took an exorbitant amount of time to design and implement. Adding to the problem, different browsers render web pages differently; even when the code is exactly the same, the differences can often times be dramatic. To combat this, a number or tools and frameworks have come about that aim to be both easy to use and cross-browser compatible[18]. One of these frameworks is Bootstrap.

Originally developed by Twitter for creating a consistent UI for internal tools, Bootstrap is a front-end framework that provides design templates that work across browsers and for various screen sizes (including mobile) and aims to provide a clean and aesthetic user interface. Web page elements such as forms, buttons, headers, tables, navigation, alerts, panels and numerous others are included in the Bootstrap package so developers don't have to create and style them themselves. Bootstrap includes a number of Cascading Style Sheet (CSS) rules to create and style the widgets and elements on the page, as well as a small amount of Javascript to make some of the elements interactive, such as the accordion component that shows and hides different panels in an accordion-like fashion. An example of a web page created with Bootstrap is shown in Figure 2.11 and some of the components Bootstrap contains can be found in Figure 2.12[2].

2.8.2 AngularJS

While Bootstrap changes the way we create the look and feel of websites, AngularJS changes how we make web pages dynamic. Traditionally most of the code front-end developers would write involves directly manipulating the elements on the HTML page when data changes or the user interacts with the application. Most of this kind

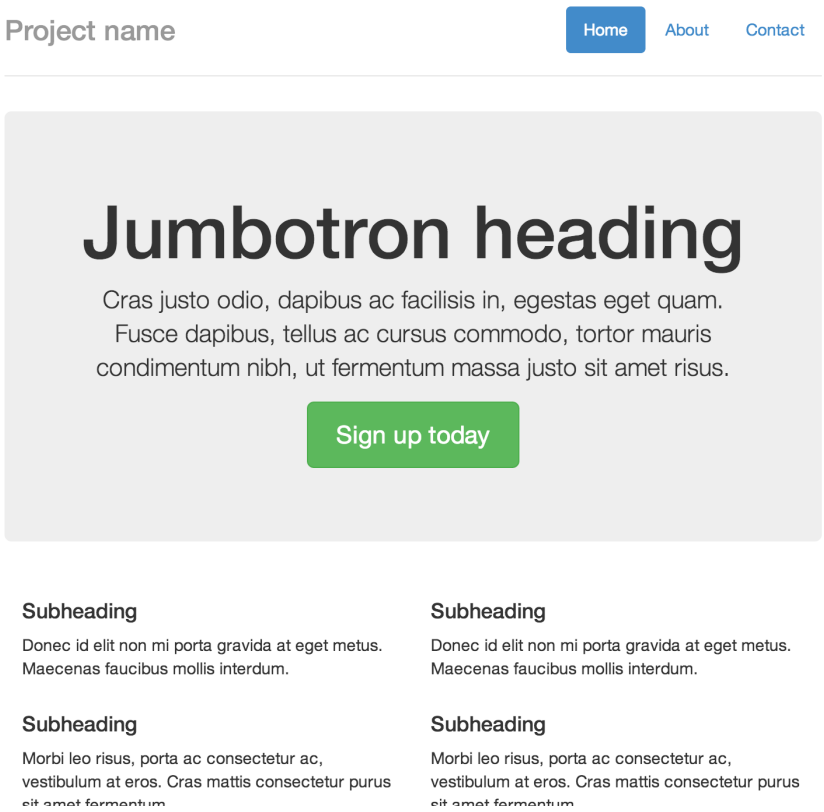

Figure 2.11: This is a simple example page that was created with the Bootstrap front-end framework. The page contains a navigation bar, page header, jumbotron component, and several paragraph elements that stack if the browser window becomes small enough[2].

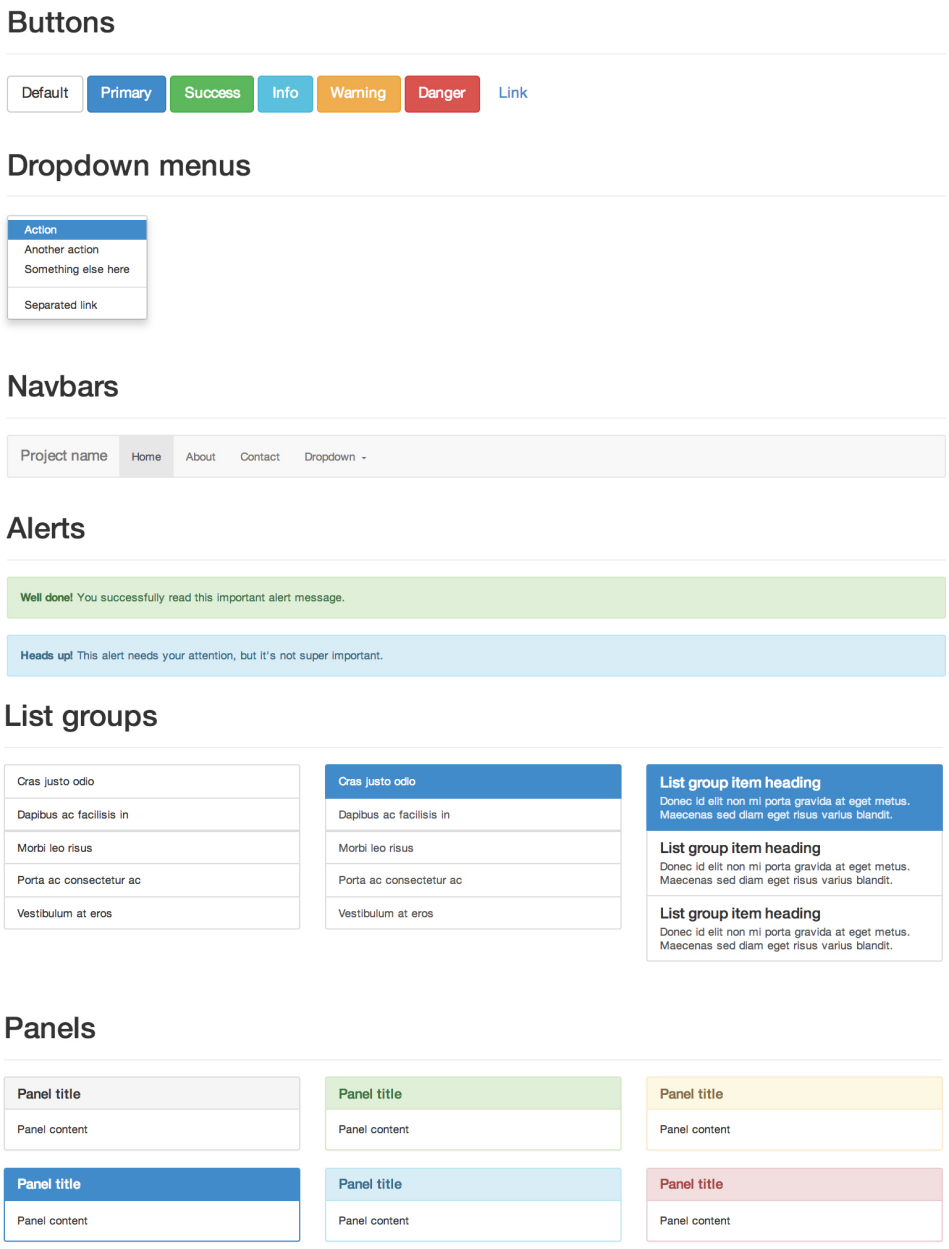

Figure 2.12: This figure shows several more Bootstrap components that are available for use in web applications[2].

of development is tedious, and writing the code for it takes time away from other tasks that are central to the functionality of the application. Angular removes the need to write the code that manipulates the elements on the web page and allows developers to focus on the background tasks that need to be performed such as fetching data from a server[10]. It does this by providing developers a way to create declarative-style directives in the HTML of the web page. An example of this would be rendering elements from a list on the page, such as a list of books. As books are added, deleted, and modified from a list in an Angular Javascript module, they are automatically updated on the web page instantly. This is a possible due to Angular's two-way data binding concept that is at the core of its functionality. As a result of the two-way data binding feature, any changes to the view are reflected in the model of the application, and any changes in the model are reflected in the view. A diagram illustrating this two-way data binding can be found in Figure 2.13[1]. If you compare this with a traditional web application using jQuery to manipulate the DOM, you can see that the complexity of accomplishing a two-way data binding would greatly increase and would require you to write a lot more code. An example comparing the code required for each of these approaches can be found in Figure 2.14 and Figure 2.15.

