

University of Nevada, Reno

A Centralized Service for Accessing the NCS Brain Simulator Through a Web Interface

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

by

Christine Maria Johnson

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

December, 2015



THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

CHRISTINE JOHNSON

entitled

**A Centralized Service for Accessing the NCS Brain
Simulator Through a Web Interface**

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

Dr. David Zeh, Graduate School Dean

December 2015

Abstract

UNR's Neocortical Simulator (NCS) is a large scale brain simulator that allows neuroscientists to run simulations with created brain models and receive output data generated by those simulations. Initially, NCS could only be accessed with text files and Python script files. The NCS web interface was recently developed to provide neuroscientists with a visual tool for creating brain models, setting simulation parameters, and analyzing simulation output data. The use of the web interface requires a protocol for communicating with NCS, as well as the management of a database used to store brain models and user accounts. This thesis presents a centralized service for managing communication between the web interface and NCS and between the web interface and the database. The service also has features for performing conversions between the Python scripts used to define simulation parameters and simulations created with the web interface, and streaming simulation data in real time while queuing any data if it is not able to be received by the user. The implementation of this service has provided the necessary link for data exchange between the web interface and NCS, and allowed for the addition of features to the web interface that will expectantly enhance user experience.

Dedication

For Sphinx and Phoenix.

Acknowledgments

I would like to thank Dr. Fred Harris, Dr. Sergiu Dascalu, and Dr. Yantao Shen for being on my committee, with special thanks to Dr. Fred Harris for giving me the opportunity to do research in the HPCVIS lab. I would like to thank Cameron Rowe for his work on the NCS web interface, and everyone else in the HPCVIS lab for their collaboration and their company. I would also like to thank Eric Klukovich for being the most dependable partner and friend anyone could ever ask for.

Contents

Abstract	i
Dedication	ii
Acknowledgments	iii
List of Tables	vi
List of Figures	vii
1 Introduction	1
2 Background and Related Work	3
2.1 Neuroscience	3
2.1.1 Overview	3
2.1.2 Neurons and Channels	3
2.1.3 Synapses	5
2.1.4 Stimulus	5
2.2 Computational Neuroscience	5
2.2.1 Overview	5
2.2.2 Models	6
2.2.3 Simulators	7
2.3 Neocortical Simulator	7
2.3.1 Overview	7
2.3.2 Simulation Composition	8
2.3.3 PyNCS	9
2.3.4 NCS Web Interface	9
2.4 Centralized Services	13
2.4.1 Overview	13
2.4.2 Client-server Architecture	15
2.4.3 Daemon Processes	15
2.5 Libraries and Frameworks	15
2.5.1 Overview	15
2.5.2 Python	15
2.5.3 Bcrypt	16
2.5.4 Protocol Buffers	16

2.5.5	Twisted	17
2.5.6	RabbitMQ	20
2.5.7	JSON	22
2.5.8	MongoDB	23
3	Design	24
3.1	Overview	24
3.2	Service Requirements	24
3.2.1	Functional Requirements	24
3.2.2	Non-functional Requirements	24
3.3	Use Case Modeling	26
3.3.1	Overview	26
3.3.2	Detailed Use Cases	27
3.4	Architecture	29
4	Implementation	31
4.1	Overview	31
4.2	Database Management	31
4.2.1	Overview	31
4.2.2	Structure	31
4.2.3	TxMongo Functions	35
4.3	Twisted Services	36
4.3.1	Overview	36
4.3.2	Add New User Service	36
4.3.3	Authentication Service	37
4.3.4	Data Proxy Service	41
4.4	Error Handling	43
4.4.1	Exceptions	43
4.4.2	Invalid Requests	43
4.5	Results	44
5	Conclusion and Future Work	51
5.1	Conclusion	51
5.2	Future Work	52
5.2.1	NCS Daemon Enhancements	52
5.2.2	NCS Enhancements	53
5.2.3	Web Interface Enhancements	53
	Bibliography	55

List of Tables

3.1	The NCS daemon functional requirements.	25
3.2	The NCS daemon non-functional requirements.	25

List of Figures

2.1	A diagram of neuron structure, from [32].	4
2.2	An example of a simple PyNCS Python script.	10
2.3	A screenshot of the NCS web interface brain model builder tab.	11
2.4	A screenshot of the NCS web interface simulation builder tab.	12
2.5	An example graph of NCS simulation report data, from [33].	13
2.6	A screenshot of the NCS web interface virtual robot tab while a simulation is running, from [16]	14
2.7	An example of a Protobuf data structure definition.	17
2.8	A diagram of multi-task program models, from [17].	18
2.9	An example of a Twisted log of an echo server starting up, echoing one message, and terminating, from [6].	21
2.10	A diagram of a direct exchange with Queue 1 bound with the binding key <i>orange</i> and Queue 2 bound with the binding keys <i>black</i> and <i>green</i> , from [24]	22
3.1	A use case diagram of the NCS web interface.	26
3.2	A system architecture diagram of the daemon and how it interacts with the other components.	30
4.1	An example of the database structure where { } items represent BSON objects and [] items represent lists of BSON objects.	32
4.2	An example of a document in the <i>users</i> collection.	33
4.3	An example of a model document.	33
4.4	An example of a simple model JSON.	34
4.5	A diagram of the Add New User Service interacting with the other system components.	37
4.6	A diagram of the Authentication Service interacting with the other system components.	38
4.7	A diagram of the Data Proxy Service interacting with the other system components.	42
4.8	An example of success and failure responses to requests made by the web interface.	44
4.9	A screenshot of the NCS web interface invoking the <i>add new user</i> request.	45
4.10	A screenshot of the NCS web interface invoking the <i>login</i> request.	45

4.11	A screenshot of the NCS web interface model builder tab. The cell models displayed in the pane on the left are retrieved by invoking the <i>get models</i> request.	46
4.12	A screenshot of the NCS web interface invoking the <i>Python script to JSON</i> request.	47
4.13	A screenshot of the NCS web interface invoking the <i>JSON to Python script</i> request.	47
4.14	A screenshot of the NCS web interface invoking the <i>save model</i> request.	48
4.15	A screenshot of the NCS web interface invoking the <i>undo model save</i> request.	49
4.16	A screenshot of the NCS web interface invoking the <i>launch simulation</i> request.	50
4.17	A screenshot of the NCS web interface reports tab. The data displayed for each report is received from the Data Proxy Service.	50

Chapter 1

Introduction

Computational Neuroscience is a fast growing field, especially with regards to using brain simulation to observe how the human nervous system responds to different stimulus types. Brain simulation, as opposed to experiments using actual cells, has become popular because it offers a flexible method of performing large scale experiments. Typical brain simulations use brain models that consist of various spiking neuron models and synapse models. The simulations use neuron spiking functions to describe a mathematical relationship between stimulus and the probability of neurons spiking in voltage.

A variety of brain simulation tools exist, including the NeoCortical Simulator (NCS), a large scale brain simulator developed at the University of Nevada, Reno. NCS is capable of running simulations with various parameters and brain models created by users, and reporting neuron spiking states and membrane voltages while a simulation is running. Alongside the actual simulator, other components have been developed to broaden the efficiency and usefulness of NCS. These components are designed to allow neuroscientists with any level of programming skill the ability to create brain models, configure simulation parameters, and view the data generated by a simulation as raw data or visually in the form of graphs and simulated environments. The NCS web interface is a visual tool for creating simulations and analyzing simulation output. Contrary to the Python script files, running simulations with the web interface requires no programming knowledge. The development of the web interface introduced networking and communication requirements for exchanging data

with NCS, and necessitates database usage for storing user accounts and created brain models.

This paper presents the NCS daemon, a centralized service for managing user accounts and abstracting the communication protocols from the web interface, NCS, and a database. Since the intent of the web interface is to serve as a user-friendly method of accessing NCS, the daemon also offers functionality to assist with conversions between legacy methods of running simulations and simulations created with the web interface. Additionally, it provides a method of asynchronous data transfer of simulation output data between NCS and the web interface.

The remainder of this document is structured in the following manner. Chapter 2 will cover the neuroscience background, the NCS system components deployed and under development, and the libraries and frameworks used for the implementation of the daemon. Chapter 3 discusses the service requirements of the daemon and how it interacts with the other components of the system. Chapter 4 describes specifically how the daemon was developed. Chapter 5 gives a summary of the presented work and ideas for future enhancements of the daemon, NCS, and the web interface.

Chapter 2

Background and Related Work

2.1 Neuroscience

2.1.1 Overview

Neuroscience is the study of the cells in the nervous system, which consists of the brain, spinal cord, and nerves of the body [34]. Studying these cells gives insight on how the nervous system processes information and reacts to received information. Although neuroscience is a broad subject with extensive amounts of specific topics, only the basic functional components of the brain will be discussed in this section solely for the purpose of understanding brain simulation. A diagram of neurons, channels, and synapses is depicted in Figure 2.1.

2.1.2 Neurons and Channels

The human brain has approximately 100 billion *neurons*. The neuron is considered the fundamental structural unit of the brain. A neuron receives input through *dendrites* to the *soma* (cell body), and sends output to neighboring neurons through the *axon*. *Excitatory Postsynaptic Potential* is input to a neuron that is the output of another neuron [21].

Neurons have an impermeable cell membrane, but they have *ionic channels* that allow charge to flow in and out. The ionic channels are gated, meaning they open and close depending on conditions such as membrane voltage, the presence of chemicals, the presence of pressure, etc. When the Excitatory Postsynaptic Potential is high

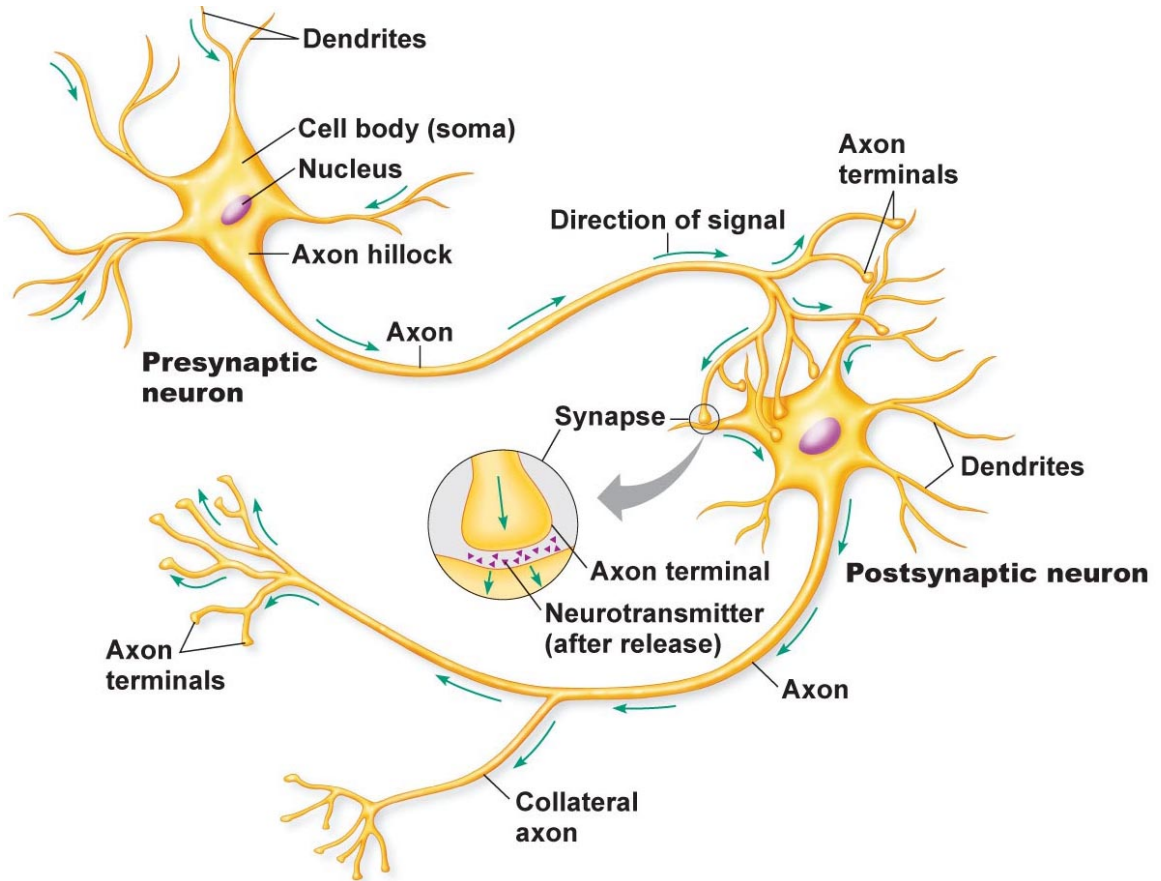


Figure 2.1: A diagram of neuron structure, from [32].

enough, it raises the cell membrane potential causing the opening of *voltage-gated ion channels* and a change in the concentration of ions. The augmented ion concentration will either cause *depolarization*, a positive change in voltage, or *hyperpolarization*, a negative change in voltage. A drastic enough depolarization is what causes the neuron to fire an *action potential*, or spike in voltage [21].

2.1.3 Synapses

The human brain has approximately 100 trillion *synapses*, which are the basis for memory and learning. A synapse is a junction between neurons causing the opening of ionic channels, or the point where the action potential reaches another neuron. A single neuron can have 10,000 synapses on its dendrites and soma. *Excitatory* synapses are those that cause neuron depolarization, and *inhibitory* synapses are those that cause hyperpolarization. Brains learn by repeated firing of one neuron to another, causing *Long-term Potentiation*, an observed increase in Excitatory Post-synaptic Potential, or synaptic strength between two neurons over a period of time [21].

2.1.4 Stimulus

To observe how the nervous system behaves, *stimulus*, or input can be applied to the system in the form of electric current to see how the system responds with changes in cell membrane potential and neuron firing rate. A *spiking function* is a function that describes the relationship between stimulus and response, or more specifically, how electric current will determine the probability the neuron will spike [21].

2.2 Computational Neuroscience

2.2.1 Overview

Computational neuroscience is the theoretical study of brain function in terms of the information-processing properties of the nervous system components [30]. Prior to computational neuroscience, neurologists studied brain behavior by performing

in vivo and *in vitro* experiments, or experiments using intact, living organisms and experiments using cells grown in a controlled environment such as a test tube, respectively. Computational neuroscience uses brain simulation as a more flexible approach for the study of nervous system behavior.

