8-2015

Middleware and Services for Dynamic Adaptive Neural Network Arrays

Joshua Caleb Willis
University of Tennessee - Knoxville, jwill221@vols.utk.edu

Recommended Citation

Willis, Joshua Caleb, "Middleware and Services for Dynamic Adaptive Neural Network Arrays. " Master's Thesis, University of Tennessee, 2015.
https://trace.tennessee.edu/utk_gradthes/3527

To the Graduate Council:

I am submitting herewith a thesis written by Joshua Caleb Willis entitled "Middleware and Services for Dynamic Adaptive Neural Network Arrays." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

J. Douglas Birdwell, Major Professor

We have read this thesis and recommend its acceptance:

Mark E. Dean, James S. Plank

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Middleware and Services for Dynamic Adaptive Neural Network Arrays

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Joshua Caleb Willis

August 2015

To my family, friends, mentors, and teachers for their support.

Abstract

Dynamic Adaptive Neural Network Arrays (DANNAs) are neuromorphic systems that exhibit spiking behaviors and can be designed using evolutionary optimization. Array elements are rapidly reconfigurable and can function as either neurons or synapses with programmable interconnections and parameters. Visualization applications can examine DANNA element connections, parameters, and functionality, and evolutionary optimization applications can utilize DANNA to speedup neural network simulations. To facilitate interactions with DANNAs from these applications, we have developed a language-agnostic application programming interface (API) that abstracts away low-level communication details with a DANNA and provides a high-level interface for reprogramming and controlling a DANNA. The library has also been designed in modules in order to adapt to future changes in the design of DANNA, including changes to the DANNA element design, DANNA communication protocol, and connection. In addition to communicating with DANNAs, it is also beneficial for applications to store networks with known functionality. Hence, a Representational State Transfer (REST) API with a MongoDB database back-end has been developed to encourage the collection and exploration of networks.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Dynamic Adaptive Neural Network Arrays (DANNAs) [42] [1] are neuromorphic systems that exhibit spiking behaviors and can be designed using evolutionary optimization. Array elements are rapidly reconfigurable and can function as either neurons or synapses with programmable interconnections and parameters. Visualization applications can examine DANNA element connections, parameters, and functionality and evolutionary optimization applications can utilize DANNA to speedup neural network simulations. To facilitate interactions with DANNAs from these applications, we have developed a language-agnostic application programming interface (API), or middleware, that abstracts away low-level communication details with a DANNA and provides a high-level interface for reprogramming and controlling a DANNA. We explored several inter-process communication methods, including: remote procedure calls (RPCs) [49], Simplified Wrapper and Interface Generator (SWIG) [50], and a Representational State Transfer (REST) [20] API. We decided upon the REST API because clients can communicate regardless of their language and it is simpler to set-up and use. An application can send HTTP requests to control, program, and monitor a DANNA device. The application programming interface (API) has also been designed in modules in order to adapt to future changes in the design of DANNA, including changes to the DANNA architecture, DANNA communication protocol, and connection. We are able to successfully communicate with a DANNA device via HTTP requests. Future middleware work includes keeping an element change log, generating random networks with a percentage utilization measurement, automatically placing networks within other networks using simulated annealing, and communicating with a DANNA device using a socket stream.

In addition to communicating with DANNAs, it is also beneficial for applications to store networks with known functionality. We looked at storing networks in Git [21] and in a NoSQL, MongoDB [27], database. We decided upon the NoSQL database because it can distribute computation across a cluster and allows for common database functionality, including searching and indexing, which is not available in Git. Hence, a DANNA library, or a REST API with a MongoDB database, has been developed to encourage the collection and exploration of networks. Networks and constraints can be retrivied, added, updated, and deleted from the database. Further work on the library includes compressing responses in the server before they are sent, adding filtering, adding searching, adding a more advanced authorization method, dividing long responses into pages, implementing sharding, and developing a website for quick access to the database.

To begin, we look at background research in our neuromorphic computing group, including Neuromorphic-Inspired Dynamic Architecture (NIDA) and DANNA, previous research in other research groups, including instruction set architectures and software frameworks, and tools used in this project (see Chapter 2). Then, we describe the guidelines for the project in Chapter 3. We detail the software that implements the guidelines in Chapter 4. Next, we present the design and implementation of the DANNA library in Chapter 5. Finally, we summarize the project in Chapter 6.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Neuromorphic-Inspired Dynamic Architecture

A Neuromorphic-Inspired Dynamic Architecture (NIDA) [42] [43] [44] [46] [45] [3] is a novel neural network design exhibiting event-based spiking activity. Information in the network is distributed across the network in neurons as charge and in synapses as delay. Elements, neurons and synapses, are positioned in 3D space. Neurons accumulate charge and fire when a threshold has been reached; then, they enter a refractory period, where they can accumulate charge, but cannot fire. Synapses are connections between neurons that allow information to flow; charge is delayed along a neuron, allowing them to store information. Distance of a synapse between two neurons determines the delay. An event-driven simulation evolves the network state over time as neurons collect charge and both neurons and synapses fire. Long-term potentiation and long-term depression modify synapse weights during simulation [2].

Evolutionary Optimization

Evolutionary optimization (EO) [44] [46] [45] is used to create a NIDA network. EO fits a network to a task by training over the parameters and the structure of the network. EO uses a population of networks to find the most suitable network for a task. The initial network population is randomly initialized. Then, evolution of the networks occur through mutation and crossover operators, which develop child networks. A fitness function, implemented by the user, determines how well a network performs for a task. The best networks are then kept for the next iteration of evolution. Networks are then created by defining a fitness function for a task and then letting EO design the network for that task. EO is a random search method and, therefore, can converge very slowly to a solution. Additionally, when scoring a network, EO simulates it and uses the simulation to modify the probabilites used to create new network populations.