Angular also provides a *resource* module that allows developers to easily interact with RESTful interfaces. For each resource only requires a location be specified in the form of a URL where the resource can be found. Other options may be provided to enhance the behavior of the resource object in the event custom functionality may be needed, or in other cases when a list of objects might be returned instead of just a single object. This makes it trivial to create a Javascript application that interacts with a RESTful web application. An example of creating and using an Angular resource can be found in Figure 2.16.

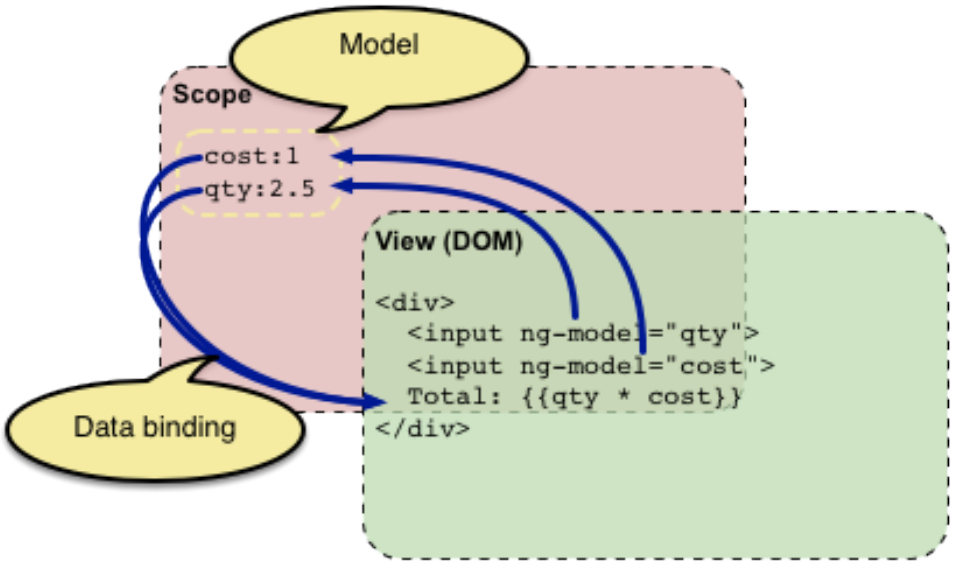

Figure 2.13: This figure illustrates the concept of Angular's data-binding feature. Changes to the scope's cost and qty variables are automatically propagated to the view without any further action required by the developer. If the user changes the values in the input areas on the web page, those changes are automatically reflected in the scope variables as well[1].

```
$.ajax({
  url: '/message_list',
  success: function (data, status) {
    var result = JSON.parse(data);
    for (var i = 0; i < result.length; i++) {
      $('ul#log').append('<li>' + result[i].msg + '</li>');
    }
  }
});
```

```
<ul class="messages" id="log">
</ul>
```

Figure 2.14: This Javascript snippet uses JQuery to perform an HTTP request to get some data from the server and add it to an HTML list. The Javascript code is highly dependent on the HTML code. The *'ul#log'* provides a specific HTML element to add the element to. What if the data needed to be used in another HTML element? We would need to modify the request function.

```
$http( '/message_list' ).then( function ( response ) {
  for (var i = 0; i < response.length; i++) {
    $scope.log.push( response[i] );
  }
});
```

```
<ul class="messages">
    <li ng-repeat="entry in log">{{ entry.msg }}</li>
</ul>
```

Figure 2.15: This Javascript snippet uses Angular to perform an HTTP request to get some data from the server and add it to an HTML list. The data being retrieved has no ties to the HTML code at all. In the HTML, we add an angular directive (ng-repeat) to loop over the list of messages and add the elements dynamically.

```
var User = $resource('/user/:userId', {userId:'@id'});
User.get({userId: 1155}, function(user) {
  $scope.users.push(user);
});
```

Figure 2.16: This figure demonstrates Angular's resource module. This module allows Angular to easily interact with RESTful web servers. The developer defines the Angular resource using the URL of the resource on the server. Once the resource has been defined, the developer can query the RESTful interface using the resource object.

Chapter 3

Design & Implementation

3.1 NCS Software

The NCS simulator provides the ability to simulate biologically-realistic neural interactions at a large scale. While NCS performs the core goals of a brain simulator, there is a much larger set of functionality that is needed for a simulator than running the simulation itself. This functionality includes building the simulation, running the simulation, storing simulation data, and analyzing results, which is imperative to the researcher using any brain simulator. We designed a suite of tools that surround the NCS simulator that provide users the ability to perform these tasks. These tools include a consistent storage and management system for the simulations users create, a programmatic Python interface, a web-based interface for building, running and analyzing simulations, and a way to visualize the results of the simulation. With the construction of these tools we have also introduced a new architectural concept, the client-server model, that is unique to current brain simulators. A diagram showing a high-level overview of NCS and its surrounding software can be found in Figure 3.1

3.2 NCS-Daemon

Part of the goal in designing NCS was to provide the simulator as a service that multiple clients could interact with rather than as a program that is run from the command line. To do this, a process or daemon must be consistently running on the root node of the NCS cluster that accepts connections from clients attempting to use