2.2.2 Models

To better understand how the brain functions, many spiking neuron models have emerged that have trade-offs between computational performance and biological accuracy. Spiking neuron models are mathematical descriptions used to simulate how action potentials are fired and propagated. The following sections describe the Hodgkin-Huxley (HH) [9, 13], leaky integrate-and-fire (LIF) [9], and Izhikevich (IZH) [14] models, which are the most well-known models used in simulations [31].

Hodgkin-Huxley

HH is the most complex and biologically accurate of the three models, designed by observing ion concentrations and currents in a giant squid axon in 1952. The creators discovered a higher potassium current on the outside of the membrane, higher sodium current on the inside, and a leak current. The model describes these currents and how the ions flow through the cell membrane via three types of channels [13].

Leaky-Integrate-and-Fire

LIF is a simplistic neuron model that approximates the biologically accurate HH model, focusing on the leaky integration property of neurons. Current leaks from channels on the neuron membrane at a rate dependent on the membrane capacitance, causing a drop in membrane voltage over a period of time. Once the model's membrane voltage falls to a predefined threshold, the model fires and resets the membrane voltage to the initial value. The combination of leaky integration and reset is what defines the basic integrate-and-fire spiking model [31].

Izhikevich

The intention of the IZH neuron model was to create a model that is as biologically realistic as the HH model, but simple enough to be computationally efficient. By varying the input parameters to the model, IZH can simulate regular spiking, bursting, and fast spiking neuron firing rates; rates that were initially identified by observing the motor cortex of a rat. When the model was first introduced in 2003, it was used in a mammalian cortex simulation of 10,000 neurons and 1,000,000 synapses in real time, with the ratio of excitatory and inhibitory neurons being 4 to 1 [14].

2.2.3 Simulators

Brain simulators use groups of neuron model instances to represent the biological system state at a series of time steps. Some common brain simulation tools include NEURON [5], NEST [25], GENESIS [3], and BRIAN [10], which offer different features and focus capabilities. As stated previously, the more biological details that are simulated, the more compute intensive the simulation becomes. Graphics Processing Units (GPUs) have become popular for brain simulations due to their accelerated compute capability. The neuron doctrine states that the nervous system is made up of discrete cells that are not continuous with any other cells [34], making their computation a perfect candidate for the GPU single instruction multiple data (SIMD) parallel architecture. The following section describes the Neocortical Simulator (NCS), a large-scale brain simulator that runs on a heterogeneous cluster of CPUs and GPUs.

2.3 Neocortical Simulator

2.3.1 Overview

NCS uses the Compute Unified Device Architecture (CUDA) for parallel computation on NVIDIA GPUs and the Message Passing Interface (MPI) for computation on distributed machines, making it capable of simulating 1,000,000 neurons and 100,000,000 synapses in real time [12]. NCS offers the IZH model, and modified LIF and HH mod-

els, and is capable of running simulations with various combinations of these models and synapses. For the mathematical description of the predefined neuron models, please refer to [12, 15]. The following is a description of the required steps and components of an NCS simulation referenced from [12].

2.3.2 Simulation Composition

Components

Every NCS simulation consists of neurons, synapses, stimuli, and reports, where reports are collections of data generated from the simulation. Some optional features include groups, aliases, and channels (calcium-dependent for LIF neurons and voltage-gated for LIF and HH neurons). At each time step, neurons report the spiking state and the membrane voltage. When a presynaptic neuron fires an action potential, a current travels from the neuron through a synapse to a postsynaptic neuron. Stimuli are applied to neuron groups, and can either clamp the membrane voltage or inject a specified amount of current. The available stimulus types are linear current, linear voltage, rectangular current, rectangular voltage, sine current, and sin voltage. Reports are created for individual groups of neurons, so the spike state and membrane voltage data can be extracted from the simulation. The available report types are neuron voltage, neuron fire, input current, and synaptic current. Simulations can utilize groups and aliases to be able to consist of varying neuron models.

Computations

At each time step, the currents for the stimuli and synapses are computed and used to update the state of each neuron. If a neuron enters a spiking state, this data must be used to update any synapses connected to this neuron at later time steps. The neuron, synapse, and stimulus updates are all executed in parallel, but a barrier is used to ensure that execution does not begin until all the input data for that group has been received. Data is transferred between computation groups via a publisher/subscriber message passing system.

Reports

A reporter object is created for each simulation component that is being reported on. Using the publisher/subscriber system, the reporter subscribes to the data source and extracts all the produced data at each time step. For more details on how the messages are distributed to the appropriate machines in the cluster, please refer to [12].

2.3.3 PyNCS

In the latest version of NCS, a Python interface was added so users can create and run simulations with scripts. An example PyNCS Python script is shown in Figure 2.2. The Python programming language was chosen because it is easy to use and was deployed as the scripting language for other simulators like NEURON, NEST, and BRIAN. NCS is written in C++ and CUDA C, so the PyNCS library was created to interface with NCS from Python scripts. The PyNCS API includes functions for adding neurons, synapses, stimulus, groups, aliases, reports, and initializing and running a simulation. For more details on the implementation of PyNCS, please refer to [33].

2.3.4 NCS Web Interface

As an alternative to using Python scripts, NCS also has a web interface that can be used to build models and run simulations. The web interface was created to make NCS available to a broader range of users because it does not require any programming to use [1, 2]. The web interface provides user profiles, so models can be saved and shared. All users have access to global models, users in a lab group can share models, and users can create personal models that are only available to the user. The web interface consists of the following components: model builder, simulation builder, reports, and virtual robot. The following is a description of each of the components in more detail.

```

#!/usr/bin/python

import os, sys
import ncs

def run(argv):

    sim = ncs.Simulation()

    # Izhikevich neuron
    regular_spiking_params = sim.addNeuron("regular_spiking","izhikevich",
        {
            "a": 0.02,
            "b": 0.2,
            "c": -65.0,
            "d": 8.0,
            "u": -12.0,
            "v": -60.0,
            "threshold": 30
        })
    group=sim.addNeuronGroup("group_1",1,regular_spiking_params,None)

    # Initialize the simulation
    if not sim.init(argv):
        print "failed to initialize simulation."
        return

    # Add stimulus
    stim_param = {
        "amplitude":10
    }

    sim.addStimulus("rectangular_current", stim_param, group, 1, 0.1, 1.0)

    # Report neuron voltage
    report=sim.addReport("group", "neuron", "neuron_voltage", 1, 0.0, 1.0)
    report.toAsciiFile("./regular_spiking_izh.txt")

    # Run the simulation for 1 second
    sim.run(duration=1.0)

    return

if __name__ == "__main__":
    run(sys.argv)
}

```

Figure 2.2: An example of a simple PyNCS Python script.

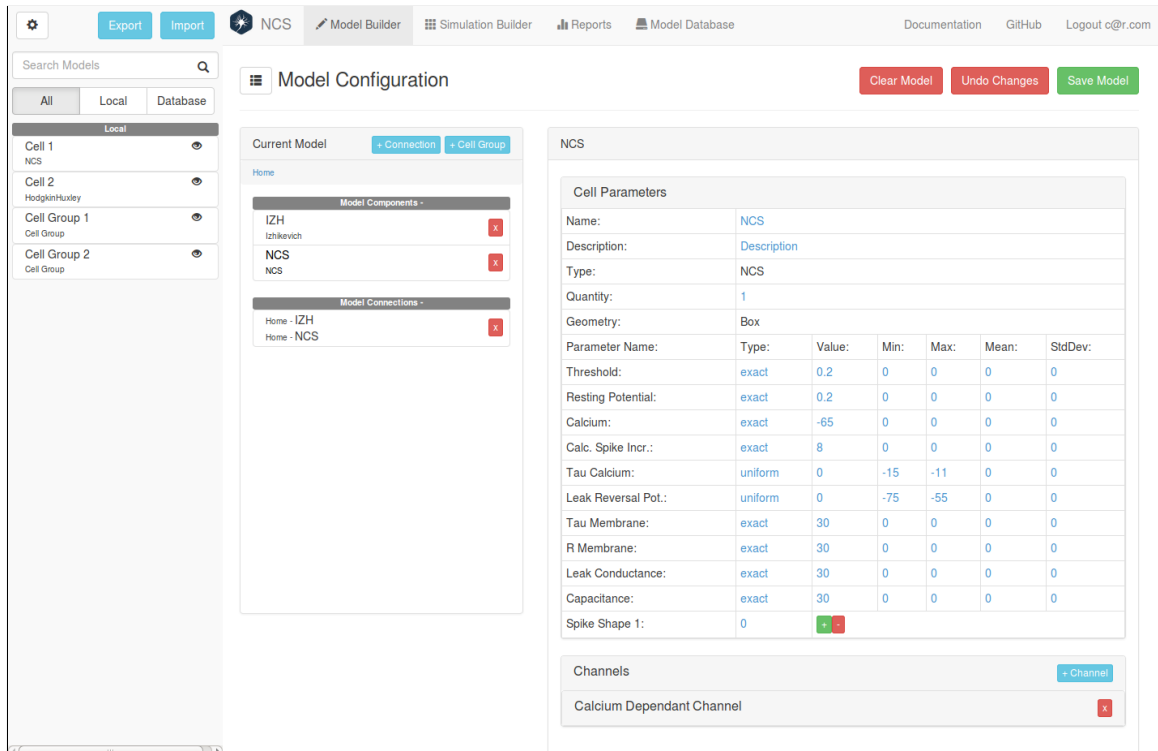


Figure 2.3: A screenshot of the NCS web interface brain model builder tab.

Model Builder

The model builder tab shown in Figure 2.3 is where the user can select a preexisting model or create a new model for a simulation. The user can create cell groups with the available neuron models, add synapses, and set parameters for each [2].

Simulation Builder

The simulation builder tab shown in Figure 2.4 is where the user can set simulation parameters, add stimulus and reports, and run the simulation [2].

Reports

The reports tab is where the user can view data reported by a simulation, streamed in real time. To promote readability and further analysis, the report data can be viewed as raw data or in graphs as shown in Figure 2.5. Although the data is streaming live, the user can also log out and log in at a later time to view the data generated by a

NCS Model Builder Simulation Builder Reports Model Database Documentation GitHub Logout c@r.com

Simulation Configuration Launch Simulation

Simulation Parameters - Minimize

Simulation Name Regular Spiking	FSV 1	Seed 0
Duration (Seconds) 5	Interactive No	Include Distance No

Input New

Input1 x

Input1 Specification

Stimulus Type
Rectangular Current

Parameter Name:	Type:	Value:	Min:	Max:	Mean:	StdDev:
Amplitude	exact	0	0	0	0	0
Start Amplitude	exact	0	0	0	0	0
End Amplitude	exact	0	0	0	0	0
Amplitude Scale	exact	0	0	0	0	0
Amplitude Shift	exact	0	0	0	0	0
Delay (Seconds)	exact	0	0	0	0	0
Time Scale	exact	0	0	0	0	0
Current (Amps)	exact	0	0	0	0	0
Phase	exact	0	0	0	0	0
Width	exact	0	0	0	0	0
Frequency	exact	0	0	0	0	0

Probability **Start Time (Seconds)**

Figure 2.4: A screenshot of the NCS web interface simulation builder tab.

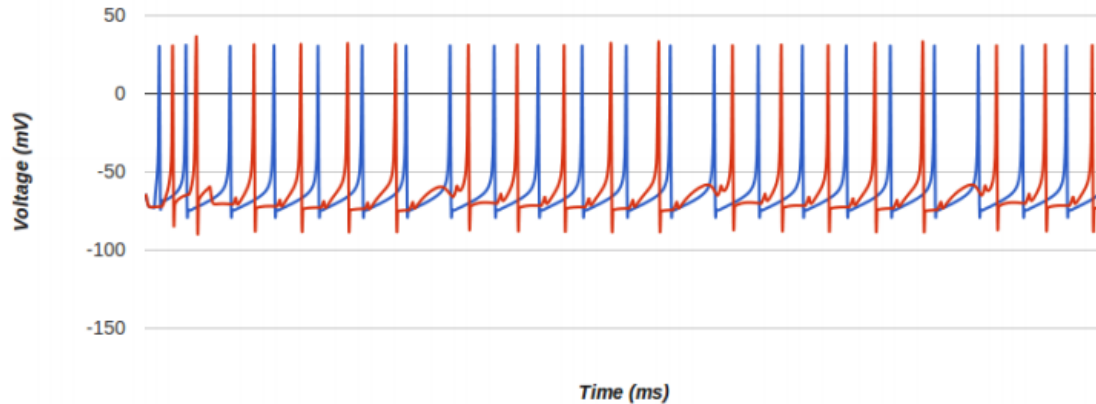


Figure 2.5: An example graph of NCS simulation report data, from [33].

simulation.

Virtual Robot

The virtual robot tab is designed to allow users to observe how simulated brain models behave in a realistic environment. The user can run a simulation with a created brain model; audio and visual input perceived by the robot in the environment serving as stimulus inputs to the simulation and streaming report data used to determine the robot's behavior [16]. An example of the robot in a virtual environment is shown in Figure 2.6. This functionality is under development because NCS currently does not support image and audio file inputs, or adding stimulus inputs after the simulation is running.