The Simulation Acceleration Problem

NIDA network simulation occurs in software. Originally, NIDA networks were simulated on a single general purpose CPU, but simulation on a general purpose CPU is slow. Therefore, simulations were distributed across a cluster of computers and executed in parallel to speed them up. It is also advantageous to examine other acceleration methods for network simulation.

2.1.2 Dynamic Adaptive Neural Network Array

A Dynamic Adaptive Neural Network Array (DANNA) [11] [1] is a neural network model that exhibits event-based spiking activity , can be reprogrammed quickly, can be created using evolutionary optimization, and is similar to the NIDA model but in a 2D grid space. Because DANNA elements are fixed to a 2D grid, connections between them are restricted to the grid, unlike in DANNA; specifically, elements connect to their nearest neighbors in DANNA. Similar to NIDA neurons, DANNA neurons collect charge and fire when a programmable threshold has been reached and the neuron is not in its refractory period. DANNA synapses are comparable to NIDA synapses; they hold a programmable weight and are used as connectors

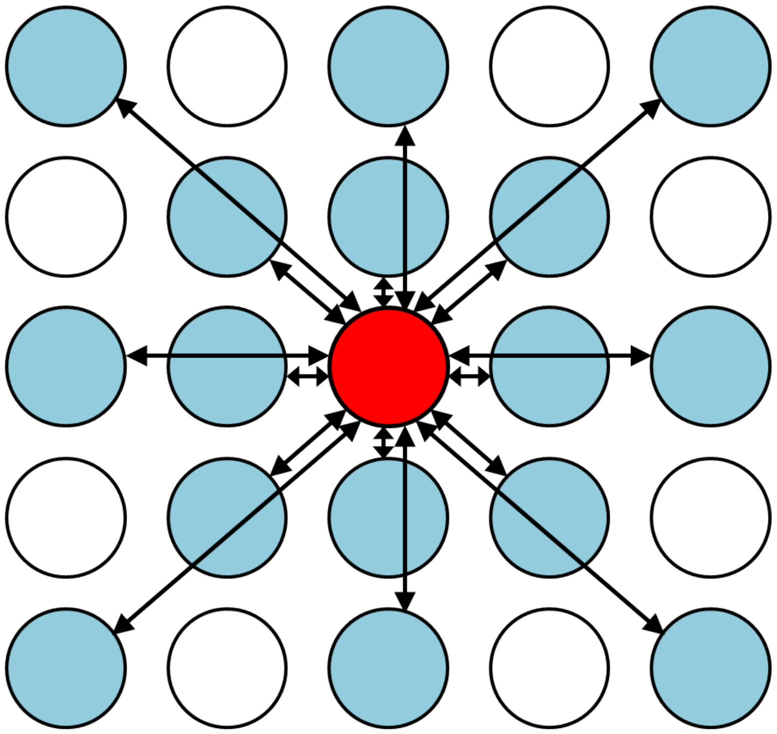


Figure 2.1: The element, in red, is connected to the elements highlighted in blue. The eight closest blue elements are the first ring. The next eight are the second ring. Image courtesy of Jason Chan.

between neurons. However, DANNA synapses differ in two areas: DANNA synapse delays are programmable and are not determined by the length of the synapse in the space, and DANNA synapses can also connect to other synapses. Elements, neurons and synapses, are connected to each other in a ring connection scheme as shown in Figure 2.1. To identify common connections between elements with the ring conection scheme in a DANNA grid, the connections are labeled. For example, Figure 2.2 shows the labeled ring connections when applied to a 4x4 grid of elements. If directions are specified using compass directions with north being up, then the top-left element in the grid can connect to its east neighbor one-hop away by enabling output 2. Similarly, the eastern neighbor can enable input 2 to receive input from the top-left element. A DANNA device includes a programming interface that allows for control of the DANNA from a connected host computer. DANNAs are currently implemented on hardware using Field Programmable Gate Arrays (FPGAs) and provide another solution to the simulation acceleration problem.

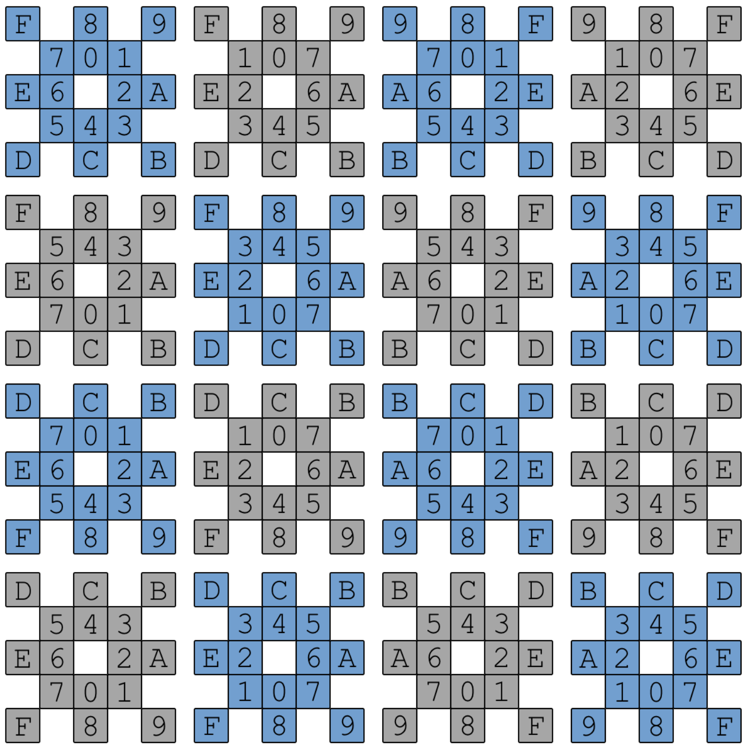


Figure 2.2: Elements can enable ports in the hex range 0x0-0x7 to connect to elements one hop away and 0x8-0xf to elements two hops away. Image courtesy of Adam Disney.