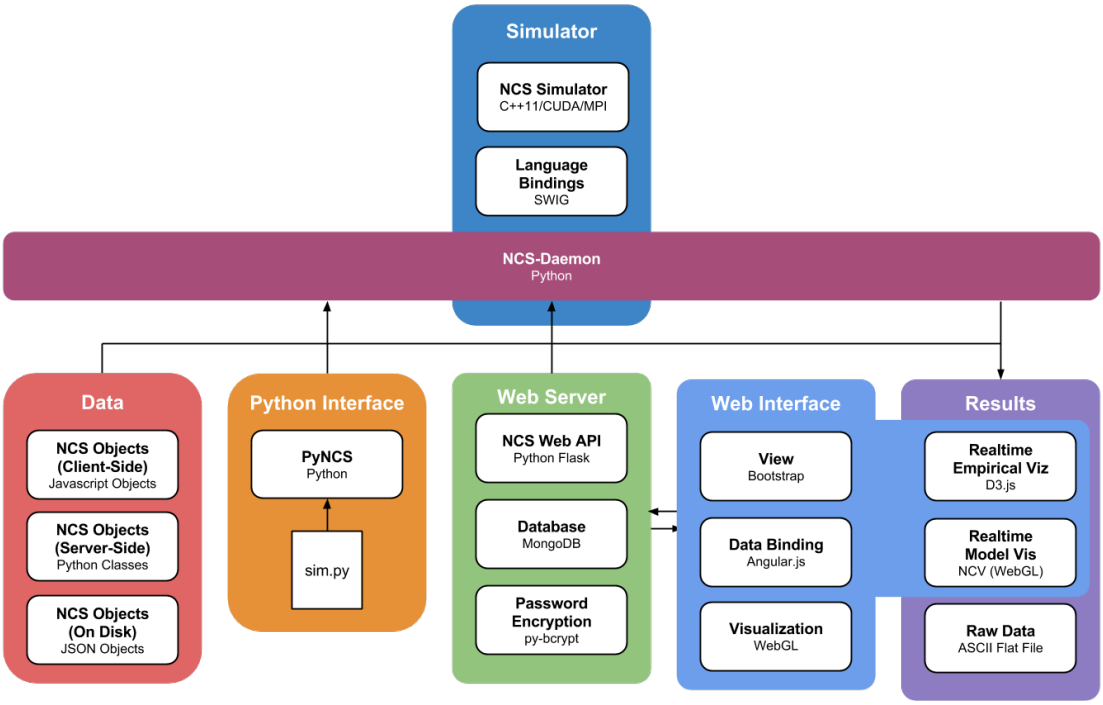

Figure 3.1: This diagram is a graphical representation of the NCS software suite. The simulator performs the core simulation tasks and is shown in dark blue. NCS-Daemon acts as a intermediary between external services and the simulator and provides a single point of entry for all simulator tasks. It allows NCS to focus on being as fast as possible by handling the organizational outside the simulator codebase. The tools that surround the simulator include a Python interface, and a web interface. These services interact with NCS-Daemon to gain access to the simulator.

the simulator. Additionally, a method of transport is needed to carry information to the daemon from the clients and vice versa. As discussed earlier, one of the best methods to transfer textual data over a network is the HTTP protocol. HTTP is ubiquitous and libraries exist for using it in virtually every programming language, making it an easy choice for implementing the communication fabric between the daemon and the clients. Finally, HTTP is a client-server-based protocol, which suits the goal of the project perfectly.

There are a myriad of different libraries and frameworks to choose from to create a server using HTTP as the underlying protocol. Since the low-level simulator functions are exposed using a thin Python layer, it makes sense to choose a Python-based HTTP library to handle the connections to the simulator. Several popular libraries exist as Python modules: the most popular of which is Django[3]. Django is a so-called full-stack web application framework, as it has facilities for creating database entities, an administrative interface, form-validation, and many other features that make it advantageous to use as a general web application framework. NCS-Daemon, however, would not make use of the vast majority of these features, making Django superfluous for the requirements of the application. Another web framework called Flask[16] provides a more minimal set of functionality. It provides the ability to designate *routes* that call certain Python functions when a request comes in to a certain URI, making it very easy to implement a REST interface. Flask also has a host of extensions that help extend the functionality of the application including WebSockets and integration with MongoDB, a popular JSON-based NoSQL database. Because of its simplicity and extensibility, we chose to implement the daemon using the Flask web application framework. A simple "Hello, world" Flask application can be found in Figure 3.2.

To design NCS-Daemon as a RESTful service, a number of resources needed to be defined. The first resource is the *simulator* resource. The *simulator* resource refers to the current status of the simulator. A GET request to this resource will provide information on the status of the simulator, a POST request containing simulation data will attempt to start a new simulation, and a DELETE request will attempt to

```python
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run("localhost", 8000)
```

Figure 3.2: This Flask application serves the plain text "Hello World" when an HTTP client sends a GET request to the '/' route.

stop the currently running simulation. The next resource is the *simulation* resource. The simulation resource refers to a specific simulation created by a user. This resource allows the GET and DELETE methods. The GET method returns information about the simulation specified, including log information, reports related to the sim, and the simulation object that was run by the simulator. A DELETE request to a specified simulation resource will delete that simulation, as well as any data associated with it. The next resource is the *report* resource. The *report* resource requests the contents of a report generated by a previous simulation. It accepts the GET and DELETE methods, which instruct the server to provide the report, and remove the report from disk respectively. In the event a report was specified as a streaming report meant for WebSockets, the resource will return an error. The final resource is the *login* resource. This resource provides an endpoint for users to authenticate with the server which is an important component of NCS-Daemon.

When implementing any network-based application, security is always a matter of concern, especially when the application is facing the Internet. Preventing unauthorized use of the NCS simulator and protecting user information are the main problems with regard to application security. To prevent unauthorized use of the simulator a login system was created. To use the simulator, the user must have an account on the server. Next, before any interaction with the daemon, the user must first send a POST request to the */login* route on the server, supplying their username and password. In

response, the server sends an authorization token specific to that user. Each request to the server must contain this authorization token in the HTTP headers or the server will reject the request as unauthorized. By providing the token to users, it prevents their credentials from being passed with each request. If an attacker were to begin intercepting communications between the client and server after the user has logged in, he would only gain the access token. The attacker would still be able to gain access to the simulator for a certain time, but the user's credentials would remain secret and could not be used to gain access to external accounts such as email and bank accounts registered using the same password. To completely protect access to the simulator and the users credentials, the connection to the server must be made over HTTPS using SSL or TLS, which are used to create a completely secure connection that cannot be decoded by other parties during or after transmission.

NCS-Daemon is implemented as a RESTful interface and as such exposes these resources to clients in the form of routes. In addition to the traditional RESTful routes exposed by the application, NCS-Daemon also has two different WebSocket routes that are also available for streaming live data to the client provided the client has a library to interact with the WebSocket protocol. The routes exposed by NCS-Daemon as well as their descriptions and allowed methods can be found in Table 3.1.

NCS-Daemon is designed to be fully testable and is designed with good software engineering practices. The project repository is hosted on GitHub and is integrated with the Travis Continuous Integration service. This service runs the project's tests each time a developer attempts to update the repository and alerts the user when their changes break the tests. Additionally, the service outputs the code coverage of the tests, which measure the amount of code in the repository was run during the tests, which keep track of how effective the testing suite is. Ideally, this number should be as close to 100% as possible. The software class diagram for NCS-Daemon can be found in Figure 3.3 and the GitHub repository showing the NCS-Daemon README with the testing and coverage badges can be found in Figure 3.4.

Table 3.1: This table shows the routes, methods and descriptions for NCS-Daemon's RESTful interface.

Route	Methods	Resource Description
/login	POST	The login route accepts requests containing a username and password and responds with an access token provided the credentials were correct.
/simulator	GET POST DELETE	The simulator route provides a resource for the state of the simulator. A GET request returns information about the current status of the simulator. A POST request will instruct the daemon to start a simulation. Finally, a DELETE request instructs the daemon to cancel the current simulation.
/simulation/:id	GET DELETE	The simulation route provides a resource for past simulations. A GET request returns information about the simulation requested. A DELETE request instructs the daemon delete the specified simulation.
/report/:id	GET	The report route provides the report generated by a simulation, including whether it can be streamed through a WebSocket.
/report_stream/:id	WebSocket	The report_stream route is used for streaming live report data over a WebSocket interface.
/geometry_stream	WebSocket	The geometry_stream WebSocket can be used to stream the locations and geometry of neurons and connections for a realtime visualization.