2.4 Centralized Services

2.4.1 Overview

Grouping a set of functionality into a centralized service promotes easier feature set change and configuration management. Requiring all communication among NCS, the web interface, and the database to go through the daemon allows for the components to remain independent while being able to exchange data with one another in an efficient manner. The addition of a centralized service provider encouraged the NCS

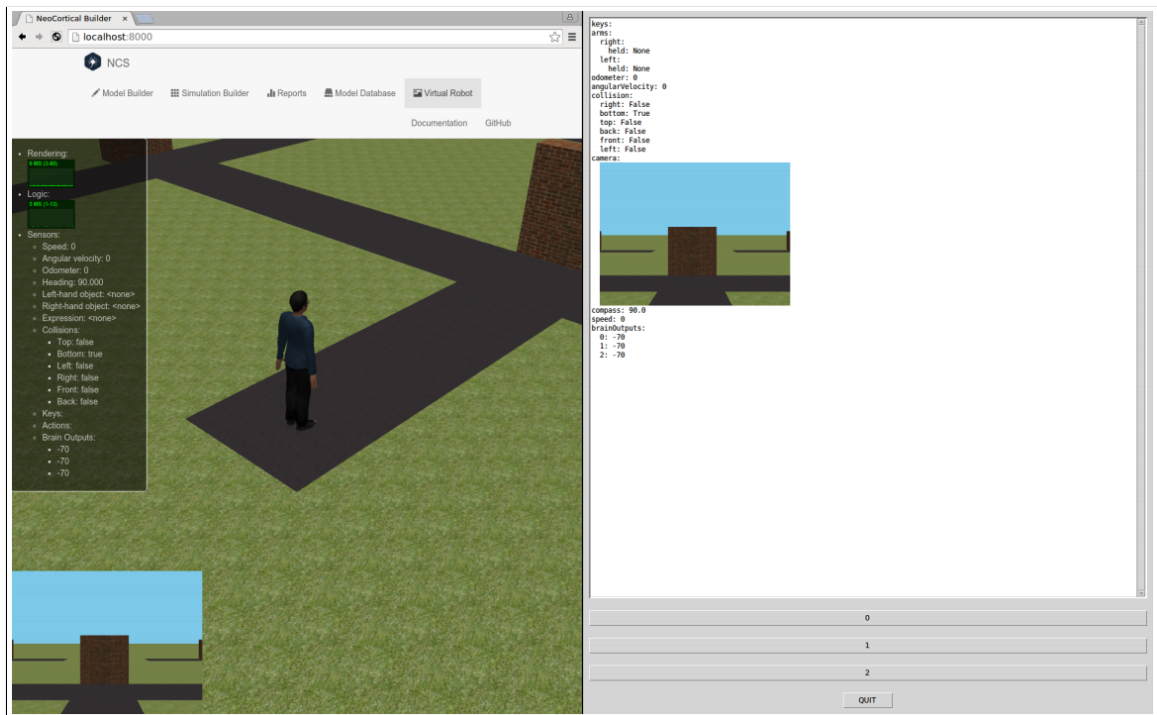


Figure 2.6: A screenshot of the NCS web interface virtual robot tab while a simulation is running, from [16]

system to use the client-server architecture, and the lack of user interaction with the centralized service allowed it to run as a daemon process.

2.4.2 Client-server Architecture

In a client-server system, every entity acts as either a server or a client. Typically, a server is a centralized entity that resides at a known location to offer a set of services to clients. Namely, a server is responsible for providing resources or services, and clients request resources or services from the server [20]. The specific client-server architecture of the application is discussed in Chapter 3.

2.4.3 Daemon Processes

The NCS daemon application does not interact with any users; it only communicates with NCS, the web interface, and the database. Thus, it does not require a user interface or even to be noticeably running on the system. Therefore, the application executes as a daemon process, meaning it is a program that runs in the background, not allowing users to interact with it directly [8]. The only situations requiring a user to interact with the daemon are to start it and terminate it.

2.5 Libraries and Frameworks

2.5.1 Overview

The following is a description of the libraries and frameworks used for the implementation of the NCS daemon.

2.5.2 Python

The daemon is implemented in the Python programming language. Python not only balances ease of use with flexibility, but it was chosen specifically because of the extensive amount libraries and frameworks it has available. Python's standard library includes JSON and socket interfaces [26], which simplified communication with the NCS web interface. Also, the Python package index has the Twisted framework, used

for asynchronous network programming [26]. Twisted is described in more detail below.

2.5.3 Bcrypt

The daemon is responsible for storing the web interface's user profiles, so the bcrypt encryption library is used to encrypt the user passwords before they are stored. Bcrypt uses a modified version of the Blowfish stream cipher, a symmetric key encryption algorithm, or an encryption algorithm that uses a single key for the encryption and decryption process [22]. Bcrypt generates long SALT values that it passes into the encryption algorithm to hash whatever it is encrypting. Each item being encrypted is given a new, random SALT value, which is also used for the decryption process. Bcrypt was chosen because it provides adequate security for the purpose of the application, and it can be used with Python if the bcrypt package is installed [27].

2.5.4 Protocol Buffers

Overview

Protobuf is a library created by Google that used for serializing data in an efficient manner. Protobuf is currently is available for C++, Java and Python [11]. It is generally not desirable to transmit raw floating point values across the network because the values will not necessarily be represented the same on machines with different architectures. Therefore, some method of encoding was required to transmit the simulation output voltages from NCS to the daemon. Protobuf was deemed an acceptable candidate because NCS is written in C++ and the daemon is written in Python.

Usage

Defining the Data Schema Protobuf requires a data schema be defined in a *.proto* file. Similar to other data structures, the schema can be an encapsulation of elements of varying data types. An example of a Protobuf data structure definition is shown in Figure 2.7. When the schema is compiled with the Protobuf compiler, the class

```

message Student {
  required string name = 1;
  required int32 id = 2;
  required float gpa = 3;
}

```

Figure 2.7: An example of a Protobuf data structure definition.

used to serialize and deserialize the data structure is generated. The schema must be compiled individually for each language used.

Serializing and Deserializing Once the Protobuf object is created, instances of it can be used to serialize and deserialize data. The data can be serialized into a string type in one language and deserialized in a different language, as long as the Protobuf object has been created for both languages. The process used for our specific purpose is described in more detail in Chapter 4.

2.5.5 Twisted

Overview

The daemon uses a client-server architecture, and like most applications of this architecture, requires scalability and network communication. Twisted is an event-driven networking framework available for Python. Twisted is a popular framework because it provides an engine for managing client-server architectures, and it provides interfaces for transport and application layer protocols, such as TCP/UDP, HTTP, IMAP, event logging, and many others [35]. Additionally, Twisted has protocols for authentication, logging, and running programs as daemon processes, all of which are required by the daemon.

Event-driven Programming Model

As stated previously, Twisted uses event-driven asynchronous execution to allow multiple tasks to run at what appears to be simultaneously. Asynchronous execution is non-blocking; if a task must wait for some reason, the program will allow another

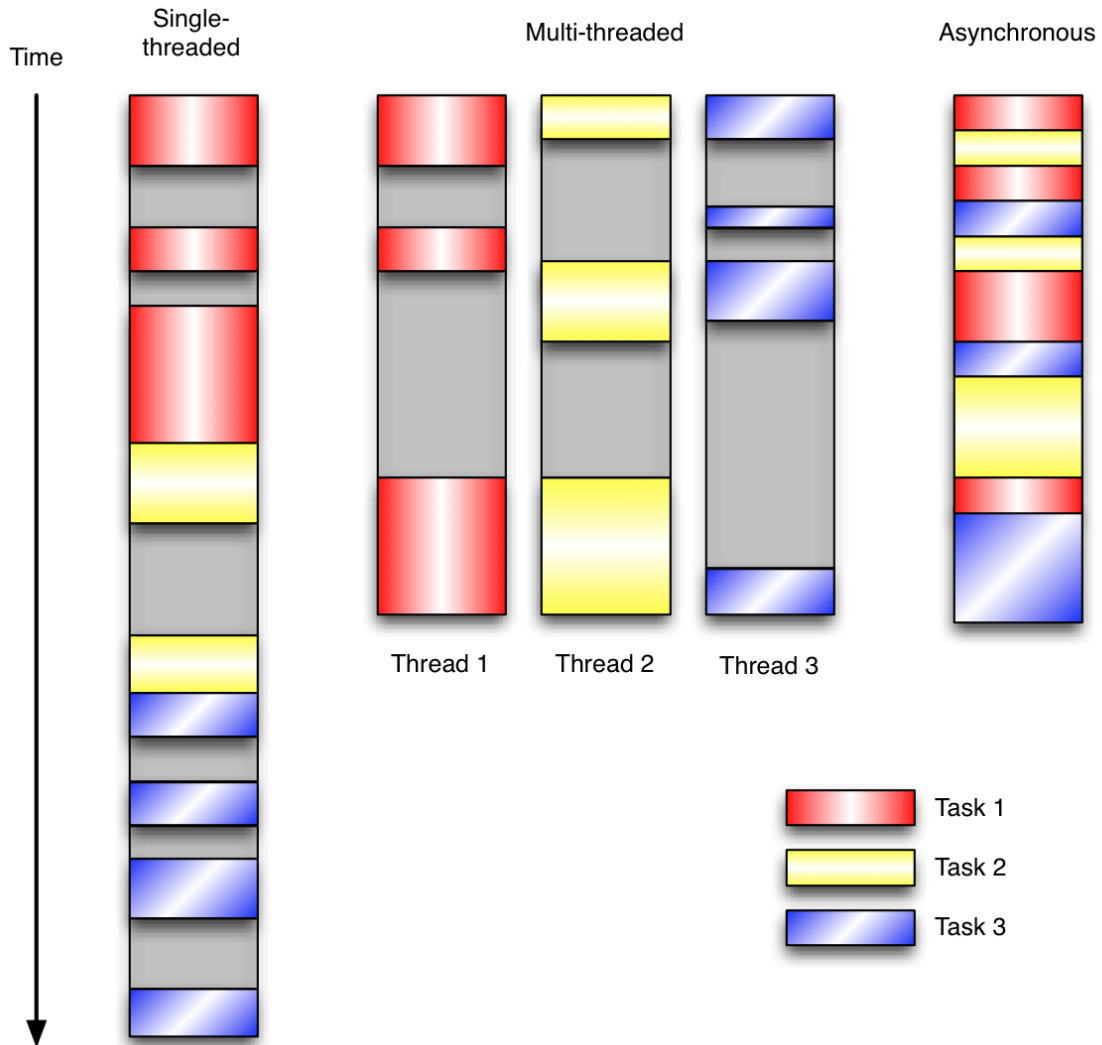


Figure 2.8: A diagram of multi-task program models, from [17].

task to execute. A diagram depicting how asynchronous and synchronous execution models vary is shown in Figure 2.8. Event-driven programming is commonly based on the reactor design pattern [23]. Programs that follow this pattern have a reactor loop that waits for events to occur. The reactor handles events by executing the appropriate callback, a function that handles what should be done when a particular event occurs.

The primary incentive for using the asynchronous programming model is control. Synchronous multi-threaded systems are difficult to manage because threads often

times require synchronization for accessing shared resources and eliminating race conditions, starvation, and deadlock. Moreover, the scheduling of the threads is controlled by the operating system as opposed to the programmer, which further attributes to the difficulty of designing a complex system.

Although asynchronous programs do not suffer from these complications, they still require a design with some degree of complexity and are only advantageous in certain situations. The asynchronous model should be considered if the system has a large number of tasks, the tasks perform I/O or other blocking operations, and the tasks are mostly independent (inter-task communication is minimal) [23]. Thus, the asynchronous programming model is a sound choice for a networked client-server system where the server creates a separate task for each client, and the clients make requests to and receive data from the server.

Application Infrastructure

Overview Twisted applications are deployed using an infrastructure consisting of services, applications, and Twisted application configuration (TAC) files. This configurable deployment method allows for easy addition of services and application daemonization. The following is a list of the descriptions of the components of the architecture referenced from [6].

Services A Twisted service object is used for anything that can be started and stopped. Twisted has service implementations for TCP, FTP, HTTP, SSH, DNS, and many others. All services in the daemon are instances of the TCP service.

Applications A Twisted application is the object that holds all the services. The application uses a service manager through which services must register. When the application is deployed, all of the registered services are made available.

TAC Files TAC files are where the services are registered with an application, and are used to configure a Twisted program.

twistd *twistd* is the command-line utility used to deploy a Twisted application by running the TAC file, and starting and stopping the reactor. By default, *twistd* daemonizes the application, but a flag can be used to run the process in the foreground. When the application is ran as a daemon process, it can be terminated with the *twistd* process ID.

Authentication

Twisted Cred The NCS web interface has user profiles, requiring it to use a form of authentication. Twisted provides an authentication system called *Cred*, which is generalizable in the sense that the specific authentication methods are overridden. *Cred* has a *ICredentialsChecker* object that provides the interface for authenticating credentials with some form of database, and granting the user with a request token upon authentication. The specific query required to verify a user's credentials is passed into the constructor of the *ICredentialsChecker* object to keep the verification method generalized. *Cred* also provides a *Realm* object that gives the user access to a list of *avatars*, or actions and data available to users. Methods that are overridden include handling a new client connection, receiving requests, logging in and out, and handling the loss of a connection [6].

Logging

Twisted provides an event logging system that is comparable to the one included in the Python standard library [6]. An example log is shown in Figure 2.9. When running a Twisted application with *twistd*, logging is written to a file if the process is daemonized and to the console if the process is ran in the foreground.

2.5.6 RabbitMQ

Overview

One of the functional requirements of the NCS web interface is users should have the capability to start a simulation, log out, and log in at a later time to view the simu-

```

2012-11-15 20:26:37-0500 [-] Log opened.
2012-11-15 20:26:37-0500 [-] EchoFactory starting on 8000
2012-11-15 20:26:37-0500 [-] Starting factory <__main__.EchoFactory ...
2012-11-15 20:26:40-0500 [Echo,0,127.0.0.1] Hello, world!
2012-11-15 20:26:43-0500 [-] Received SIGINT, shutting down.
2012-11-15 20:26:43-0500 [__main__.EchoFactory] (TCP Port 8000 Closed)
2012-11-15 20:26:43-0500 [__main__.EchoFactory] Stopping factory <__...
2012-11-15 20:26:43-0500 [-] Main loop terminated.

```

Figure 2.9: An example of a Twisted log of an echo server starting up, echoing one message, and terminating, from [6].

lation output data. This functionality requires the daemon to support asynchronous messaging. RabbitMQ is an application-layer protocol message broker, or message intermediary, that implements the Advanced Message Queuing Protocol (AMQP). AMQP defines the message queue standard where a producers publish messages to *exchanges*, which are then copied into the specified queues. Message brokers either deliver the queued messages to the consumers subscribed to the queues, or the consumers can poll the queues for new messages. Using a message intermediary allows for asynchronous sending and receiving of messages because sent messages are queued until the receiver is ready to receive them. Other useful features of RabbitMQ are the ability to function on a distributed system, and the message queues are limited in size by only the amount of disk space unless otherwise specified [24].