2.2 Related Work

The neuromorphic computing research community has explored many different neural network models, middleware and API designs, communication protocols, and device implementations. The software created is usually custom built for the neuromorphic system. However, researchers have explored generic systems in various contexts, including instruction set architectures (ISAs), communication protocols, and software frameworks.

2.2.1 Neuromorphic Instruction Set Architecture

The Neuromorphic Instruction Set Architecture (NISA) [23] introduced an instruction set architecture, a set of encoded instructions to control a computer, for neuromorphic systems. Hashmi et al. describe an instruction set for performing operations with neuromorphic models that hides the underlying neuromorphic system. It also creates an important divide between the underlying neuromorphic system and the learning algorithms that are used to create networks for the system. Networks are represented in an Extensible Markup Language (XML) [7] format and can be created in the Aivo integrated development environment (IDE), which allows for subnetworks to be connected to create larger networks. Once a network has been created or imported into the IDE, it can simulated on a CPU or GPGPU.

2.2.2 Address-Event Representation

In 1992, researchers at the California Institute of Technology introduced a novel communication protocol, the address-event representation (AER), for their VLSI neural network model [25]. Before, the typical neuromorphic protocol sampled all network elements for activity at a point in time and outputted their weights. However, they found that at many time cycles, network elements had little to no activity. Therefore, they introduced a protocol that places addresses of firing elements on a bus when they fire, instead of at every time cycle. This protocol reduced the amount of data they received from the device. It is a widely used output standard for neuromorphic devices in the neuromorphic computing community. Our output system is similar to AER. Element addresses of firing elements are outputted; however, we only output from designated output elements. While we do not output all fire events, our fire events include the weight of the charge and time-stamp. We have also recently expanded the output to included state (element accumulator contents, number of fires since the last capture, and the number of fires in the fire queue) from every element in a DANNA grid, which allows us to see all fire events and more closely resembles the AER protocol.

2.2.3 Software Frameworks

The neuromorphic community has developed two software frameworks, PyNCS and PyNN, that contain a generic neural network model and a simulation and monitoring interface for neuromorphic systems. Both frameworks are Python [41] modules and, hence, are highly portable.

In order for PyNN, a product of the Fast Analog Computing with Emergent Transient States (FACETS) project, to support a neuromorphic system, a neuromorphic developer maps input from the PyNN API to their neuromorphic system and output from their neuromorphic system to the PyNN API [10]. PyNN does not consider the low-level stack - the low-level APIs, device drivers, and communication protocols - above the hardware; it only represents the data it is outputting and the data it is receiving. A low-level stack implementation for one neuromorphic system may share similarities with other neuromorphic system implementations. For example, devices with different designs could use the same communication protocol. In that case, it would be beneficial to use the same code; however, PyNN back-end libraries are independent and do not share code. Futhermore, documentation for implementing a back-end for PyNN is incomplete; a mark-down file is available in the PyNN code-base, but it is partially completed. Articles about PyNN discuss the implementation at a high-level, but do not detail the software that needs to be implemented for a neuromorphic system to be usable by PyNN.

The PyNCS [48] design is a modular software framework can more easily adapt to changes in neuromorphic systems. The modules are interfaces that can be extended or combined together to provide support for new neuromorphic system interfaces. For example, PyNCS has a configuration module, similar to our constraints module, that identifies device functionality and a communication module, similar to our packets and streams, that handles transmissions with a device. Once these modules have been implemented for the device the rest of the PyNCS library can be used with the device, including simulation functionality and network modeling.

Because both frameworks provide a high-level interface for simulating and monitoring neuromorphic systems, both can be used to compare the results of tasks on the systems they support. For example, the FACETS project and its successor project, BrainScaleS, have developed a benchmark library for certain tasks using the PyNN framework and the multiple neuromorphic systems it supports [5].

While the PyNCS developers believe their software framework is more comprehensive and modular than PyNN, PyNN is supported by more research groups and projects. Currently, PyNN has documented support [40] for NEURON [33], NEST [31], PCSIM [37], and Brian [4], while PyNCS has documented support [39] for their own neuromorphic systems.

PyNN's generic network and element models seem to cover the functionality needed for DANNA. However, it appears to lack documentation for implementing a new neuromorphic system. PyNCS follows a modular design, but is not as well supported as PyNN by the neuromorphic computing community.

PyNN Back-ends

Neuromorphic systems that implement the PyNN API are referred to as back-ends, because neuromorphic systems could be implemented in software or hardware [9].