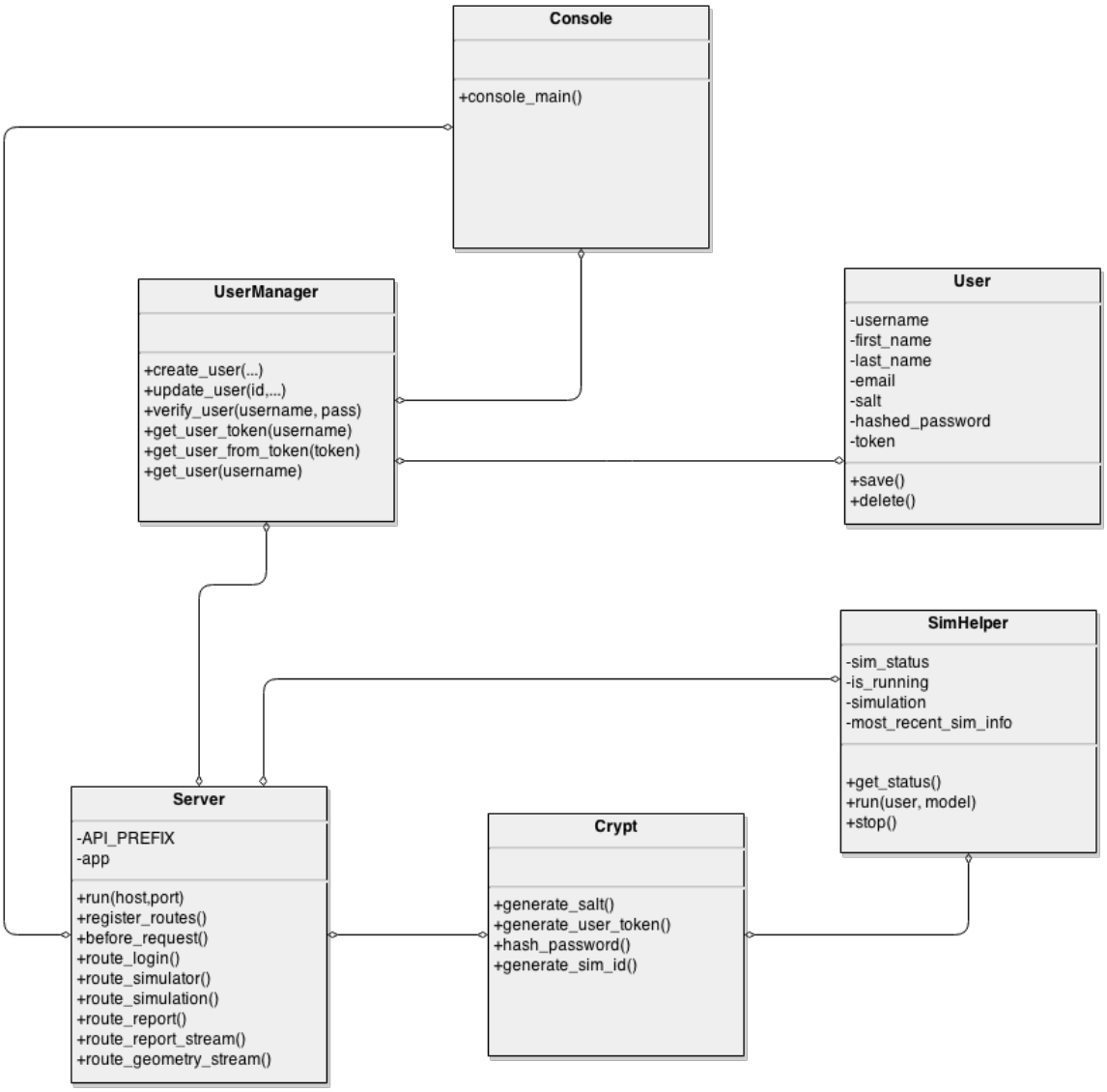

Figure 3.3: This is the NCS-Daemon's class diagram with testing components omitted.

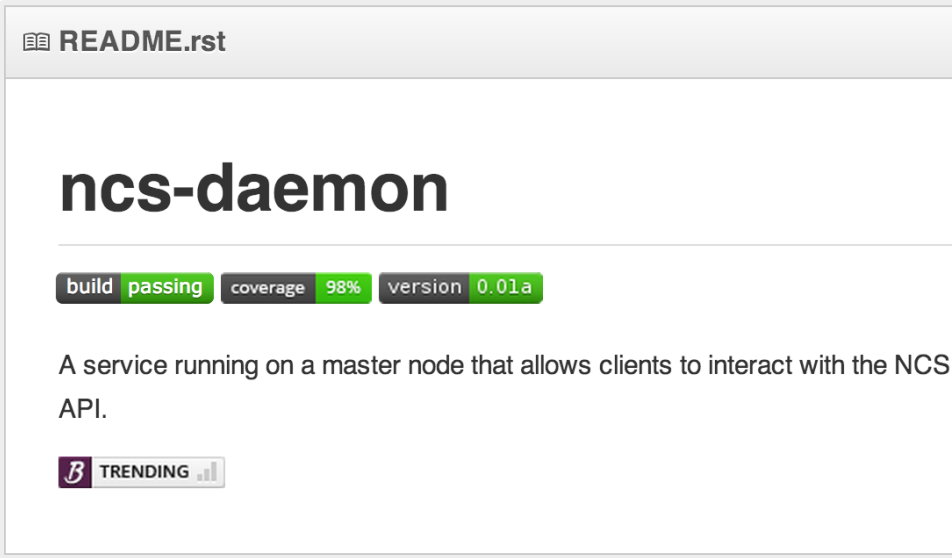

Figure 3.4: The README file for NCS-Daemon contains badges that link to the current testing state of the project, including if its tests are passing, and what the code coverage percentage is.

3.3 Simulation Concepts & Transfer Format

NCS simulations are defined as a hierarchical structure of groups and neuron groups as well as several other entities each with their own parameters. These entities include neurons, synapses, stimuli, reports, neuron_aliases, and synapse_aliases. Neuron entities contain the model used to calculate whether or not a neuron will fire on a particular time tick. Neurons can be of several types, each with their own set of parameters that affect the behavior of the neuron. Currently, the simulator supports Leaky-Integrate-and-Fire (LIF), Hodgkin-Huxley, and Izhikevich models for neurons. Synapses define the properties of connections between neurons including synaptic learning and plasticity. Stimuli define where electrical current will be applied during the simulation and how it should be delivered. Reports define which properties and which neurons or synapses should be collected and either streamed over a WebSocket or written to disk during the simulation. Lastly, neuron_aliases and synapse_aliases provide a way to refer to large collections of neurons or synapses under a single name

instead of all their collective names, reducing the work required for referencing large groups or neurons or synapses.

Since NCS doesn't have native support for higher-level constructs, a system needed to be developed allows for users to organize the groups of cells more effectively. To accomplish this, we created a hierarchical system of groups as a way to structure the model. A group may contain multiple types of entities, the first of which are subgroups. Subgroups are instances of other groups placed within a larger group that have a label that distinguishes themselves from other subgroups in the same group. Groups can also contain neuron groups, which are simply collections of neurons of the same type that act as a single unit. Groups are also where connections are defined. A connection is created between two neuron groups with a synapse type and a certain probability of connection between them. Finally, aliases are created within groups. Aliases allow a single label to refer to multiple objects. For example, if there are two neuron groups labeled "A" and "B", then a neuron alias named "AB" could be created that refers to both of these subgroups. This system allows users to better organize their models, and create larger models more effectively than creating numerous unorganized neuron groups. This system of organization is illustrated in Figure 3.5.