Routing Message Pattern

RabbitMQ supports a variety of message patterns: work queues, publish/subscribe, routing (receiving messages selectively), topics (receiving messages based on a pattern), and Report Procedure Call (RPC) [24]. The daemon uses the routing message pattern, allowing a producer to have a direct exchange with selected consumers. More specifically, a message goes to the queues whose binding key matches the routing key of the message. A diagram of a producer's messages routed to consumers based on the routing key is illustrated in Figure 2.10. It is possible to bind multiple queues with the same binding key to essentially allow multiple consumers to receive the same

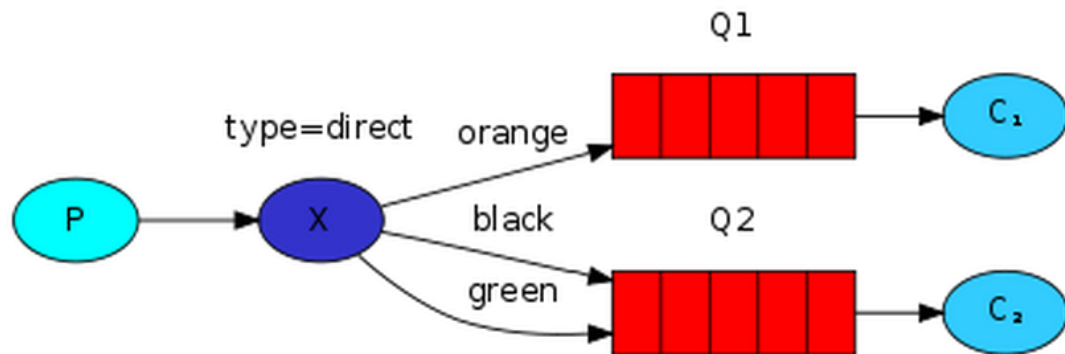


Figure 2.10: A diagram of a direct exchange with Queue 1 bound with the binding key *orange* and Queue 2 bound with the binding keys *black* and *green*, from [24]

messages, but this feature is not utilized in our implementation. More details on the specific use of the direct exchange are discussed in Chapter 4.

TxAMQP

When using other protocols with Twisted, it is important that the libraries used to access these protocols are designed for asynchronous program execution in the sense that the function calls must be non-blocking so to not hold up the Twisted program. TxAMQP is a Python library used for communicating with AMQP message brokers and clients, include those using RabbitMQ, from Twisted programs [28].

2.5.7 JSON

JavaScript Object Notation (JSON) objects are human-readable collections of data used to store and transmit data. JSON objects are similar to map or dictionary data structures in that they are collections of attribute-value pairs, but they are serialized so they can be transmitted across the network. JSON is supported by many programming languages and frameworks, including Python and web applications [4]. The NCS web application and the daemon use JSON to store user and simulation data so it can be transmitted between the two applications.

2.5.8 MongoDB

MongoDB is a NoSQL database, meaning it does not store data in tables like traditional relational databases, and it does not use SQL queries to access and manipulate the data. Instead of storing data in *columns* and *tables* like traditional databases, MongoDB stores data in *documents* and *collections*, respectively. The documents are stored in Binary JSON (BSON) format, so no conversion is necessary by the programmer to store JSON objects in the database [18]. The daemon uses MongoDB to store user profiles and created brain models for the NCS web interface.

TxMongo

PyMongo is the standard API for using MongoDB with Python. However, PyMongo is not designed for asynchronous program execution because the function calls are blocking. TxMongo is alternative library that provides asynchronous communication with a MongoDB database so it can be used with Twisted [7, 29].

Chapter 3

Design

3.1 Overview

The NCS daemon is the centralized service that allows the web interface to access NCS. More specifically, it is responsible for authenticating users for the web interface, storing brain models, running simulations, and streaming the report data. The design of the daemon was restricted in some aspects because NCS and the web interface were already existing components in the system. This chapter describes the daemon functional and non-functional requirements, architecture, and how it interacts with the other system components. The web interface use cases are also discussed to provide a clearer understanding of the daemon functional requirements.

3.2 Service Requirements

3.2.1 Functional Requirements

The functional requirements stemmed from the behavioral requirements of the daemon when interacting with NCS and the web interface. The list of functional requirements are listed in Table 3.1.

3.2.2 Non-functional Requirements

The non-functional requirements stemmed from the requirements of communicating with NCS and the web interface, and implementation specifics for meeting the behavioral requirements. The non-functional requirements are listed in Table 3.2.

Table 3.1: The NCS daemon functional requirements.

Number	Description
FR01	The daemon shall add new user profiles to the web interface user group.
FR02	The daemon shall authenticate users given the user credentials.
FR03	The daemon shall retrieve the personal, lab, and global models for a user.
FR04	The daemon shall save personal, lab, and global models.
FR05	The daemon shall allow a user to revert a model to a previous save.
FR06	The daemon shall use a brain model and simulation parameters to run a simulation on NCS.
FR07	The daemon shall convert NCS simulation Python scripts to brain model and simulation parameter JSON objects.
FR08	The daemon shall convert brain model and simulation parameter JSON objects to NCS simulation Python scripts.
FR09	The daemon shall stream NCS simulation report data in real time.
FR10	The daemon shall queue any streamed report data if the user is not logged into the web interface.
FR11	The daemon shall log any errors that occur.

Table 3.2: The NCS daemon non-functional requirements.

Number	Description
NR01	The daemon shall be written in Python.
NR02	The daemon shall use Twisted to manage a client-server architecture.
NR03	The daemon must use Protocol Buffers to serialize the streamed report data.
NR04	The daemon shall use RabbitMQ to provide asynchronous report data streaming.
NR05	The daemon shall accept JSON objects from the web interface for requests, and model and simulation parameters.
NR06	The daemon shall use MongoDB to store user profiles and brain models.
NR07	The daemon shall use PyNCS to run simulations.
NR08	The daemon shall use PyNCS to perform conversions between JSON objects and Python scripts.
NR09	The daemon shall only use APIs that have non-blocking function calls.

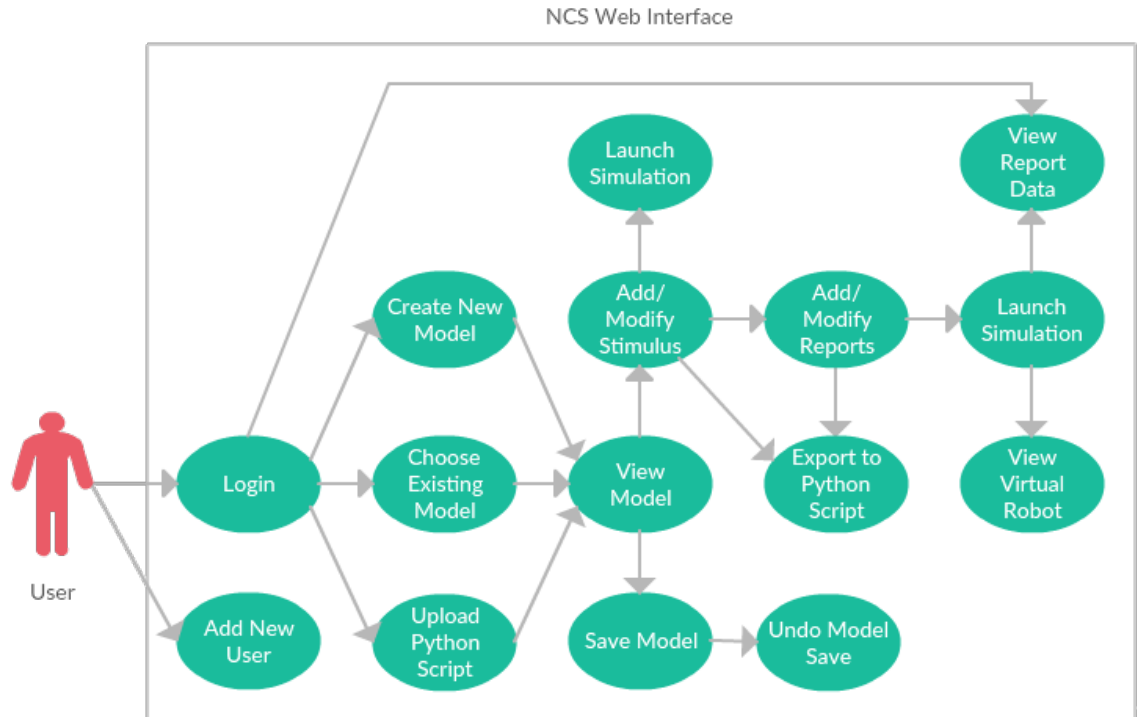


Figure 3.1: A use case diagram of the NCS web interface.

3.3 Use Case Modeling

3.3.1 Overview

For better understanding of the responsibilities of the daemon, this section describes the web interface use cases. Aside from showing how the user interacts with the web interface, these use cases represent why the daemon must be able to handle requests from the web interface, and how the daemon interacts with the other system components to service those requests is described in later sections. A diagram of the use cases is shown in Figure 3.1 and the use cases are discussed in the subsequent section.

3.3.2 Detailed Use Cases

Add New User

The home page of the web interface gives the user the options to either log in with their account or join as a new user. If the user selects to join, he or she will be required to provide an email address that will serve as a user name, a password, first name, last name, institution name, and lab ID. The user will be added only if a valid email address is given and the user does not already exist.

Login

The user is required to log into the web interface to access any of the website functionality. When logging in, the user is prompted for a user name and password. If valid credentials are given, the user will be taken to a new page on the web interface where he or she will have access to the main functionality.

Create New Model

In the model builder tab, the user can use the cell models supported by NCS to create a new brain model. Once cell groups are created, the user can add synapses to the model and set various parameters for the cells and synapses. Depending on the cell model type, the user may also have the option of adding different channel types and setting various parameters for each.

Choose Existing Model

The user can select an existing brain model from the personal, lab, or global groups. When a model is selected, its components and their parameters will populate the model builder interface where it can be manipulated if desired.

Add/Modify Stimulus

In the simulation builder tab, the user can add stimulus to specified groups of neurons in the brain model. The user can select from a variety of stimulus types and can

set parameters depending on the type selected. If the simulation parameters are populated from an existing model or an uploaded Python script, the user can modify or confirm the current stimulus parameters.

Add/Modify Reports

In the simulation builder tab, the user can add reports to specified groups of neurons in the brain model. The user can select from a variety of report types and can set parameters depending on the type selected. If the simulation parameters are populated from an existing model or an uploaded Python script, the user can modify or confirm the current report parameters.

Export to Python Script

Once a full set of simulation parameters has been created, the user can export the simulation to a download-able PyNCS Python script file.

Launch Simulation

Once a full set of simulation parameters has been created, the user can run the simulation on NCS.

View Report Data

If the user added a report to at least one neuron group in the simulation, the user can view the live streaming simulation data in the reports tab. The user can also view the data at a later time, including after logging out and logging back in, and after the simulation has finished running.

View Virtual Robot

Although this feature is under development, it is intended to allow users to see how visual and audio data perceived by the robot stimulate the cell groups in the simulation, and how output data from reporting cell groups affect the robot's behavior.

3.4 Architecture

The system containing NCS, the web interface, and the daemon follow the client-server architecture pattern to provide the use cases listed in the previous section. The web interface always acts as a client that makes requests to and receives data from the daemon. The daemon acts as a proxy server, processing requests from the web interface that sometimes require it to make requests to NCS and manipulate the database. A diagram of the system components is shown in Figure 3.2. The daemon contains three Twisted services: the Add New User Service, the Authentication Service, and the Data Proxy Service. The arrows in the diagram represent one component making a request to another, excluding any data that sent as a response of the request. The requests made by the web interface correspond to the use cases listed in the previous section, and the other requests are results of the web interface requests.

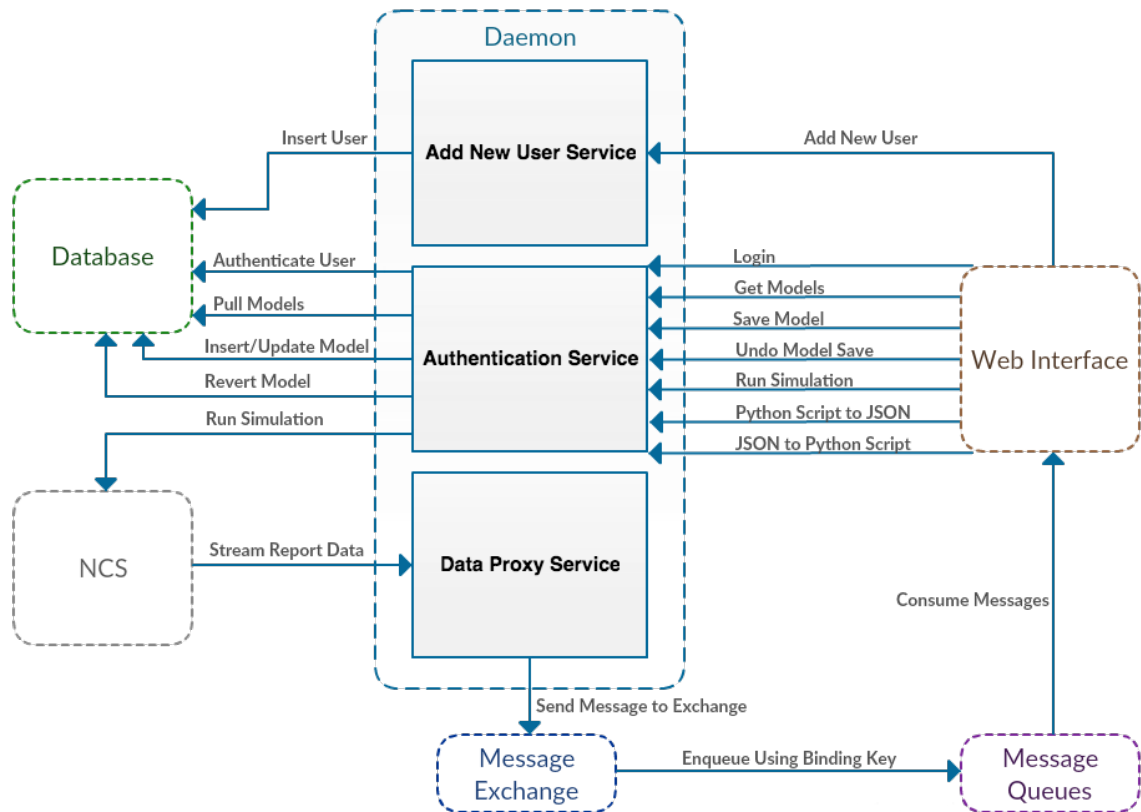


Figure 3.2: A system architecture diagram of the daemon and how it interacts with the other components.