PyNN is used as a high-level interface for the FACETS neuromorphic wafer system. The FACETS neuromorphic wafer system is composed of programmable neuromorphic ASICs, named High Input Count Analog Neural Network chips (HICANNs), that implement a collection of neurons and synapses. Packets are communicated to the the wafer system by off-wafer ASICs, Digital Network Chips (DNCs), and are routed by FPGAs. A PC containing a low-level software communication interface is linked to the wafer system via a multi-Gigabit Ethernet connected to the FPGAs. The communication interface follows the common network protocol Automatic Repeate-reQuest (ARQ) [18], which ensures that packets are reliably sent between two connected devices by having the receiver send acknowledgements of requests to the sender.

The SpiNNaker system [24] is a distributed, parallel neuromorphic system composed of multiple neuromorphic chips, where high-level simulation and network design is facilitated through PyNN. SpiNNaker has a low-level API that resides directly above the device drivers that communicate to the neuromorphic chips. The low-level API includes functionality for maintaining the neural network model in SpiNNaker's distributed chip system. System settings, neurons, and connections are located in formatted files that are passed as input to the low-level API. Output files, including neuron data and synaptic weight files, are produced from the low-level API. A debugging module, a system activity GUI, and a PyNN mapper have been developed directly above the low-level API. MATLAB and SoC Designer simulations have been run to confirm correctness of the SpiNNaker platform. In addition, simulations can be developed in PyNN for result comparison to other neuromorphic systems.

2.3 Tools

2.3.1 Build System

To ensure that our middleware could be compiled on various platforms, we assessed build systems and IDEs for ease of use, configuration readability, and license agreement. Specifically, we reviewed GNU Make, Tup, Premake 4, Eclipse, and NetBeans.

GNU Make is not inherently cross-platform; users must take special precautions through tedious platform checks to ensure that all operating systems are adequately supported. GNU Make has been shown to perform poorly with a recursive pattern in large projects by Miller [26], but our code base is not large enough to have this issue.

Tup is a file-based build system that its developers claim performs better than GNU Make [52]. But, we had trouble using Tup on OS X, and there are issues with the license. Tup uses a GPL-v2 license or a commercial license. The GPL-v2 license requires projects using Tup to be open-source, modifiable, and freely redistributable. The terms of agreement for the commercial license are determined on a case-by-case basis. Because of license issues, we chose not to proceed with Tup.

Premake 4 generates makefiles and Visual Studio, Xcode, Code::Blocks, and CodeLite project files through the use of a Lua build configuration file [38]. Premake's documentation and examples are readable and ease the cross-platform build process. In addition, the Lua configuration files are easy to read and understand.

Eclipse is an interactive development environment (IDE) primarily used in Java development, but it also supports C++ through the C/C++ Development Kit (CDT), a collection of plug-ins that enhance the Eclipse workbench [16]. CDT generates a makefile for a project created in Eclipse. A user can also provide their own makefile for Eclipse to use [15].

NetBeans is a modular IDE; plugins allow for functionality changes by installing or uninstalling plugins [32]. NetBeans uses generated makefiles for C++ projects.

Both Eclipse and NetBeans have support for the popular platforms, Linux, OS X, and Windows. In addition, they both generate makefiles for the platforms. In both cases, external software needs to be installed on Windows, usually Cygwin, so that Windows has support for GCC and GNU Make. However, both are large pieces of software taking up a significant amount of disk space. With our desire to keep the middleware lightweight, it is best to avoid an IDE for its build system.

GNU Make is not developer friendly for cross-platform development, we have license issues with Tup, and IDEs introduce a large amount of overhead. Therefore, we have decided to use Premake to generate platform independent, portable GNU makefiles for our middleware.

2.3.2 Project File Format

In a typical software project, a specific file format is agreed upon for configuration files and other project specific files. We evaluated custom ASCII, XML, and JSON file formats for readability, portability, disk consumption, and future usage.

We believe that a custom ASCII format would not be advantageous because a user must learn new syntax and semantics and create a custom parser so that the file can be read into memory. In our case, a custom ASCII file used the least amount of disk space. It consumes the least amount of white space and can compress its formatting because it does not have to adhere to a specific syntax.

Extensible Markup Language (XML) is a standard file format; its syntax and semantics are similar to Hypertext Markup Language (HTML). In HTML, tags such as <div> and <p> are predefined. However, tags can be freely named and created in XML. This lack of standard semantic meaning can lead to readability issues and is the primary reason we will not be using XML. Linus Torvalds recently produced a more eloquent argument against the use of XML, "XML is crap. Really. There are no excuses. XML is nasty to parse for humans, and it's a disaster to parse even for computers. There's just no reason for that horrible crap to exist" [51].

JavaScript Object Notation (JSON) is a standard that follows the anonymous object, array, string, and value syntax of JavaScript. Its structure is based on key-value pairs; therefore, the semantics of the JSON structure are easy to grasp. JSON is increasingly being used in applications, often replacing XML, INI, or other file formats. For example, it is the primary data format used in the web API community and has recently grown more popular than XML, as can be seen in Figure 2.3 [22]. In addition, most programming languages have libraries for parsing JSON. Because of JSON's standard syntax and the need to adapt the data representation to a key-value pair structure, JSON consumes more disk storage than custom ASCII and, hence, consumes more network resources. Therefore, it is beneficial to be able to create multiple representations of the same JSON; the standard JSON would be