In order to transfer and store simulations, some kind of textual representation for the simulation model and its entities must be created. Due to the hierarchical nature of the simulation model, with simulations possibly containing many of the same group type, the tree of groups could grow at a sizable rate if this structure were fully expanded into JSON. This makes the size of the files much larger, which is bad for both transporting the JSON over a network or storing it on disk. Additionally, it makes decoding the simulation once it reaches the daemon more complicated and slow due to the redundant information and very deep model structure. To combat this, we have implemented a normalized approach to storing the data in a textual format, akin to the concepts of a relational database. Instead of expanding the simulation tree into its full form, the tree is traversed and groups are added to a list of groups. Inside the

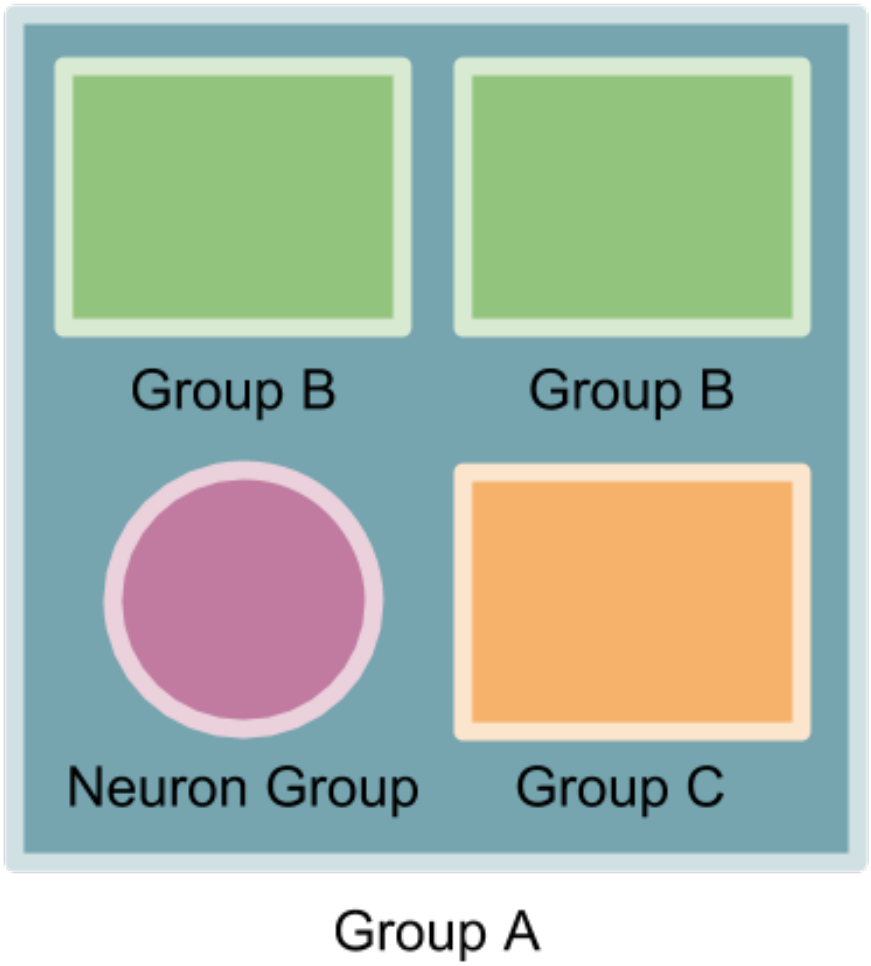

Figure 3.5: This diagram shows a visual representation of a group in NCS. Group A contains two instances of Group B, a neuron group, and another group, Group C. A JSON representation of a group can be found in Figure 3.8.

```
{
    "top_group": "5728ec1d381bad25744bc54c42ba425",
    "neurons": [

    ],
    "synapses": [

    ],
    "stimuli": [

    ],
    "reports": [

    ],
    "groups": [

    ],
    "neuron_aliases": [

    ],
    "synapse_aliases": [

    ]
}
```

Figure 3.6: The JSON transfer format without any entities defined. The *top_group* property specifies the id of the topmost group in the simulation heirarchy. When a client submits a simulation to NCS-Daemon the arrays would be populated with data.

groups, rather than expanding subgroups into their corresponding JSON, we maintain a reference to the other group in the form of a unique identifier, that is generated randomly. This saves space by reducing redundancy that would be common in many models. Since the other entities are not defined in a hierarchical fashion, they do not need to be normalized and can be added to a list with other entities of the same type. A template showing the simulation transfer format with normalized groups can be found in Figure 3.6.

In addition to their specifications, entities also contain metadata that describes information about the entity, such as its type, name, description, author, institution, and author's email. When stored on disk or in a database such as MongoDB, this metadata can be used to determine where a model was created, who it was created

```
{
    "_id": "c4f5ea85f4c5ae86c4ebc4a854",
    "entity_type": "neuron",
    "entity_name": "neuron_izh_1",
    "description": "This is an extended description of the entity",
    "author": "Nathan Jordan",
    "author_email": "njordan@cse.unr.edu",
    "specification": {
        "type": "izhikevich",
        "a": 0.02,
        "b": 0.2,
        "c": -65.0,
        "d": 8.0,
        "u": -12.0,
        "v": -65.0,
        "threshold": 30
    }
}
```


Figure 3.7: An Izhikevich neuron model with metadata represented in JSON. The metadata surrounds the Izhikevich parameters contained within the specification object.

by, and which scientific papers it might have been created for or from. This will aid researchers in collaborating on experiments, and keeping models organized. An example of an Izhikevich neuron entity including its metadata can be found in Figure 3.7 and a group entity with subgroups, neuron groups, aliases, connections and metadata can be found in Figure 3.8.

3.4 PyNCS

Now that the server and transfer schema have been defined, there needs to be a convenient way to interact with the server without having to deal with HTTP and the JSON representation during transfer. To address this, we have created a Python library called PyNCS. PyNCS allows users to communicate with an NCS server using an intuitive Python interface. To connect to the server, the user instantiates a *Simulator* object with their username and password. If the library fails to connect to the simulator, an authentication exception is thrown. An example showing the usage of

```
{
    "_id": "df90sahf0sd9ha8sdhf8dhsa",
    "entity_type": "group",
    "entity_name": "BasalGanglia",
    "description": "This is an extended description of the entity",
    "author": "Nathan Jordan",
    "author_email": "njordan@cse.unr.edu",
    "specification": {
        "geometry": {
            "width": 100.0,
            "height": 200.7,
            "depth": 300.3
        },
        "subgroups": [
            {
                "group": "sdfh8ahsdf80has0d89fh0as9dhf",
                "label": "striatum",
                "location": {"x": 123.5, "y": 456.5, "z": 789.5}
            },
            {
                "group": "hf8as0df8asdhf80ahsd80f",
                "label": "pallidum",
                "location": {"x": 2354.5, "y": 456.5, "z": 543.5 }
            }
        ],
        "neuron_groups": [
            {
                "neuron": "hfd80sahs80dhf0shda90fhshdf",
                "label": "gpe",
                "count": 400,
                "geometry": {"width": 100.4, "height": 200.7, "depth": 300.8},
                "location": {"x": 234.9, "y": 456.4, "z": 543.2}
            },
            {
                "neuron": "hasd890fhas80fhas80dhf",
                "count": 30,
                "label": "gpi",
                "geometry": {"width": 100.1, "height": 200.6, "depth": 300.7},
                "location": {"x": 784.4, "y": 926.1, "z": 999.2}
            }
        ],
        "neuron_aliases": [
            {
                "alias": "gpx",
                "labels": ["gpe", "gpi"],
                "aliases": ["an_alias"]
            }
        ],
        "synaptic_aliases": [
            {
                "alias": "sample_synapse_alias",
                "labels": ["GP_synapse"]
            }
        ],
        "connections": [
                {
                    "presynaptic": "gpe",
                    "postsynaptic": "gpi",
                    "probability": 0.5,
                    "synapse": "hfd8nfiodnsa80dbhfbas80df",
                    "recurrent": false
                }
        ]
    }
}
```