Chapter 4

Implementation

4.1 Overview

This chapter describes each of the components that were implemented to provide the behavioral requirements of the NCS daemon. It also specifies where and how the libraries and frameworks described in Chapter 2 were used in the development of the system.

4.2 Database Management

4.2.1 Overview

The daemon uses MongoDB to store user profiles and brain models for the web interface. TxMongo is used as the API to access the database from the Twisted application. This section describes the structure of the database and the TxMongo functions used for accessing the database.

4.2.2 Structure

Overview

In the database there are defined collections for users, global models, and lab models. Storing the lab and global models with each user would require multiple copies of the same models, and it would be more difficult to ensure that all users have consistent versions of the models. For space efficiency and model consistency, lab and global

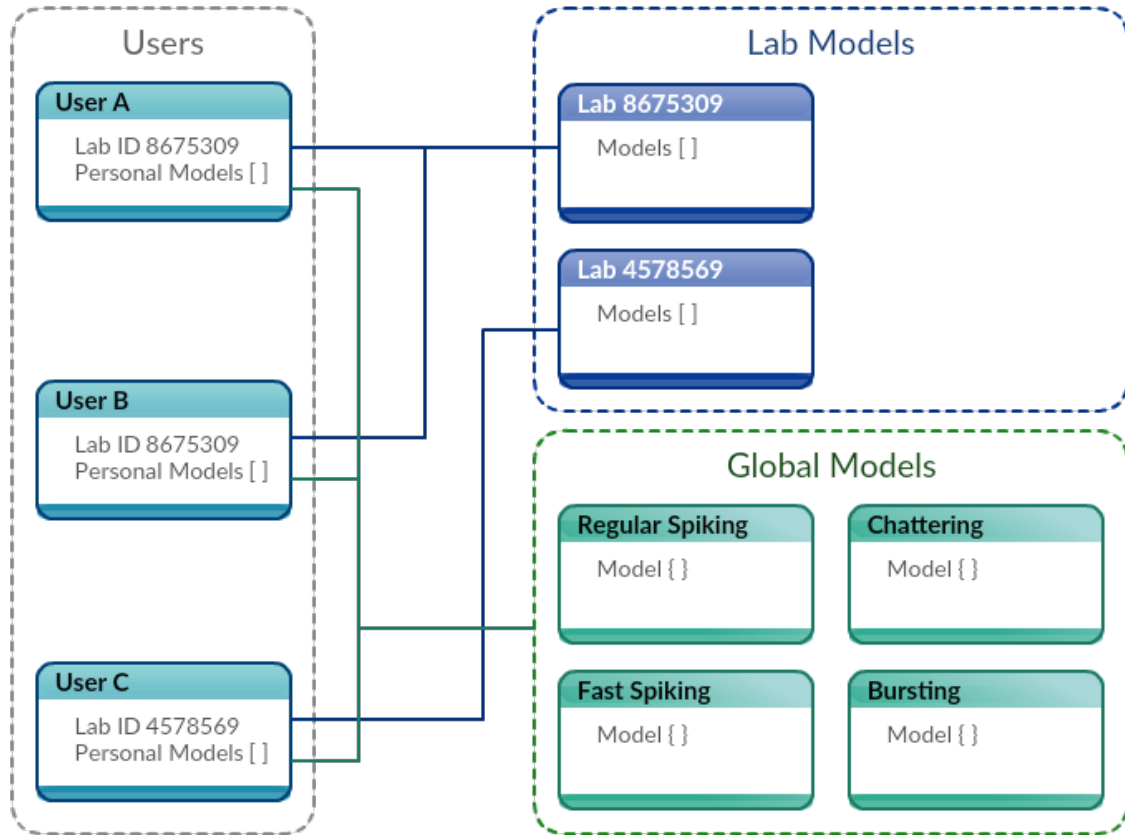


Figure 4.1: An example of the database structure where { } items represent BSON objects and [] items represent lists of BSON objects.

models are stored in their own collections and personal models are stored in the appropriate user document. Every user is given the set of global models, and the set of appropriate lab models is retrieved with a unique lab ID number. A diagram of the database structure is shown in Figure 4.1. The document structures for users and models are discussed in more detail in the following sections.

Users

The *user* collection contains documents for all users that have access to the web interface. The fields in the user document include the user's login credentials, some personal information, and a list of any saved personal models stored in JSON format. Passwords are encrypted before the user document is created. An example of a user document is shown in Figure 4.2. For space reasons, the model list in the example is

```

{
  "username": "jdoe@nevada.unr.edu",
  "password": "b8f2785ad8fd4dc015cff05225f455557993abef",
  "first_name": "Jane",
  "last_name": "Doe",
  "institution": "UNR",
  "lab_id": 8675309,
  "salt": "ad65d5054042fds44ba3fdc97cee80c6",
  "models": [ ]
}

```

Figure 4.2: An example of a document in the *users* collection.

```

{
  "model_rev_0":
  {
    "date_created": "06/01/2015",
    "last_modified": "06/01/2015",
    "model": { }
  },
  "model_rev_1":
  {
    "date_created": "06/08/2015",
    "last_modified": "06/08/2015",
    "model": { }
  }
}

```

Figure 4.3: An example of a model document.

empty.

Models

The same JSON structure is used to store models regardless of if they are stored in the global collection, a lab collection, or in the appropriate user document. For every model, two versions of the model are stored: a previous version and a current version. The previous version is stored so the user has the option to undo the last model save. An example of a model document is shown in Figure 4.3. The “model” field holds the model JSON object used by the web interface. For space reasons, the actual model JSON objects are not included in Figure 4.3, but an example of a model JSON can be viewed in Figure 4.4.

```

{
  "author": "Jane Doe",
  "cellAliases": [],
  "cellGroups": {
    "cellGroups": [
      {
        "name": "IZH Cells",
        "num": 150,
        "type": "Izhikevich",
        "parameters": {
          "a": {
            "type": "exact",
            "value": 0.2
          },
          "b": {
            "type": "exact",
            "value": 0.2
          },
          "c": {
            "type": "exact",
            "value": -65
          },
          "d": {
            "type": "exact",
            "value": 8
          },
          "threshold": {
            "type": "exact",
            "value": 30
          },
          "u": {
            "maxValue": -11,
            "minValue": -15,
            "type": "uniform"
          },
          "v": {
            "maxValue": -55,
            "minValue": -75,
            "type": "uniform"
          }
        }
      }
    ],
  },
  "description": "Regular Spiking",
  "name": "Test Model",
  "synapses": []
}

```

Figure 4.4: An example of a simple model JSON.

4.2.3 TxMongo Functions

Insert

The *insert* function is used to add documents to the database. When calling this function, the collection must be specified in which the document is desired to be inserted. If the collection does not exist, it will be created and the document will be inserted into it. This function is used to insert users and models into the database.

Find

The *find* function is used to query the database for document(s). If the function is called without any parameters, all the documents in the database are returned. If a field is specified as the function's parameter, all the documents in the database matching the field description are returned. This function is used to authenticate users, determine if a user already exists when adding new users, and pull all models available for a user.

Update

The *update* function is used to modify a current document in the database. This function takes the original document and the updated document as parameters. If the specified document is not found, the new document is not inserted into the database. This function is used to save changes to a model and revert a model to a previous version.

Drop

The *drop* function is used to remove a collection from the database. It is only used by this application to remove all collections to essentially drop the database if needed.

4.3 Twisted Services

4.3.1 Overview

As shown in Figure 3.2, the daemon is comprised of the Add New User Service, the Authentication Service, and the Data Proxy Service. The three services are all instances of the Twisted TCP service; the services listen for clients on static IP addresses and port numbers. The Add New User Service accepts new connections from the web interface for adding new users to the list of users that can access the web interface. The Authentication Service accepts new connections from the web interface for logging in, and only once the user has been authenticated, requests to get models, save models, undo model saves, run simulations, and perform Python script and JSON object conversions. The Data Proxy Service accepts new connections from NCS for receiving simulation report data.

4.3.2 Add New User Service

The Add New User Service is responsible for processing the request to add a new user to the group of users that can access the web interface. A diagram of the Add New User Service interacting with the other system components to process this request is shown in Figure 4.5. The request must be a JSON object containing an email address as the user name, a password, first name, last name, institution name, lab ID. When the request is received, the Add New User Service queries the users collection in the database with the provided user name as the query field. If any documents are returned from the query, a failed message is sent back because the user already exists. If no documents are returned from the query, bcrypt is used to generate a random SALT value and encrypt the password using the SALT value. The password in the JSON is replaced with the encrypted password and the SALT value is added to the JSON object. An empty field for personal models is also added to the JSON object. The JSON object is then inserted into the users collection in the database and a success message is sent back. The user is now able to login to the web interface.

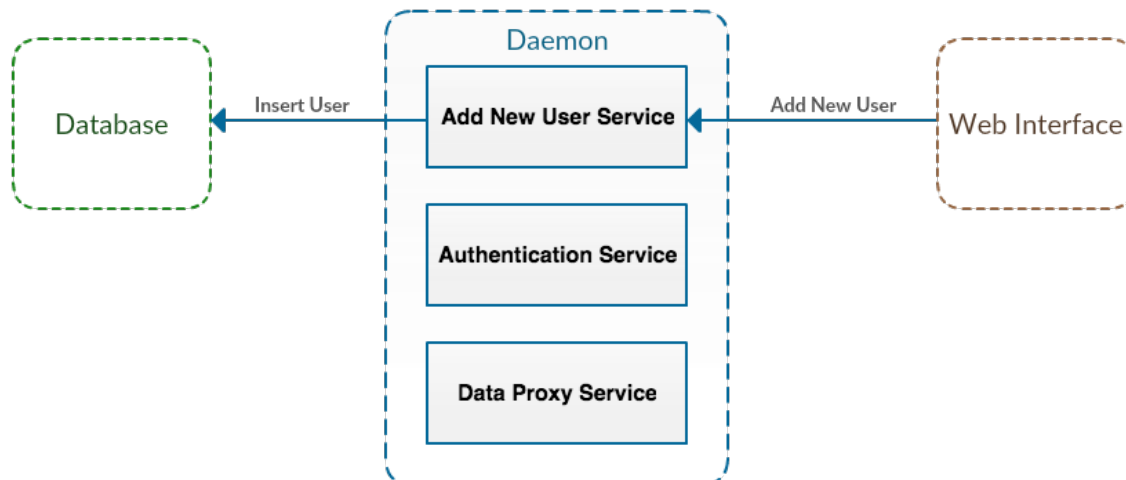


Figure 4.5: A diagram of the Add New User Service interacting with the other system components.

4.3.3 Authentication Service

Overview

The Authentication Service is responsible for processing requests to login, get models, save models, undo model saves, runs simulations, convert Python scripts to simulation JSON objects, and export simulation JSON objects as Python scripts. A diagram of the Authentication Service interacting with the other system components is shown in Figure 4.6. The daemon uses the Twisted Cred interface to authenticate users for the web interface. When the Authentication Service receives a request to login, it uses the *iCredentialsChecker* object to authenticate the user credentials. To check the credentials, the *iCredentialsChecker* object queries the users collection in the database with the provided user name as the query field. If a document is returned from the query, the password stored in the user document is then compared to the provided password. Since the passwords are not stored in plain text, bcrypt is used to encrypt the provided password using the SALT value stored in the user document before the passwords are compared. If either the user is not found in the database or the passwords do not match, a failure message is sent back and the TCP connection is closed. If the user is authenticated, a success message is sent back and the connection is

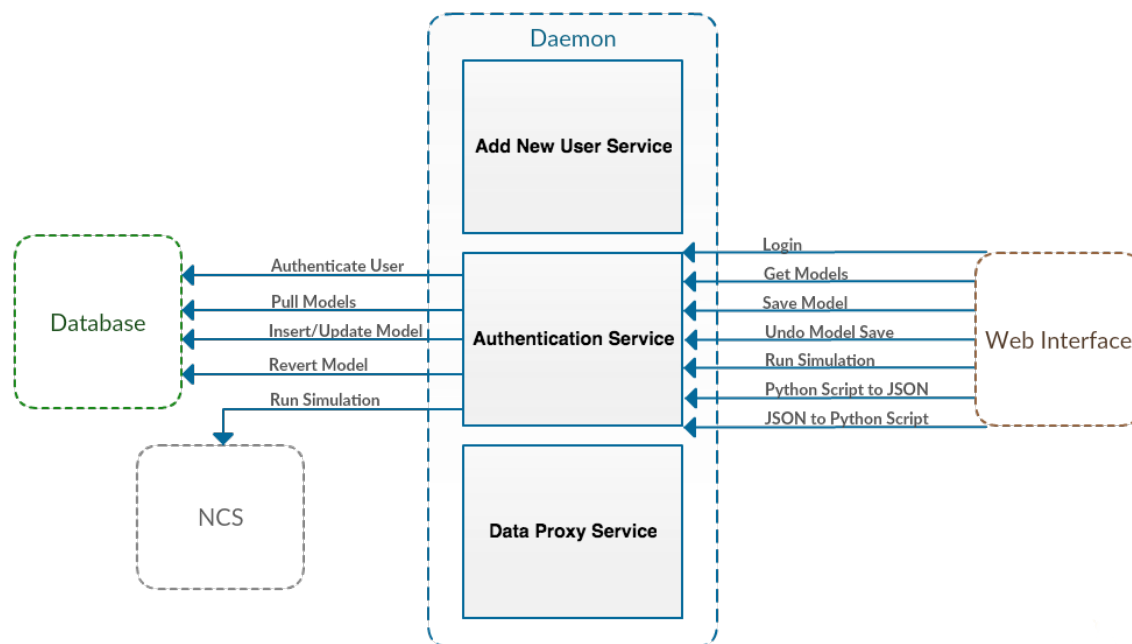


Figure 4.6: A diagram of the Authentication Service interacting with the other system components.

granted an avatar. An ID created from a combination of the user name and lab ID is assigned to the avatar so the user can be identified when making requests. Once the connection is granted an avatar, all other requests handled by the Authentication Service can now be made. The requests received by the Authentication Service must be in a JSON object with a “request” field. For any request received by the Authentication Service that is not a login request, the Authentication Service checks if there is a valid avatar for the connection before processing the request. The following describes these requests in more detail.