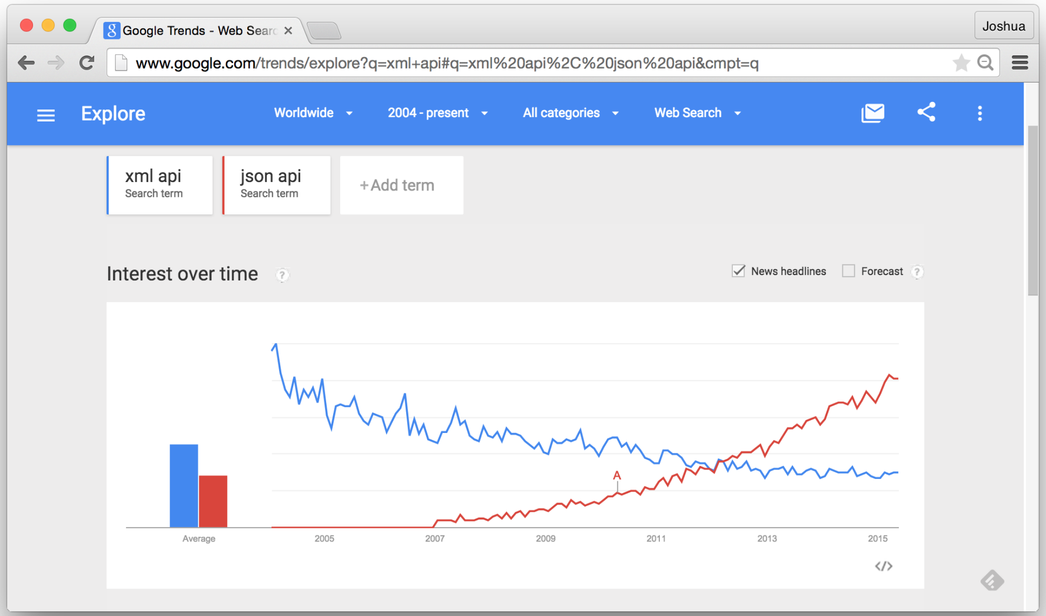
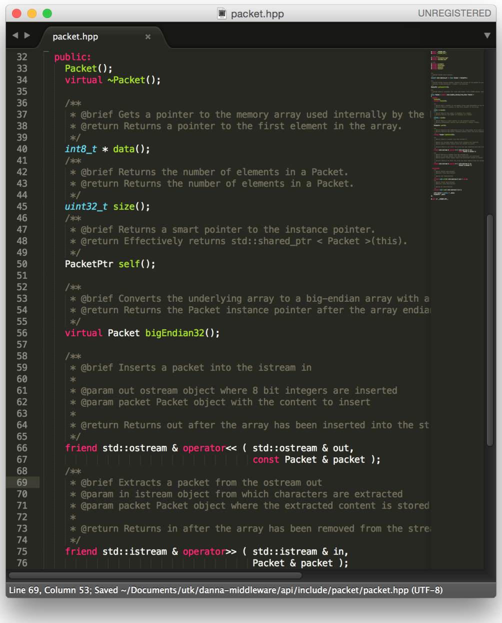


Figure 2.3: Data from Google Trends show that the number of articles covering JSON APIs has exceeded the the number of articles covering XML APIs. This suggests that JSON APIs have become more prevalent than XML APIs.

human-readable, while a compressed version would save disk storage. JSON can be compressed by minimizing the JSON structure to one line and then using Gzip to convert the file to binary.

In conclusion, custom ASCII file formats do not have a standard semantic meaning and require a custom parser. XML's structure can lead to readability issues. JSON is commonly used in the web application community, is readable, is supported in many programming languages, and can be compressed for space savings. We believe it will be continue to be used heavily in the years to come and have adopted it as our file format.

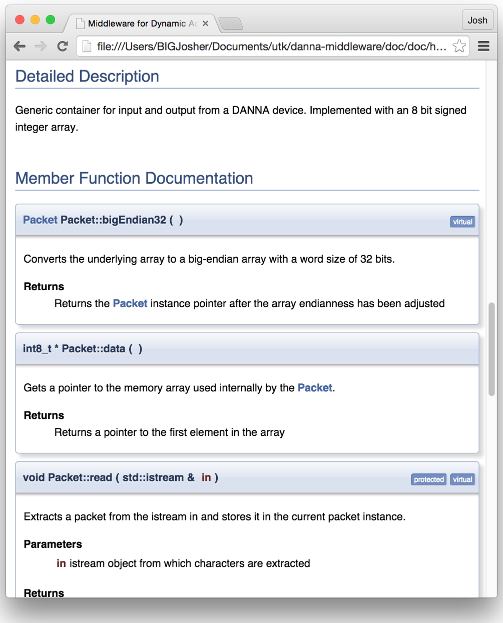


packet.hpp

```
 public:
   Packet();
   virtual ~Packet();

   /**
    * @brief Gets a pointer to the memory array used internally by the
    * @return Returns a pointer to the first element in the array.
    */
   int8_t * data();
   /**
    * @brief Returns the number of elements in a Packet.
    * @return Returns the number of elements in a Packet.
    */
   uint32_t size();
   /**
    * @brief Returns a smart pointer to the instance pointer.
    * @return Effectively returns std::shared_ptr < Packet >(this).
    */
   PacketPtr self();

   /**
    * @brief Converts the underlying array to a big-endian array with a
    * @return Returns the Packet instance pointer after the array endia
    */
   virtual Packet bigEndian32();

   /**
    * @brief Inserts a packet into the istream in
    *
    * @param out ostream object where 8 bit integers are inserted
    * @param packet Packet object with the content to insert
    *
    * @return Returns out after the array has been inserted into the st
    */
   friend std::ostream & operator<< ( std::ostream & out,
                                      const Packet & packet );
   /**
    * @brief Extracts a packet from the ostream out
    * @param in istream object from which characters are extracted
    * @param packet Packet object where the extracted content is stored
    *
    * @return Returns in after the array has been removed from the stre
    */
   friend std::istream & operator>> ( std::istream & in,
                                      Packet & packet );
```

Line 69, Column 53; Saved ~/Documents/utk/danna-middleware/api/include/packet/packet.hpp (UTF-8)

(a) Source code with Doxygen style commenting



file:///Users/BIGJosher/Documents/utk/danna-middleware/doc/doc/h...