Figure 3.8: JSON Group specification with metadata parameters.

```
# Correct
server = Simulator(
    host='ncscluster.example.edu',
    port=8081,
    username='my_username',
    password='my_password'
)

# Throws ConnectionError
server = Simulator(
    host='no_ncsdaemon_here.example.edu',
    port=8081,
    username='my_username',
    password='my_password'
)

# Throws AuthenticationError
server = Simulator(
    host='ncscluster.example.edu',
    port=8081,
    username='my_username',
    password='not_my_password'
)
```

Figure 3.9: This example gives three different scenarios for connections. The first example results in a successful connection, the second example attempts to connect to a host where NCS-Daemon is not running, and the last example attempts to connect to NCS-Daemon with the incorrect credentials.

the simulator via PyNCS can be found in Figure 3.9. Models are created using Python classes that are defined in the PyNCS package. These classes provide a way to create models in an object-oriented fashion without use of dictionaries or less-structured Python types. When creating entity objects, the parameters are checked against a known list of acceptable parameters for that entity as to prevent mistakes when creating a model, and alerting the user to their mistakes before the simulation is run, so they can more easily identify where the error is and fix it quickly. For example, creating an Izhikevich neuron and specifying x as a parameter will throw an exception. An example of this functionality can be found in Figure 3.10.

The Python classes also enforce which parameters are required and optional. This allows users to only specify which parameters that are pertinent to the entity they

```
# This successfully defines an Izhikevich neuron
izh = IzhNeuron(
    a=0.5,
    b=0.5,
    c=0.5,
    d=8.0,
    u=-12.0,
    v=Normal(-65.0, 0.5),
    threshold=30.0
)

# This throws an EntityError because we tried to specify x
izh = IzhNeuron(
    a=0.5,
    b=0.5,
    c=0.5,
    d=8.0,
    u=-12.0,
    x=10.0,
    threshold=30.0
)
```

Figure 3.10: This figure shows how an Izhikevich neuron would be declared in PyNCS. The first example successfully creates an Izhikevich neuron type, whereas the second example results in an exception.

are creating without specifying extra parameters that are empty. This makes PyNCS code more readable by reducing a level of verbosity, makes the code faster to write, and keeps code size to a minimum.

Once the model and other entities have been created, the user must create a *Simulation* object. The simulation object encapsulates a single instance of a simulation. The most important service the *Simulation* object performs is maintaining references to the reports generated by the simulation. These reports can be downloaded to the client from the server for further processing, visualization or anything else the user might wish to use the simulation data for. They are not downloaded automatically due to the possibility that the reports can become quite large dependant on the number of neurons or synapses being reported on and the frequency at which data on these entities is collected. The *Simulation* object can also be used as a WebSocket client to stream realtime data from the server. Lastly, the simulation object contains diagnostic information and log data related to the simulation so the user can view and address any potential problems that may arise during a simulation. The *Simulation* object can only be used for one simulation; to create and run another simulation, an additional *Simulation* object must be create. An example of a *Simulation* object being created and run can be found in Figure 3.11.

3.5 Web Interface

While the PyNCS interface provides a way for users with knowledge of programming (specifically Python programming) to create and run simulations programmatically, this is a skill set that many potential neuroscientists using the NCS simulator may not have or want to exercise. To address the needs of all neuroscientists, a more user-friendly and less-technical interface is needed to interact with the brain simulator. Simulators such as NEURON and GENESIS discussed in the background section use old interfaces developed for native windowing environments like X11. The problem with these interfaces is that they are now antiquated, as newer technologies exist to create these environments for modern operating systems. Additionally, they must be

```
izh = IzhNeuron(
    a=0.5,
    b=0.5,
    c=0.5,
    d=8.0,
    u=-12.0,
    v=Normal(-65.0, 0.5),
    threshold=30.0
)
syn = FlatSynapse(
    delay=10.0,
    current=60.0
)
nrn_grp1 = NeuronGroup(
    neuron=izh,
    count=30,
    label="izh1",
    geometry=Geometry(),
    location=Location()
)
nrn_grp2 = NeuronGroup(
    neuron=izh,
    count=50,
    label="izh2",
    geometry=Geometry(),
    location=Location(),
)
conn = Connection(
    presynaptic="izh1",
    postsynaptic="izh2",
    probability=0.5,
    synapse=syn
)
grp = Group(
    entity_name="my_group",
    subgroups=[],
    neuron_groups=[nrn_grp1, nrn_grp2],
    neuron_aliases=[],
    synapse_aliases=[],
    connections=[conn]
)
stim = RectCurrentStimulus(
    amplitude=3.0,
    width=2,
    frequency=10,
    probability=0.6,
    time_start=0,
    time_end=1,
    destinations=["my_group:izh1"]
)
rpt = Report(
    report_method=Report.METHOD_FILE,
    report_type=Report.TYPE_NEURON,
    report_target=[nrn_grp1],
    probability=0.5,
    time_start=0.0,
    time_end=1.0
)
sim = Simulation(
    top_group=grp,
    stimuli=[stim],
    reports=[rpt]
)
server = Simulator(
    host='ncscluster.example.edu',
    port=8081,
    username='my_username',
    password='my_password'
)

if server.get_status() == Simulator.IDLE:
    server.run(sim)
```

Figure 3.11: This figure shows how a simulation would be created and run in PyNCS. The sim object is created using the top-level model and a list of stimuli and reports. The Simulator object is created using a hostname and port as well as a username and password which establishes a connection to NCS-Daemon. Once a connection has been established, the status of the simulator can be checked. If the simulator is idle, we can run the created simulation.

run on the users computer directly, forcing them to install software on their machine which may not be desirable. To address these problems, we have created a web-based interface that acts as a client to the NCS-Daemon server and allows neuroscientists to create simulations using only a web browser.

To create the web interface we took advantage of a couple tools that would greatly reduce design time. Firstly, we used the Bootstrap front-end framework to create a consistent look and feel, and save a lot of development time by not having to develop custom UI widgets. Secondly, we made use of AngularJS as the Javascript framework powering the page. Angular allows easy integration with the DOM without having to directly manipulate it as well as providing a way to easily interact with the RESTful interface of NCS-Daemon. Currently, there are five tabs in the NCS Web Interface: model builder, sim builder, reports, model database, and robot simulator.

The model builder, shown in Figure 3.12, is a graphical way for users to create their models. The design of the model builder posed some particularly difficult UI design challenges. While some entities like neurons are single-layer entities, group entities form a hierarchical structure that could be several layers deep. Users need to be able to navigate efficiently through these levels and do so without feeling like the UI has become too cluttered. We felt that a tree-type interface would lead to too much clutter and greatly degrade user experience. To develop a more effective interface, we took some queues from file explorers of different operating systems. File explorers also work on a hierarchical data set (a filesystem is a tree) and have been developed to make it easy for users to navigate around the filesystem and perform operations on it. A particular example of this is the Windows Explorer interface that uses breadcrumbs to show the user where they are and a main content area where the content of the location is displayed.