Get Models

The request retrieve models pulls the appropriate personal, lab, and global models from the database. This request should be made when the user logs in and anytime the models need to be refreshed in the web interface. The personal models are found by querying the database for the model collection associated with the user. The lab models are found by querying the database for the model collection associated with

the user's lab ID. There is a single collection that holds all the global models, so this collection is returned as the global models. As mentioned in Section 4.1.2, the current version and previous version are stored for each model. For each document returned by the model queries, only the current version of the model is sent to the web interface.

Save Model

The request save a model inserts or updates a model in the personal, lab, or global model collections in the database. Whether the model is being inserted or updated is determined by querying the database with the author, model name, and location as the query fields. If no results are returned from the query, the model is inserted as the previous version and the current version. If a document is returned by the query, the current version of the model is moved into the previous version in the model JSON object, and the new model that was sent in the request is inserted as the current version. The "last modified" field for the current version is updated to the current date, and the new JSON object is used to update this document in the database.

Undo Model Save

The request to undo a model save reverts a model to the previous version. The model location, model name, and author are provided with the request, and the database collection determined by the specified location is queried with the model name and author as the query fields. If no documents are returned, an error message is sent back. Otherwise the model stored in the previous version of the document is copied into the current version, and this model is sent back to the web interface. Since at this point the same model is stored in both the previous and current versions, only a single model revert is supported when a model is saved.

Run Simulation

PyNCS is used to run NCS simulations from Python. When a user launches a simulation from the web interface, the model and simulation parameter JSON objects

are submitted with the request to the daemon. Since NCS does not support JSON objects, a mapping of the simulation defined in the JSON object to PyNCS function calls is performed. In the process of converting the JSON object to PyNCS calls, a Python script file containing the appropriate PyNCS function calls is created for the following reasons: the JSON object to Python script conversion process is reused when the request to export a simulation to a Python script is made, and the initialization process of NCS only allows for one simulation to be ran per executable instance. The latter restriction was accommodated by writing the simulation script to a Python file and running the script as a linux *subprocess*, or child process. The Twisted *spawnProcess* function was used to run the script to ensure asynchronous execution. Running the simulation as a child process allows the daemon to run multiple simulations without stopping and restarting execution.

A function was added to PyNCS that has an additional parameter than the function used to add a report to the simulation. The additional parameter is a report identifier string, which is used by NCS to label the report data so the daemon and the web interface can determine which simulation generated the data. The string is a combination of the user name and report name, which creates a unique label that can be used to identify which simulation the data is associated with. Before requesting to run a simulation, the web interface creates a RabbitMQ message queue with the binding key equal to the unique report identifier. The web interface polls the message queue for report data after the simulation is launched if the user is logged in.

Python Script to JSON

The web interface allows a user to upload an NCS simulation Python script to populate the parameters in the model builder and simulation builder tabs. This requires converting the PyNCS function calls to a JSON object that follows the structure of the model and simulation parameter JSON objects used by the web interface. The PyNCS function calls build maps of the parameters that are then converted to C++ objects. To get the parameter maps, additional functions were added to PyNCS

to return the parameters as dictionaries instead of creating the C++ objects. The complete steps for the conversion process are as follows:

1. Modify the Python script file by replacing all the original PyNCS function calls with the corresponding PyNCS calls that return the parameters. At the end of the file, add the Python code for writing the parameter dictionaries to a file with a unique file name.
2. Run the modified Python script file as a child process.
3. Open the created file containing the parameter dictionaries and create the model and simulation parameter JSON object given the parameters in the file.
4. Delete the modified script and parameter dictionary files.

Upon a successful conversion, the JSON object is sent back to the web interface.

JSON to Python Script

The request to export simulation parameters to a Python script requires the daemon to take the model and simulation parameter JSON objects and find the appropriate PyNCS function calls to achieve these parameters. The conversion process is the same one used when running a simulation, but the created Python script file is sent back to the web interface as opposed to executed as a child process.

4.3.4 Data Proxy Service

Overview

The reports tab in the web interface displays report data that can be viewed as a live stream while the simulation is running or as an accumulated set after the simulation is finished. The Data Proxy Service is responsible for taking report data that it receives from NCS and sending it to the web interface so it can be viewed. A diagram of the Data Proxy Service interacting with the other system components is shown in Figure 4.7. This section describes the NCS modifications necessary for the live data

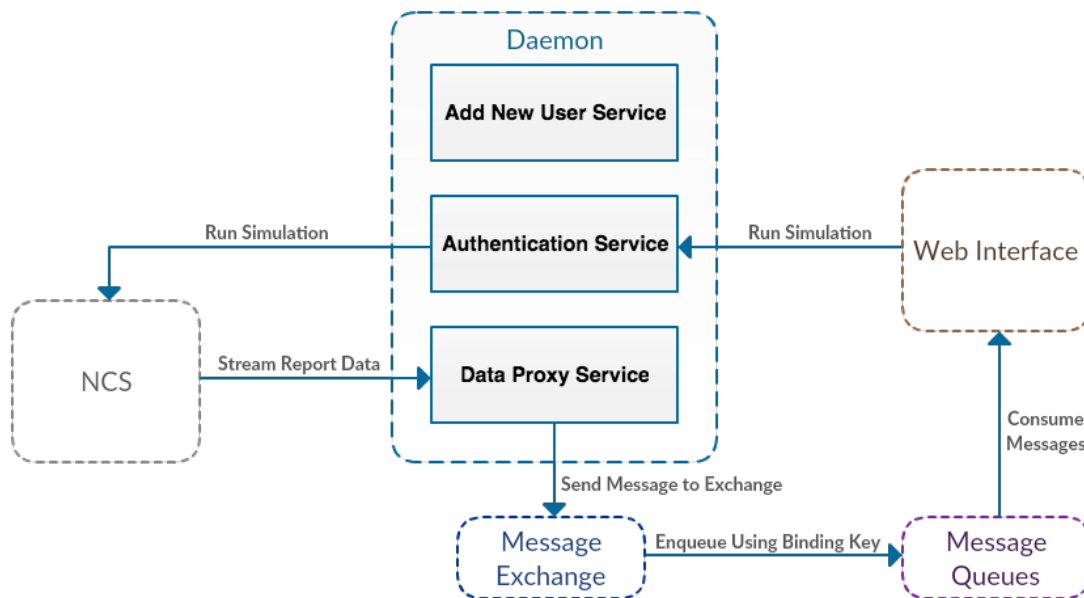


Figure 4.7: A diagram of the Data Proxy Service interacting with the other system components.

stream, and how the Data Proxy Service processes the data to offer asynchronous message passing to the web interface.

NCS Modifications

As mentioned in Chapter 2, NCS uses a publisher/subscriber system to extract all generated report data from the source at each time step. A function was added to NCS to open a TCP socket connection with the Data Proxy Service to send out any report data that was extracted. NCS supports reporting data for neuron voltage, neuron fire, input current, and synaptic current, thus, the streamed report data is typically floating point values. Protobuf is used by NCS to serialize the data before sending it, and by the Data Proxy Service to deserialize the data when it is received. Section 4.8 explains that a unique report identifier is passed in when the report is created. NCS sends the report identifier to the Data Proxy Service before it starts sending the report data, allowing the Data Proxy Service to determine where it needs to send the data so the web interface can associate it with the correct simulation.

Simulation Data Processing

When a new connection is made with the Data Proxy Service, the first message it expects to receive is the unique report identifier string. As stated before, the web interface will have created a RabbitMQ message queue with the unique report identifier as the binding key before requesting to start the simulation. The following messages received from NCS are expected to be report data. The Data Proxy Service uses Protobuf to deserialize the data, and then uses txAMQP to publish a message containing the data that uses a routing key equal to the report identifier. The RabbitMQ direct exchange uses the routing key to deliver the message to the queue with the matching binding key. If the user who launched the simulation is logged in, the web interface will receive the new messages immediately because it is polling the message queue. Otherwise, the messages will remain in the queue until the user is logged in.

4.4 Error Handling

4.4.1 Exceptions

Twisted uses pairs of callbacks and *errbacks* to handle program exceptions. If no exceptions occur, all the callbacks in the chains are executed. If an exception is raised, the appropriate errback is fired and the remaining callbacks in the chain continue with normal execution. This ensures raised exceptions are caught and do not propagate, and that the program does not terminate. Any exceptions that occur are logged using the logging system provided with Twisted.

4.4.2 Invalid Requests

For every request received from the web interface, a success or failure message is sent back. Examples of success and failure responses are shown in Figure 4.8. Each request has a unique set of possible failure responses, but all requests will receive a failure response if they do not follow the request structure. Other requests that warrant failure responses are those that lack the required data to process that request. The

```
{
  "request": "addUser",
  "response": "success"
}

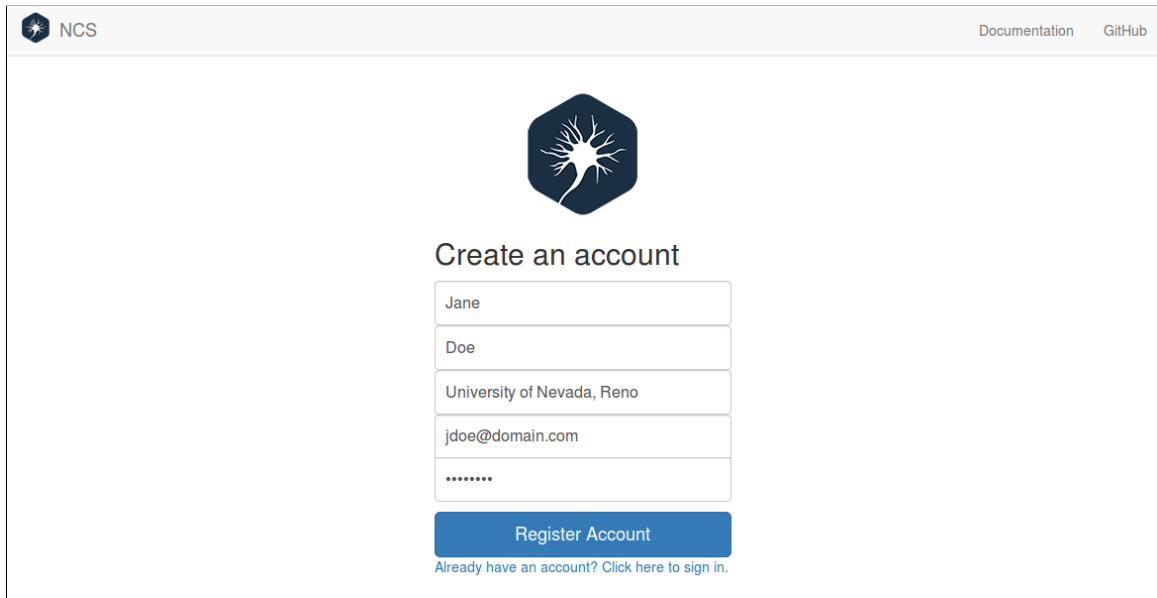
{
  "request": "addUser",
  "response": "failure",
  "reason": "User already exists."
}
```

Figure 4.8: An example of success and failure responses to requests made by the web interface.

web interface is responsible for handling the failure responses appropriately.

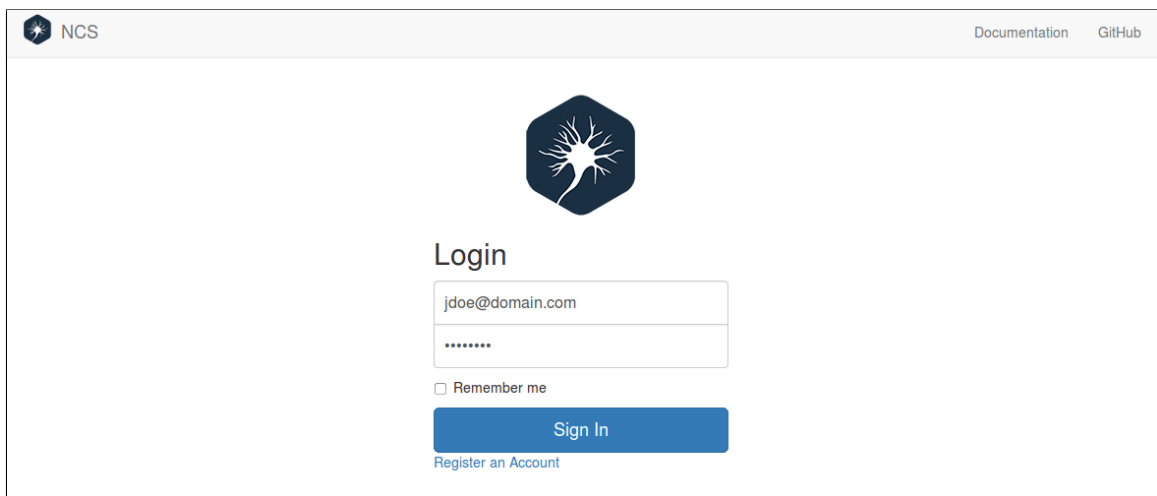
4.5 Results

The following figures are screen captures of the NCS web interface sending requests to the daemon.



The screenshot shows the NCS web interface with a header containing the NCS logo and navigation links for 'Documentation' and 'GitHub'. The main content area features a large hexagonal logo with a white tree-like structure on a dark blue background. Below the logo is the heading 'Create an account'. The form consists of five input fields: a name field with 'Jane', a last name field with 'Doe', an address field with 'University of Nevada, Reno', an email field with 'jdoe@domain.com', and a password field with seven dots. A blue 'Register Account' button is positioned below the fields. At the bottom, there is a link: 'Already have an account? Click here to sign in.'

Figure 4.9: A screenshot of the NCS web interface invoking the *add new user* request.



The screenshot shows the NCS web interface with a header containing the NCS logo and navigation links for 'Documentation' and 'GitHub'. The main content area features a large hexagonal logo with a white tree-like structure on a dark blue background. Below the logo is the heading 'Login'. The form consists of two input fields: an email field with 'jdoe@domain.com' and a password field with seven dots. Below the fields is a checkbox labeled 'Remember me'. A blue 'Sign In' button is positioned below the checkbox. At the bottom, there is a link: 'Register an Account'.

Figure 4.10: A screenshot of the NCS web interface invoking the *login* request.