Detailed Description

Generic container for input and output from a DANNA device. Implemented with an 8 bit signed integer array.

Member Function Documentation

Packet Packet::bigEndian32 ( ) virtual

Converts the underlying array to a big-endian array with a word size of 32 bits.

Returns
Returns the Packet instance pointer after the array endianness has been adjusted

int8_t * Packet::data ( )

Gets a pointer to the memory array used internally by the Packet.

Returns
Returns a pointer to the first element in the array

void Packet::read (std::istream & in) protected virtual

Extracts a packet from the istream in and stores it in the current packet instance.

Parameters
in istream object from which characters are extracted

Returns

(b) Doxygen generated HTML from source code commenting

Figure 2.4: Figure (a) shows the commenting style Doxygen expects. Figure (b) shows the HTML that Doxygen generated by parsing the commenting style.

2.3.3 Documentation

Documentation can be generated from the source code using Doxygen [12], which we have configured to output in HTML. Figure 2.4 shows a sample of the HTML documentation generated from comments in our source code.

2.3.4 Unit Testing: Catch

Catch [6] is a header-only unit testing framework for C++ and Objective-C. It can be used to follow either the Test Driven Development (TDD) or Behavioral Driven Development (BDD) methodology.

2.3.5 Representational State Transfer (REST)

Representational State Transfer (REST) was introduced with version 1.1 of the Hypertext Transfer Protocol (HTTP) in 2002 [20]. When a user is browsing the Internet, he or she types an address in the URL bar of the Internet browser. The base (first) part of this address names a server; a slash and further addressing denotes a resource on the server. For example, "http://www.google.com/" indicates the "google.com" server. When a user clicks "Go" or hits enter after the URL entered, the browser sends a request for the resource. In our example, a HTTP GET request is sent for "http://www.google.com." The server then responds with a representation of the root, or default, resource on the server. In this case, it responds with an Hypertext Markup Language (HTML) page. This is a request for a representation of the resource on the server through a transfer (see 2.5), hence Representational State Transfer.

2.3.6 Mongoose-Cpp

Mongoose-Cpp [29] is a C++ server framework that follows the Model-View-Controller (MVC) design pattern. It has a router that registers callbacks to server resources. It also includes several examples servers, including a REST API that returns Javascript Object Notation (JSON) responses.

2.3.7 Node.js

Node.js [35] is a JavaScript server framework running on the Chromimum V8 JavaScript engine. It has a variety of built-in modules, or libraries, that can be used immediately. Node's memory footprint is 40MB-50MB, which we consider small enough to be termed lightweight.

Node has an accompanying command-line interface (CLI) package manager, Node Package Manager (NPM), that pulls in external libraries for use inside a project. We used the package manager to install Express.js and Mongoose.js in our project.

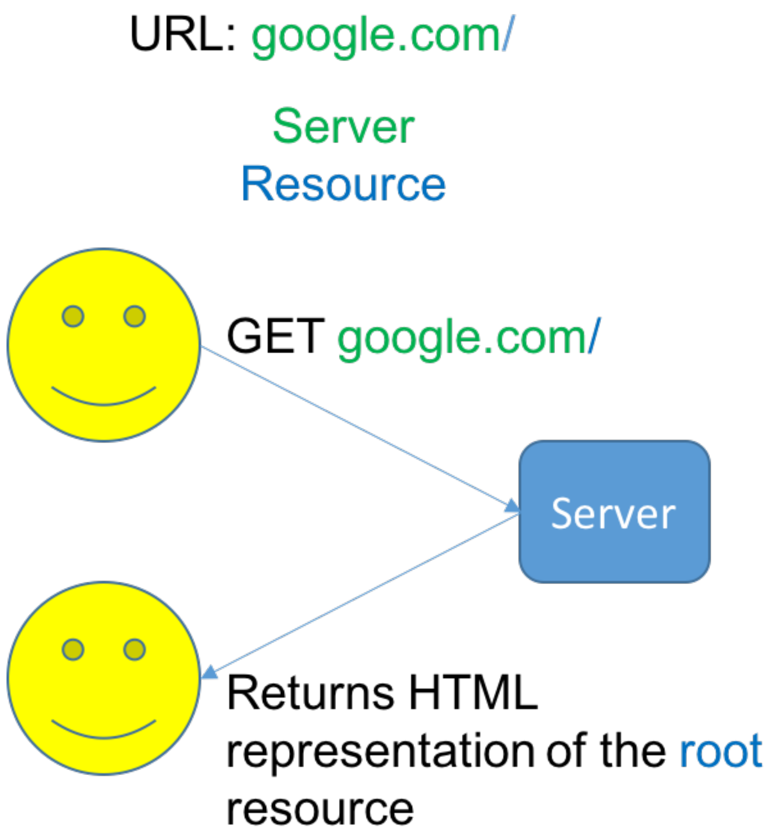


Figure 2.5: A request is sent for a resource and a representation of that resource is returned.