In our design, we implemented a system in which breadcrumbs are used to navigate the tree and keep track of where the user is in the hierarchy (similar to file explorer) and a content area in the form of a panel updates as the user navigates. As the user moves through the groups they are creating, links are created and removed along a

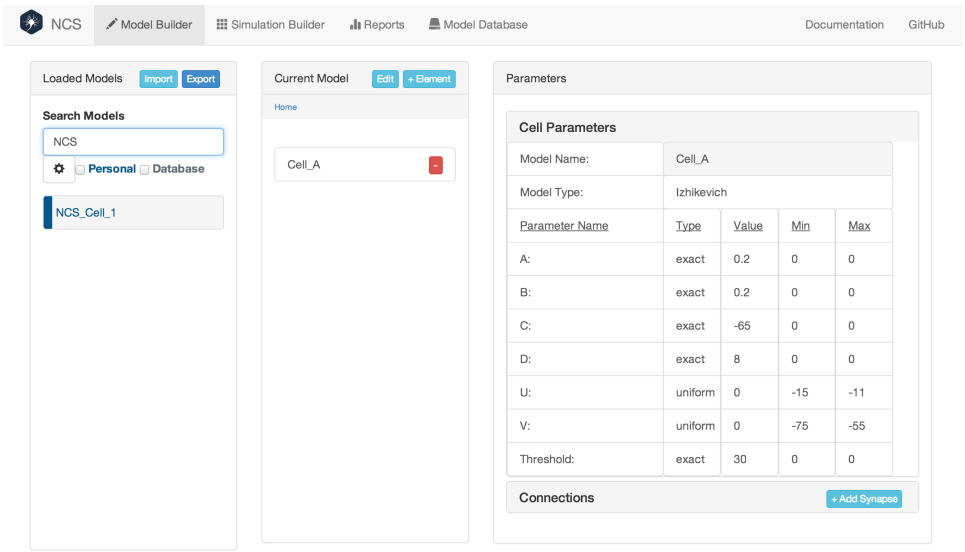

Figure 3.12: The model builder is organized into three sections split vertically across the page. The leftmost panel contains models that the user has built or loaded from a file or database. The second panel displays entities that are within the currently active group. The rightmost panel displays parameters of a selected entity. In this case, it is displaying the Izhikevich parameters for Neuron_1.

top bar that indicates where they are in the model. If they want to move to a different position in the hierarchy, they simply click the link corresponding to the name of the group they want to move to and the content area updates accordingly. In the content area is a list of subgroups the user can further navigate into. When the user selects one of these subgroups a new subgroup is created, the user is shown the contents of the new subgroup (when created it is empty) and a breadcrumb is added to the list.

The rightmost panel is where model specifications are created. This panel lets the user modify the properties of entities they have selected. For example, if the user were to select a neuron from the model selection list on the left parameters related to the neuron would be editable to the user on the right panel, such as the voltage threshold or resting potential. Values updated here are propagated to the back-end model and are updated there.

Once the user has created or imported the model they wish to execute in the simulator, they then switch to the sim builder tab. The sim builder tab, shown in

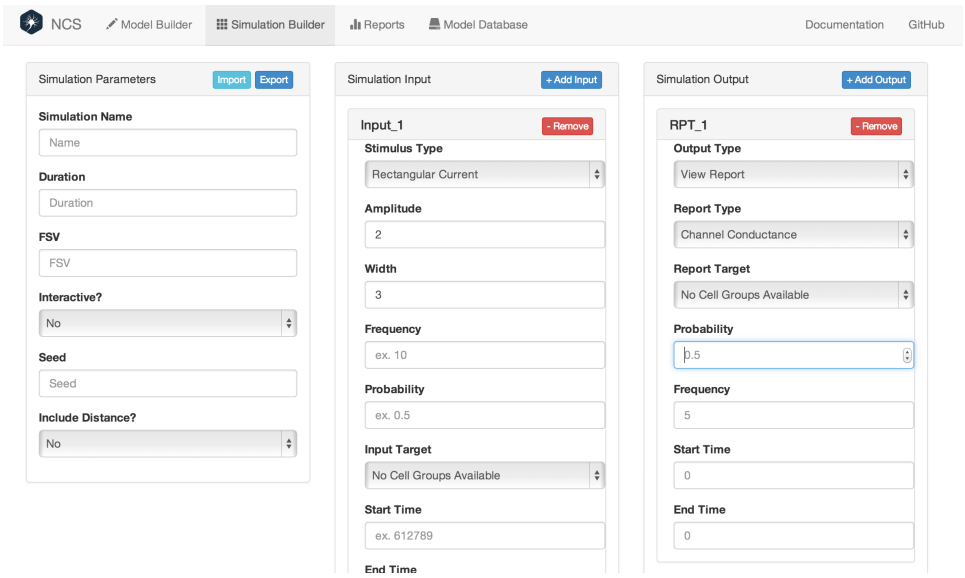

Figure 3.13: The sim builder provides the functionality for adding reports and stimuli to the simulation. The leftmost panel changes global simulation parameters, the middle panel controls stimuli, and the rightmost panel controls reports.

Figure 3.13 allows the user to create interactions with the model that they have built and actually create the simulation. Firstly, the user should create at least one stimulus to inject into a select group of cells within the model to bring it to life. The stimulus can be generated from a few different sources, including a file-based stimulus, or a socket-based stimulus. The socket-based stimulus provides external programs the ability to provide input to the simulation. Once the stimuli are created, the user can create reports for the model. Reports are the way data can be extracted from the simulation, and several metrics can be reported on, including synaptic plasticity to membrane potential. Once the user is finished with these tasks, the simulation can be run from this tab.

To view the results of a simulation, the user navigates to the report tab which is shown in Figure 3.14. The reports tab has the ability to view graphs of simulation data. This data can come from either a static report after the simulation has finished, or be streamed from NCS-Daemon and the simulator to the browser via a WebSocket interface. This feature gives researchers the ability to view simulation data in real-

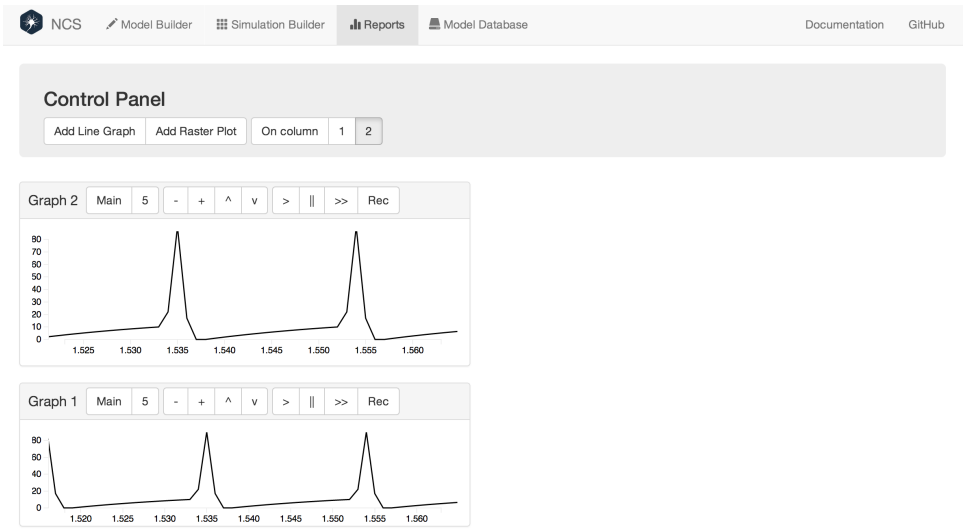

Figure 3.14: The reports tab allows users to view static and realtime data about a simulation. The graphs can be saved as high-quality SVG images for presentations or reports.

time. Graphs can be built from the data and exported to an SVG (vector) format for inclusion in high quality publications and posters. An example of one of these graphs is shown in Figure 3.15.