The screenshot displays the NCS web interface model builder tab. The interface includes a search bar, navigation tabs (Export, Import), and a sidebar with a list of models (Cell 1, Cell 2, Cell Group 1, Cell Group 2). The main area is titled "Model Configuration" and displays the "Current Model" (NCS) with its components (IZH, NCS) and connections (Home - IZH, Home - NCS). A "Cell Parameters" table is shown with columns for Parameter Name, Type, Value, Min, Max, Mean, and StdDev. The table lists parameters such as Threshold, Resting Potential, Calcium, Calc. Spike Incr., Tau Calcium, Leak Reversal Pot., Tau Membrane, R Membrane, Leak Conductance, Capacitance, and Spike Shape 1. A "Channels" section at the bottom shows a "Calcium Dependant Channel".

Parameter Name	Type	Value	Min	Max	Mean	StdDev
Threshold:	exact	0.2	0	0	0	0
Resting Potential:	exact	0.2	0	0	0	0
Calcium:	exact	-65	0	0	0	0
Calc. Spike Incr.:	exact	8	0	0	0	0
Tau Calcium:	uniform	0	-15	-11	0	0
Leak Reversal Pot.:	uniform	0	-75	-55	0	0
Tau Membrane:	exact	30	0	0	0	0
R Membrane:	exact	30	0	0	0	0
Leak Conductance:	exact	30	0	0	0	0
Capacitance:	exact	30	0	0	0	0
Spike Shape 1:		0				

Figure 4.11: A screenshot of the NCS web interface model builder tab. The cell models displayed in the pane on the left are retrieved by invoking the *get models* request.

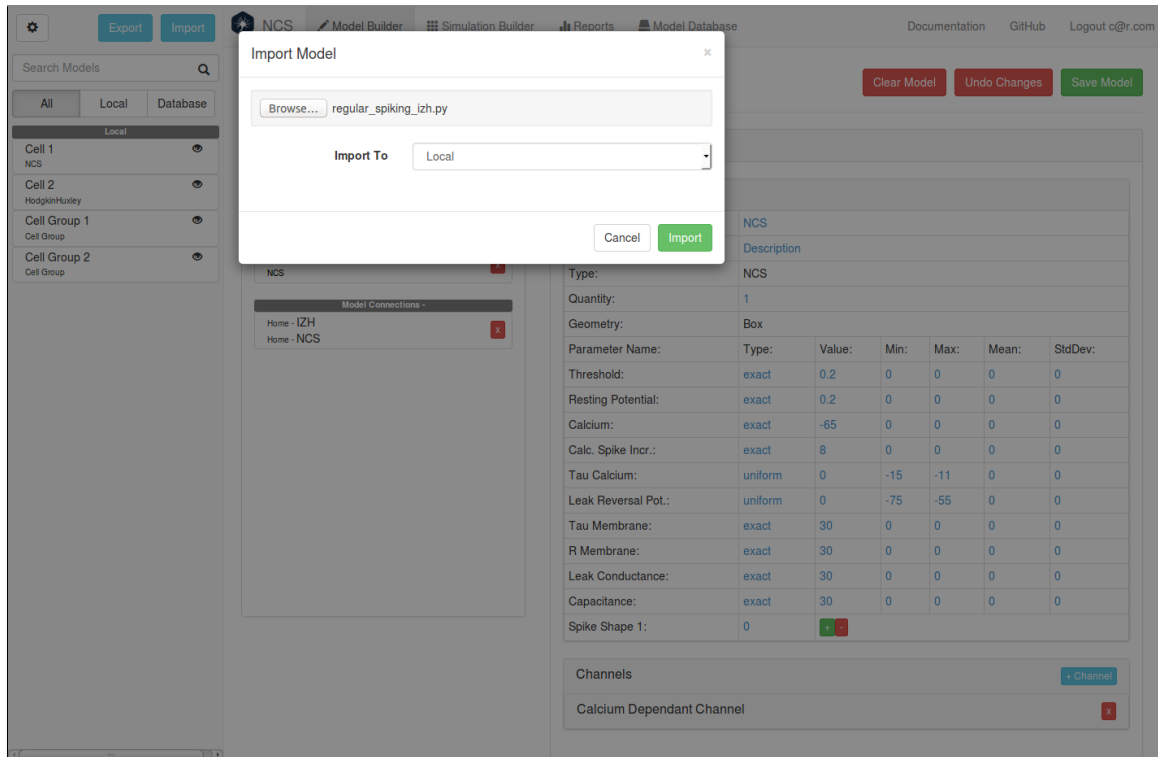


Figure 4.12: A screenshot of the NCS web interface invoking the *Python script to JSON* request.

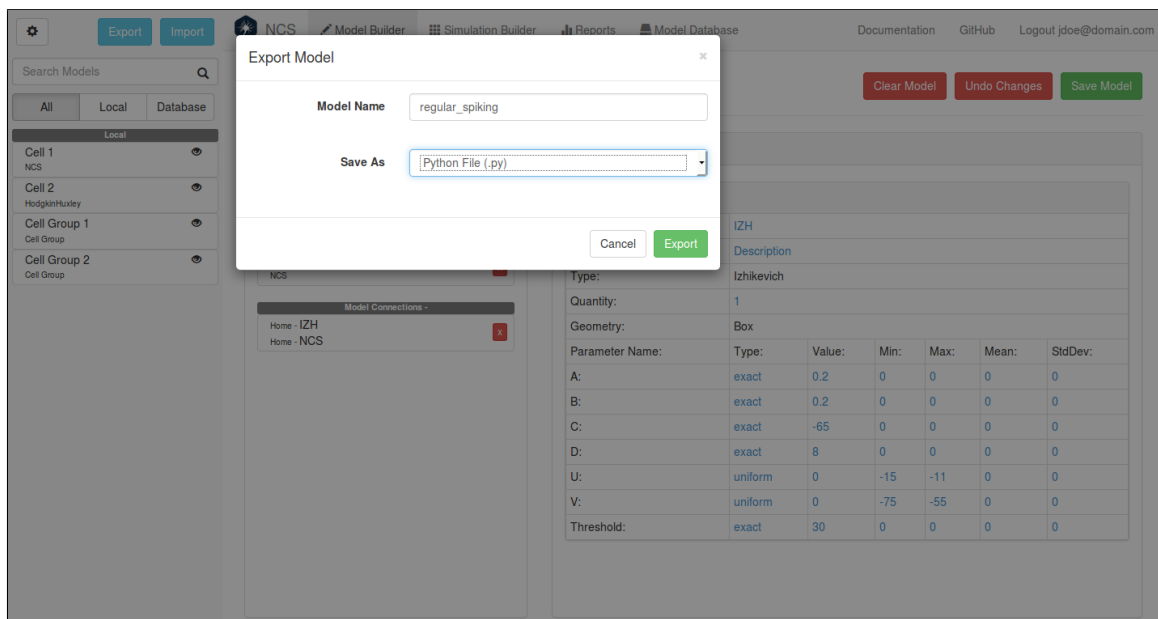


Figure 4.13: A screenshot of the NCS web interface invoking the *JSON to Python script* request.

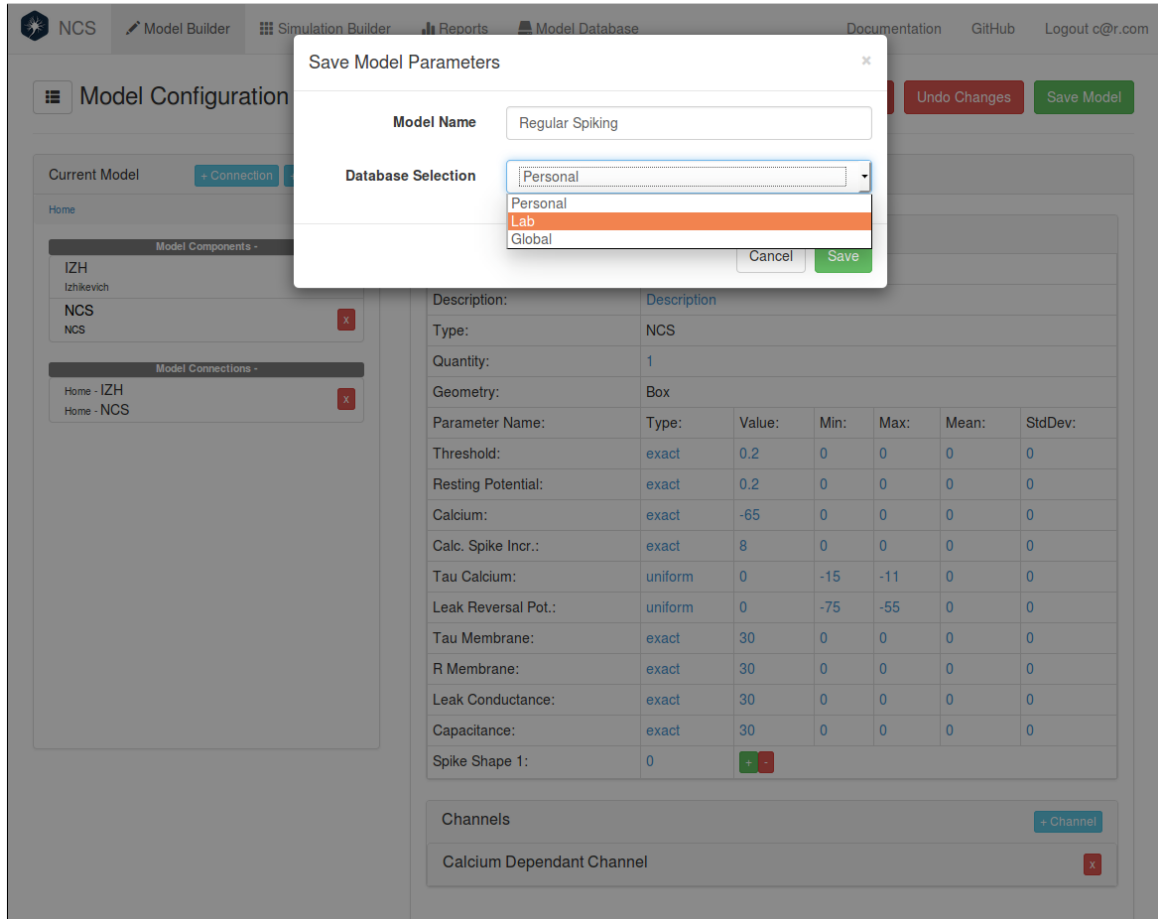


Figure 4.14: A screenshot of the NCS web interface invoking the *save model* request.

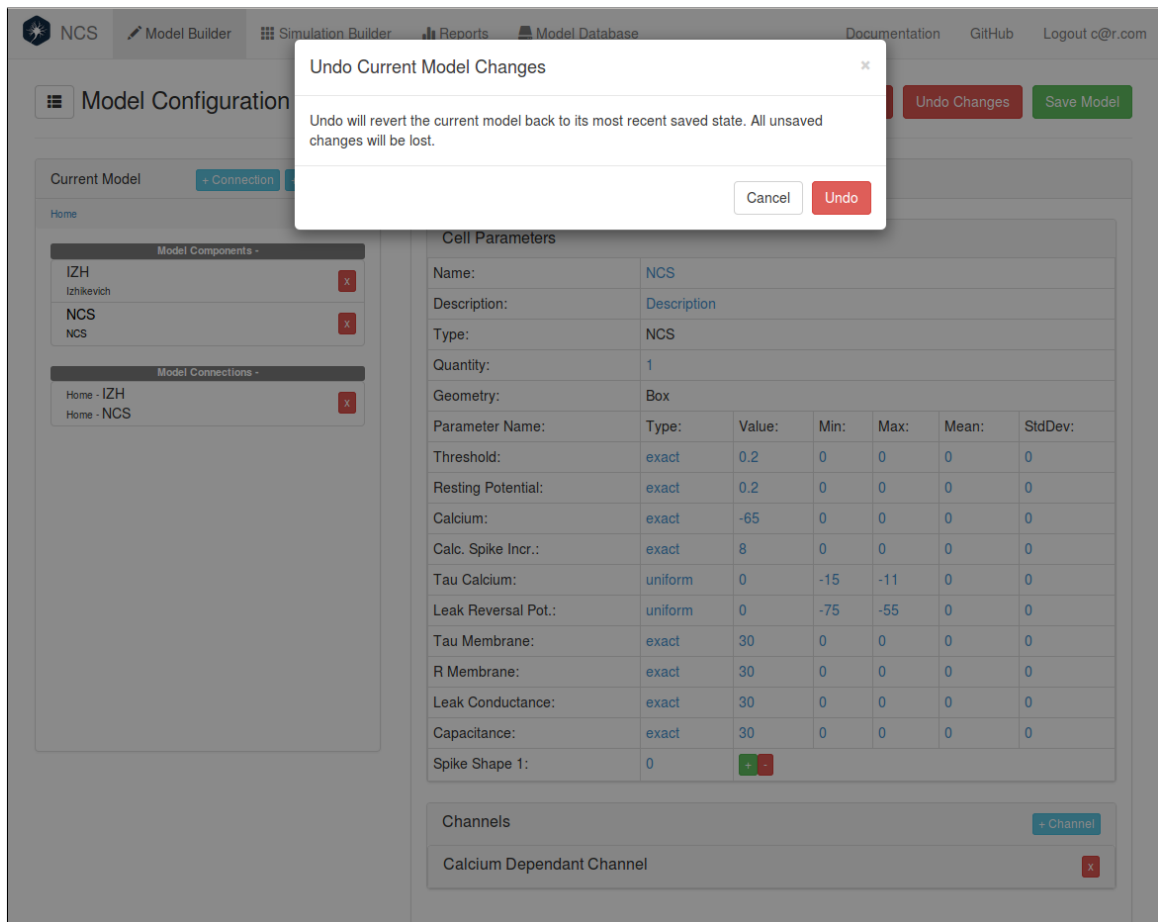


Figure 4.15: A screenshot of the NCS web interface invoking the *undo model save* request.