Express.js [17] is a Node.js module that provides a router that handles routes, which are a combination of a HTTP request action and the requested resource. Mongoose.js [30] is a Node.js module that connects and interacts with a MongoDB database.

2.3.8 MongoDB

MongoDB [27] implements a NoSQL database that stores Binary JSON (BSON), provides standard database functionality, and is deployable in a distributed environment using sharding. Data are stored in a document (analogous to a SQL row/tuple) inside a collection (analogous to a SQL table/relation).

In a small database setup, one computer would be used to host a database. If the computer begins to run low on disk space or the CPU is consistently maxed out because of a high number of queries, then further resources are added to the computer. In constrast, NoSQL has adopted a distributed philosophy. A database is broken into pieces and spread accross a cluster of computers. MongoDB refers to the distribution of its database as sharding [28].

2.4 Summary

We began by discussing background research in our neuromorphic computing group, including NIDA, a 3D-space software-only neuromorphic computing system, and DANNA, a 2D-space software and hardware neuromorphic computing system. We then looked related work from other research groups, including a neuromorphic computing instruction set architecture, a common output format, and software frameworks. Finally, we overviewed the tools that were used in this project. Next, we will look at the requirements of the project in Chapter 3

Chapter 3

Specification

A specification states guidelines and requirements for the design of a system. It helps ensure all project goals are completed and implemented in a timely fashion. The following specification details the representation of DANNA in software, the functionality required of the middleware, the communication protocol to and from the DANNA device, and the API for the DANNA library. Each listed point is seen as a requirement for the project.

1. DANNA Application Programming Interface (API)

    1.1. Element

    1.1.1. An element may be configured as any one of a neuron, a synapse, a pass-thru, or a central pattern generator (CPG).

    1.1.1.1. An element is connected to neighboring elements using input/output (I/O) signal lines arranged in rings of eight (8) as seen in Figure 3.1.

    1.1.1.2. An input signal line has two states that may be set: enabled and disabled.

    1.1.2. Neuron

    1.1.2.1. A neuron has a programmable threshold parameter.

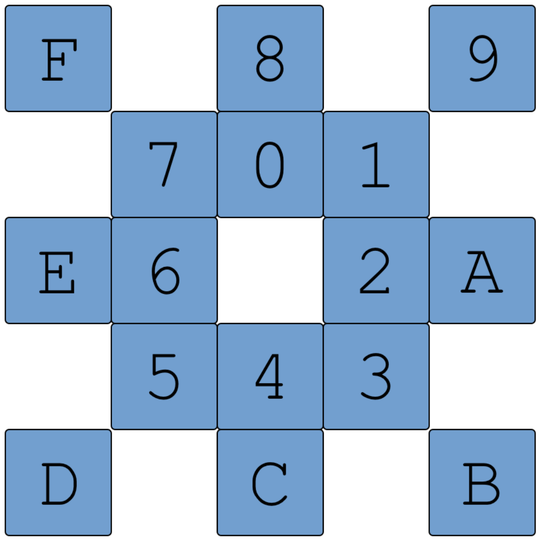


Figure 3.1: The interior ring, hex labels 0x0-0x7, allow an element to connect to neighbors one hop away. The exterior ring, hex labels 0x8-0xF, allow an element to connect to neighbors two hops away. Image courtesy of Adam Disney.

1.1.2.2. Any combination of input signal lines of a neuron may be enabled.

1.1.3. Synapse

1.1.3.1. A synapse has a programmable delay.

1.1.3.2. A synapse has a programmable weight.

1.1.3.3. Exactly one input signal line of a synapse must be enabled.

1.1.3.4. Exactly one output signal line of a synapse must be designated as monitored by the synapse for long-term potentiation / long-term depression (LTP / LTD).

1.1.4. Pass-Thru

1.1.4.1. Exactly one input signal line of a pass-thru must be enabled.

1.1.4.2. Information item: An event received by a pass-thru exits the output signal line after one network clock cycle delay.

1.1.4.3. A pass-thru has a designated (constant) output weight value.

1.1.5. Central Pattern Generator (CPG) Specification - This is a placeholder for future work.

1.2. Network

1.2.1. A network has a rectangular bounding box defined by a positive numbers of rows and and a positive number of columns.

1.2.2. A network contains a collection of zero or more elements.

1.2.3. The network contains a collection of the connections between elements.

1.2.4. The network contains a collection of input ports.

1.2.5. The network contains a collection of output ports.

1.2.6. Given any two elements A and B within the network, the network must be able to find the set of (directed) paths from A to B, where the set may be empty.

1.2.7. The network must be able to find the set of elements within the network that are reachable from a designated contained element, where element B is reachable from element A if and only if there exists at least one path from A to B.

1.2.8. The network must be able to find the set of elements within the network that are connected to an element within the network by a path.

1.2.9. A network has a character string name.

1.2.10. A network has a character string branch name.

1.2.11. A network has a creation date.

1.2.12. A network has a release date.

1.2.13. A network has a last modified date.

1.2.14. A network has a character string list of one or more the author.

1.2.15. A network has a character string group.

1.2.16. A network contains a fixed width character string for an optional copyright.

1.2.17. A network contains a fixed width character string for an optional description.

1.3. Constraints

1.3.1. The constraints define the number of rows implemented by a DANNA device family.

1.3.2. The constraints define the number of columns implemented by a DANNA device family.

1.3.3. The constraints define the a non-negative integer number of rings surrounding an element, as implemented by a DANNA device family.