The robot simulation tab is still heavily in development. It allows simulation output to control the actions of a virtual robot in a 3D world and allow the observations of the robot on its environment to become inputs to the simulator. The simulator could

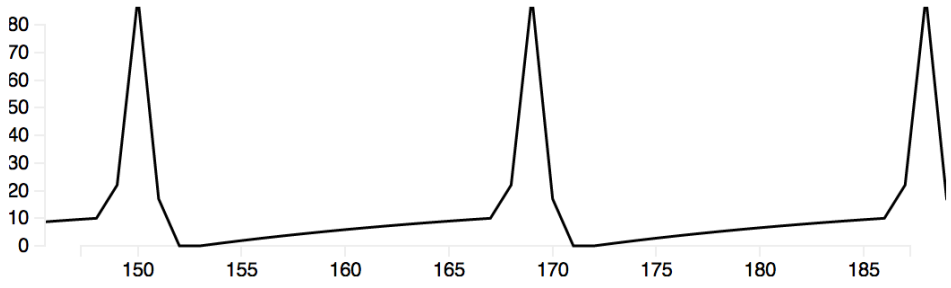

Figure 3.15: The reports generated in the reports tab can be exported as high-quality images for use in publications or in presentations.

be used to demonstrate the decision making ability of a brain model with the actions of the robot in the virtual environment, such as navigating a maze or performing actions when recognizing colors or patterns.

Chapter 4

Conclusions and Future Work

4.1 Conclusion

To extend the NCS brain simulator, we have created a number of tools that allow users to interact with the simulator in a variety of ways. NCS-Daemon provides a base level of interaction, handling the lower-level operations on the simulator. It introduces a the client-server architecture to the ecosystem of neural simulators, allows for web-based interaction, organizes simulation data, and provides a way to stream real-time data from the simulation to clients.

PyNCS creates a way for researchers to use the Python language to create models and interact with the simulator. The library provides features such as parameter checking that make it more user friendly and easier to debug than the minimalistic Python layer. This language binding also gives easy access to a number of scientific tools such as SciPy or NumPy that might be used to process the resulting data.

Finally, the web interface provides a method to interact with the simulator without requiring any knowledge of programming. The interface allows users to build, run, analyze, and visualize neural simulations in a single web application. This tool, as well as PyNCS and NCS-Daemon, bring new technologies to the neural simulator landscape that has long been plagued with aging technologies, and allow researchers more effective ways of performing experiments and creating scientific breakthroughs.

4.2 Future Work

NCS and its surrounding tools provide a basic interface for working on brain simulations, but there is always more that could be done. Implementing a job queue for simulation queuing would be advantageous to waiting until another user is done to start a simulation. One could simply create the simulation and submit it and NCS-Daemon would run it when it reaches the front of the queue and notify the user in some way when the simulation is complete.

Another possible addition would be an external robotics interface that would allow the simulator to interact with a ROS robot such as a PR2. By using ROS it would also allow a connection with a virtual robot in the ROS environment, or physical robot implementing the ROS software. With this ability, researchers could apply their models to real-world robotic scenarios.

A web-based realtime visualization of the simulation is currently being developed using WebGL that allows users to visualize the model they created while the simulation is running and see individual neurons firing[7]. The geometry would be streamed via a WebSocket dedicated to this purpose. The user can navigate around the model, change colors, and turn parts of the model on and off.

Interfaces can always be improved upon, and the NCS web interface is no different. As web technologies and interface design change and evolve, there will be a constant need to rework existing designs. The NCS web interface could use additional styling, better UI design, and additional features to make it more viable as a tool for building an simulating models, as well as analyzing the results.

Lastly, while Python was chosen to implement the client library for NCS (PyNCS), it is far from the only language option for interacting with the simulator. Since NCS-Daemon is implemented as a HTTP-based service, it can be easily expanded to other languages. Languages like Ruby would make an excellent candidate for another way to interact with the simulator.

Bibliography

[1] Angularjs: Developer guide: Conceptual overview. https://docs.angularjs.org/guide/concepts. (Visited on 04/28/2014).

[2] Bootstrap. http://getbootstrap.com/. (Visited on 04/28/2014).

[3] Google trends: python django, python flask, python bottle, python pyramid, jan 2004-jan 2014. http://www.google.com/trends/explore#q=python%20django%2C%20python%20flask%2C%20python%20bottle%2C%20python%20pyramid&date=1%2F2004%20121m&cmpt=q. (Visited on 05/07/2014).

[4] James M Bower and David Beeman. *The book of GENESIS: exploring realistic neural models with the GEneral NEural SImulation System*. Electronic Library of Science, The, 1995.

[5] Tim Bray. The json data interchange format. RFC 4627, JSON Working Group, November 2013.

[6] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M Bower, Markus Diesmann, Abigail Morrison, Philip H Goodman, Frederick C Harris Jr, Milind Zirpe, Thomas Natschlager, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew P. Davison, Sami El Boustani, and Alain Destexhe. Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of computational neuroscience*, 23(3):349–398, 2007.

[7] Justin E Cardoza, Alexander K Jones, Denver J Liu, Roger V Hoang, Devyani Tanna, Laurence C Jayet Bray, Sergiu M Dascalu, and Frederick C Harris Jr. Design and implementation of a graphical visualization tool for ncs. In *Proceedings of The 2013 International Conference on Software Engineering and Data Engieering*, pages 37–43, Los Angeles, CA, 2013. SEDE.

[8] Nicholas T. Carnevale and Michael L. Hines. *The NEURON Book*. Cambridge University Press, New York, NY, USA, 2006.

[9] Kristina Chodorow. *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2013.

[10] Peter Bacon Darwin and Pawel Kozlowski. *AngularJS web application development*. Packt Publishing, 2013.

[11] Ian Fette and Alexey Melnikov. The websocket protocol. RFC 6455, Internet Engineering Task Force (IETF), December 2011.

[12] Roy Fielding. Representational state transfer. *Architectural Styles and the Design of Netowork-based Software Architecture*, pages 76–85, 2000.

[13] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. RFC 2616, Network Working Group, June 1999.

[14] Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[15] Dan FM Goodman and Romain Brette. Brian simulator. *Scholarpedia*, 8(1):10883, 2013.

[16] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc, 2014.

[17] Roger V Hoang, Devyani Tanna, Laurence C Jayet Bray, Sergiu M Dascalu, and Frederick C Harris Jr. A novel cpu/gpu simulation environment for large-scale biologically realistic neural modeling. *Frontiers in neuroinformatics*, 7, 2013.

[18] Reuven M Lerner. At the forge: Twitter bootstrap. *Linux Journal*, 2012(218):6, 2012.

[19] Adam Lith and Jakob Mattsson. Investigating storage solutions for large data-a comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data. 2010.

[20] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: A case study. *Caine*, 9:157–162, 2009.

[21] Terrence Joseph Sejnowski, Christof Koch, and Patricia S Churchland. Computational neuroscience. *Science*, 241(4871):1299–1306, 1988.

[22] Corey M Thibeault, Roger V Hoang, and Frederick C Harris Jr. A novel multi-gpu neural simulator. In *BICoB*, pages 146–151, 2011.

[23] Mihalis Tsoukalos. Using django and mongodb to build a blog. *Linux Journal*, 2014(238):3, 2014.

[24] Wikipedia. Client-server model - wikipedia the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Client%E2%80%93server_model&oldid=605705936, 2014. [Online; accessed 29-April-2014].