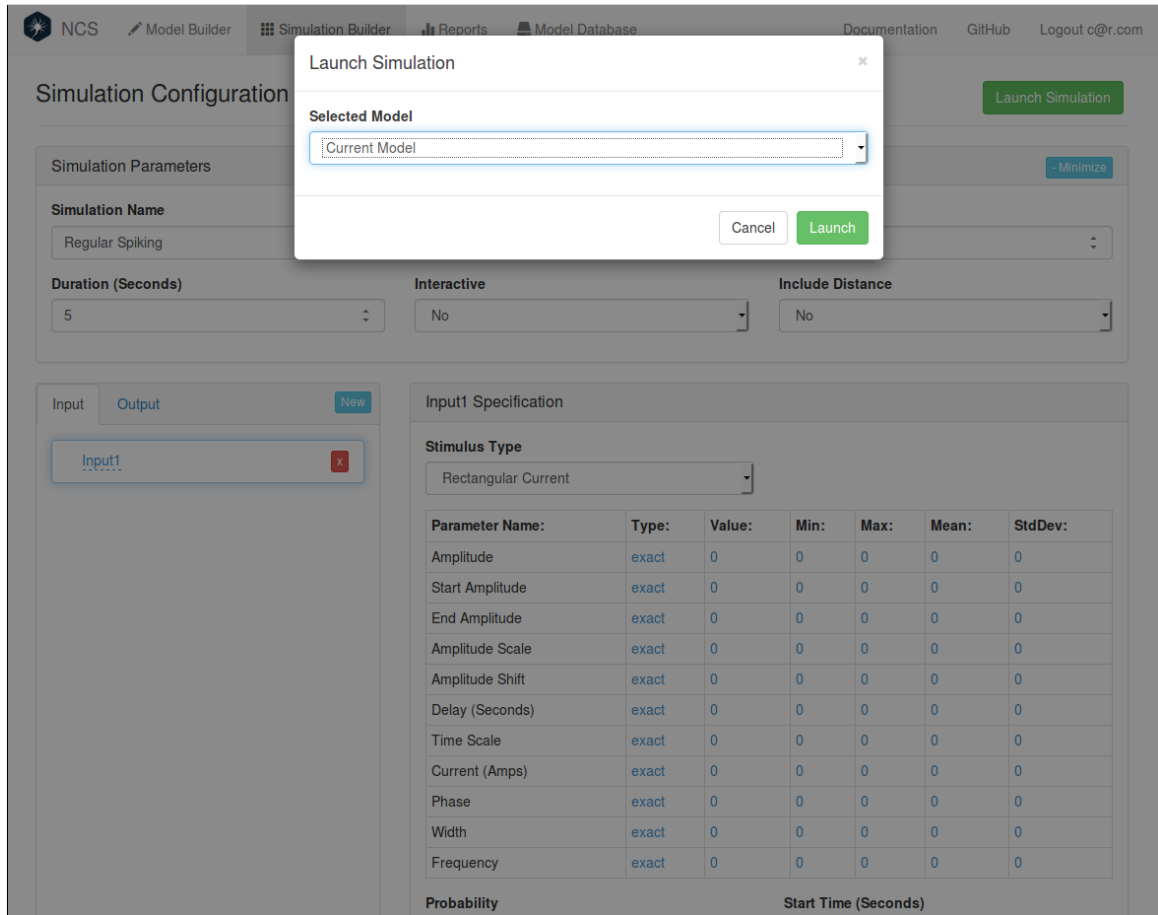


Figure 4.16: A screenshot of the NCS web interface invoking the *launch simulation* request.

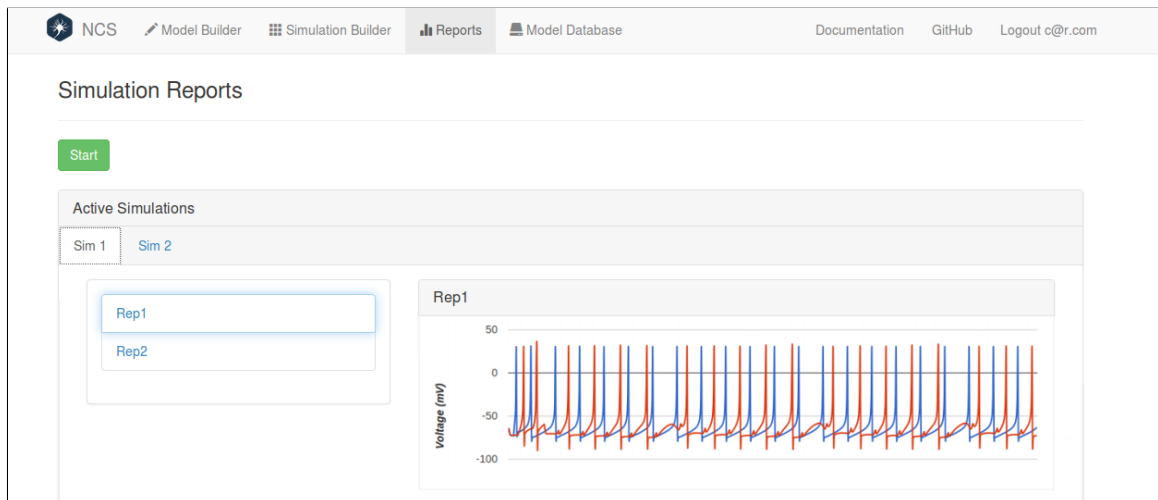


Figure 4.17: A screenshot of the NCS web interface reports tab. The data displayed for each report is received from the Data Proxy Service.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

NCS is a large scale brain simulator that has the ability of running user-created brain models with various simulation parameters. NCS also reports simulation data such as neuron spiking state and membrane voltage at each time step in a running simulation. This thesis presented the NCS daemon, a centralized service that allows users to access NCS from a web interface and manages user accounts for the web interface. More specifically, the daemon authenticates users for access to the web interface, stores brain models, runs simulations on NCS, streams NCS simulation data in real time, and performs conversions between the legacy Python script method of running simulations and simulations created using the web interface. The daemon utilizes event-driven asynchronous execution to manage a scalable client-server architecture, and it uses message queues to provide the flexibility of asynchronous sending and receiving of simulation report data.

The addition of the daemon to the NCS system has given neuroscientists who use NCS for experiments the option of having a visual tool for building brain models, configuring simulations, and analyzing simulation output data. It allows the web interface to offer a greater number of features than the Python scripts, and it provides functionality for easy transition between the methods for improved user experience. In the future, the daemon will proceed to grow through the expansion of services it provides to continue enhancing NCS usability.

5.2 Future Work

5.2.1 NCS Daemon Enhancements

A feature that would provide more flexibility in lab management would be the ability to assign user roles to the web interface accounts to restrict the type of access various users have to the functionality of the website. An example of a potential role would be an administrator, who would be the only user role allowed to add new labs and approve the addition and modification of the global brain models. Another example might be a lab administrator, who would have the privilege of adding new users to their lab and approving the addition and modification of brain models in their lab's models. The default user role, once added to a lab by the lab administrator, would have access to the remaining basic functionality of the website.

Other features that would enhance the user experience relate to the flexibility of saving and reverting models. Currently, the full previous and current versions of every model are stored, and only a single undo model save is permitted per the most recent model save. Storing only the changes that were made at each model save would reduce the amount of space used for each model, and a log of all revisions could be presented to the user when choosing to revert to a previous version.

Additionally, when a user saves a model to the lab or global models, a notification could be sent to the web interface suggesting that the models be refreshed for all users effected by the model update. Since the daemon currently does not send these notifications, it is the web interface's responsibility to detect the updates or pull the models periodically.

Although the NCS daemon was designed specifically for NCS and the NCS user interfaces, the authentication, database access, encryption, and data streaming are components that could be used in many other systems. Another enhancement could be to completely decouple the general features into a plugin that could be used with any system.

5.2.2 NCS Enhancements

One drawback of NCS is that it is not intuitive to pause and restart simulations; a simulation typically runs for the specified duration once it has been launched. It is possible to use the barriers in the messaging passing system discussed in Chapter 2 to force the simulation to wait in between groups of computations, but this is not supplied as a user feature. Making this available as a feature would be beneficial for the user, and it would also present the opportunity for easily changing the simulation parameters while the simulation is running. The ability to dynamically add stimulus is required for the completion of the virtual robot.

Another feature not supported by NCS is the utilization of neuron geometry. More details on this are discussed below in the brain visualization section.

5.2.3 Web Interface Enhancements

Virtual Robot

The virtual robot tab in the web interface was developed to show how video and audio stimulus from a virtual environment affect simulations, and to show how a human would theoretically behave given the output data of the simulation. This feature cannot be deployed without the ability to add stimulus while a simulation is running. Another hindrance of its deployment is the absence of video and audio file conversions to input values accepted by NCS. Converting the video and audio files to electrical current values could be done with Gabor filtering, a linear filtering technique similar to the human visual system in the way that it detects edges and extracts features from images. Aside from image processing, Gabor filters can also be used with single dimension signals such as speech [19], making it a reasonable method choice for the video and audio file conversions.

Brain Visualization

The brain visualization feature will be an additional tab in the web interface. The purpose of this tab will be to give a graphic visualization of the simulation components

while the simulation is running. The visualization will require the use of the streaming report data, but it also requires that NCS use geometry parameters for the simulation components, which is currently not supported.

Bibliography

- [1] E. Almachar, A. Falconi, K. Gilgen, D. Tanna, N. Jordan, R. Hoang, S. Dascalu, L. Jayet Bray, and F. Harris. NeoCortical Repository and Reports: Database and Repository for NCS. *Proceedings of the International Conference on Software Engineering and Data Engineering (SEDE-2014)*, 2014.
- [2] J. Berlinski, C. Rowe, M. D. Chavez, N. Jordan, D. Tanna, R. Hoang, S. Dascalu, L. Jayet Bray, and F. Harris. NeoCortical Builder: A Web Based Front End for NCS. *Proceedings of the 27th International Conference on Computer Applications in Industry and Engineering (CAINE-2014)*, 2014.
- [3] J. Bower and D. Beeman. *The Book of GENESIS*. 2003.
- [4] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. 2014.
- [5] N. Carnevale, M. Hines, and J. Moore. NEURON: for empirically-based simulations of neurons and networks of neurons. <http://www.neuron.yale.edu/neuron>, 2013. [Online; accessed 17-September-2015].
- [6] A. Fettig. *Twisted Network Programming Essentials*. O'Reilly Media, Inc., 2005.
- [7] A. Fiori and B. Curtis. Welcome to TxMongo's documentation! <https://txmongo.readthedocs.org/en/latest/>, 2015. [Online; accessed 16-September-2015].
- [8] FreeBSD. Processes and Daemons. <https://www.freebsd.org/doc/handbook/basics-processes.html>. [Online; accessed 17-September-2015].
- [9] W. Gerstner and W. M. Kistler. *Spiking Neuron Models. Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.
- [10] D. Goodman and R. Brette. The Brian Simulator. *Frontiers in Neuroscience*, 3(2):192–197, 2009.
- [11] Google Developers. Protocol Buffers. <https://developers.google.com/protocol-buffers/?hl=en>, May 2015. [Online; accessed 15-September-2015].
- [12] R. Hoang, D. Tanna, L. Jayet Bray, S. Dascalu, and F. Harris. A Novel CPU/GPU Simulation Environment for Large-Scale Biologically Realistic Neural Modeling. *Frontiers in Neuroinformatics*, 7, 2013.

- [13] A. Hodgkin and A. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, 1952.
- [14] E. Izhikevich. Simple Model of Spiking Neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, 2003.
- [15] L. Jayet Bray. *A Circuit-Level Model of Hippocampal, Entorhinal and Prefrontal Dynamics Underlying Rodent Maze Navigational Learning*. PhD thesis, University of Nevada, Reno, 2010.
- [16] T. Kelly. Neocortical Virtual Robot: A framework to allow simulated brains to interact with a virtual reality environment. Master’s thesis, University of Nevada, Reno, 2015.
- [17] J. McKellar. Twisted. <http://www.aosabook.org/en/twisted.html>. [Online; accessed 15-September-2015].
- [18] MongoDB, Inc. MongoDB. <https://www.mongodb.com/>, 2015. [Online; accessed 16-September-2015].
- [19] J. R. Movellan. Tutorial on Gabor Filters. 2002.
- [20] Network Computing. Client/Server Fundamentals. <http://www.networkcomputing.com/netdesign/1005part1a.html>, February 1999. [Online; accessed 15-September-2015].
- [21] M. A. Paradiso, M. F. Bear, and Connors B. W. *Neuroscience: Exploring the Brain*. Lippincott Williams & Wilkins, 2007.
- [22] PassLib. passlib.hash.bcrypt - BCrypt. <https://pythonhosted.org/passlib/lib/passlib.hash.bcrypt.html>, 2015. [Online; accessed 15-September-2015].
- [23] D. Peticolas. An Introduction to Asynchronous Programming and Twisted. <http://krondo.com/blog/?p=1247>, 2015. [Online; accessed 15-September-2015].
- [24] Pivotal Software, Inc. RabbitMQ. <https://www.rabbitmq.com>, 2015. [Online; accessed 16-September-2015].
- [25] H. E. Plesser, M. Diesmann, M. O. Gewaltig, and A. Morrison. NEST: The Neural Simulation Tool. *Encyclopedia of Computational Neuroscience*, pages 1849–1852, 2015.
- [26] Python Software Foundation. Applications for Python. <https://www.python.org/about/apps/>, 2015. [Online; accessed 15-September-2015].
- [27] Python Software Foundation. bcrypt. <https://pypi.python.org/pypi/bcrypt/1.1.0>, 2015. [Online; accessed 15-September-2015].
- [28] Python Software Foundation. txAMQP. <https://pypi.python.org/pypi/txAMQP>, 2015. [Online; accessed 16-September-2015].

- [29] Python Software Foundation. TxMongo. <https://pypi.python.org/pypi/txmongo>, 2015. [Online; accessed 16-September-2015].
- [30] E. Schwartz. *Computational Neuroscience*. Mit Press, 1993.
- [31] D.I. Standage and T.P. Trappenberg. Differences in the subthreshold dynamics of leaky integrate-and-fire and Hodgkin-Huxley neuron models. *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference*, 1:396–399, 2005.
- [32] C. L. Stanfield, W. J. Germann, M. J. Niles, and J. G. Cannon. *Principles of Human Physiology*. Pearson/Benjamin Cummings, 2011.
- [33] D. Tanna. NCS: Neuron Models, User Interface, and Modeling. Master's thesis, University of Nevada, Reno, 2014.
- [34] H. Tuckwell. *Introduction to Theoretical Neurobiology: Volume 2, Nonlinear and Stochastic Theories*. Cambridge University Press, 2005.
- [35] Twisted Matrix Labs. Twisted. <https://twistedmatrix.com/trac/>, September 2015. [Online; accessed 15-September-2015].