1.3.4. The constraints define the locations of input ports implemented by a DANNA device family.

1.3.5. The constraints define the locations of output ports implemented by a DANNA device family.

1.3.6. The constraints define the range of the threshold parameter for all neurons implemented by the configuration.

1.3.7. The constraints define the range of the weight parameter for all synapses implemented by the configuration.

1.3.8. The constraints define the range of the delay parameter for all synapses implemented by the configuration.

1.3.9. The constraints define the range of the LTP / LTD increment parameter for all synapses implemented by the configuration.

1.3.10. Serialization

1.3.10.1. A method must be implemented to serialize constraints to a character string.

1.3.10.2. A method must be implemented to deserialize constraints from a character string.

1.4. API Functionality

1.4.1. The API is implemented by a software process running as a service on a server computer.

1.4.2. The API is accessible to other software processes, which may run on a different (client) computer having a communication or network link to the server computer.

1.4.3. Clients connect to the server with a persistent TCP connection.

1.4.4. State

1.4.4.1. An element's state is all information required to recreate the element in a software process or a member of a compatible DANNA device family.

1.4.4.2. An element's current state is retrievable.

1.4.4.3. A network's current state is retrievable.

1.4.4.4. A DANNA's current state is retrievable.

1.4.4.5. A randomly generated DANNA is retrievable.

1.4.4.6. A path between two elements in a DANNA must be retrievable.

1.4.4.7. A snapshot command must be able to retrieve the DANNA model configured on the current DANNA device.

1.4.5. Program

1.4.5.1. An element can be placed in a network at a programmable location identified by row and column coordinates relative to the origin of the network.

1.4.5.2. A network can be placed in a DANNA at a programmable location identified by row and column coordinates relative to the origin of the DANNA.

1.4.5.3. An element can be loaded into a DANNA device.

1.4.5.4. A DANNA can be loaded into a DANNA device.

1.4.5.5. A DANNA in a DANNA device can be reset.

1.4.5.6. A DANNA device can be reset.

1.4.6. Control

1.4.6.1. A run command starts the DANNA device.

1.4.6.2. A step command runs the DANNA device for a programmable number of network cycles.

1.4.6.3. A halt command stops the DANNA hardware simulation.

1.4.6.4. An input charge event must be able to be sent to all input ports in the DANNA device.

1.4.7. Output

1.4.7.1. All charge events sent from the output ports of the DANNA device are retrievable.

1.4.7.2. The state of the DANNA is retrievable. The state of the DANNA includes the state of every element. Every element's state contains a neuron's threshold or a synapse's weight, the number of fires since the last capture, and the number of stored fires (synapse only).

2. DANNA Software to Hardware Interface

2.1. Input Packets

2.1.1. Program Packets

2.1.1.1. A load packet loads an element onto the DANNA device.

2.1.1.2. A reset packet resets the DANNA device.

2.1.2. Control Packets

2.1.2.1. A run packet starts the DANNA device.

2.1.2.2. A step packet executes the DANNA device for a programmable number of network cycles.

2.1.2.3. A halt packet stops the DANNA device.

2.1.2.4. A fire packet creates an input charge event at every input port.

2.1.3. State Packets

2.1.3.1. A capture packet grabs a snapshot of the current DANNA device. A snapshot contains the contents of the accumulator (a neuron's charge or a synapse's weight), the number of fires since the last capture, and the number of stored fires for every element model in the DANNA model.

2.1.3.2. A shift packet outputs one bit from every column in a snapshot.

2.2. Output Packets

2.2.1. Version 1: Output Charge Events

2.2.1.1. A output packet contains a timestamp and output port charge event weights.

2.2.1.2. An output packet is sent when any of the output ports output a charge event.

2.2.1.3. A 64-bit nonnegative integer timestamp identifies the DANNA device cycle that the output packet was sent at.

2.2.1.4. There are 16 output port charge event weights.

2.2.2. Version 2: Output Charge Events and Monitoring

2.2.2.1. A output packet contains a timestamp, output port charge event weights, one bit from each column in a snapshot, a shift flag, a last output flag, and a DANNA configuration identifier.

2.2.2.2. A 64-bit nonnegative integer timestamp identifies the DANNA device cycle that the output packet was sent at.

2.2.2.3. There are 32 output port charge event weights.

2.2.2.4. There are 128 columns of element model snapshots. One bit of a column is outputted at a time.

2.2.2.5. The shift flag can be enabled or disabled. If enabled, the output packet contains snapshot data.

2.2.2.6. The last output flag can be enabled or disabled. If enabled, the DANNA device will no longer send output packets.

2.2.2.7. A 16-bit nonnegative integer identifier specifies the DANNA configuration for the DANNA design implemented on the DANNA device.

3. DANNA Applications

   3.1. DANNA Applications integrate with the API specified in 1.5 to retrieve the state of a DANNA model, program a DANNA device, and control a DANNA device.

4. DANNA Library

   4.1. The DANNA Library contains a collection of DANNA models.

      4.1.1. All DANNA models can be retrieved.

      4.1.2. A DANNA model can be retrieved.

      4.1.3. A DANNA model can be added.

      4.1.4. A DANNA model can be updated.

      4.1.5. A DANNA model can be deleted.

   4.2. The DANNA Library contains a collection of DANNA configurations.

      4.2.1. All DANNA configurations can be retrieved.

      4.2.2. A DANNA configuration can be retrieved.

      4.2.3. A DANNA configuration can be added.

      4.2.4. A DANNA configuration can be updated.

      4.2.5. A DANNA configuration can be deleted.

   4.3. The DANNA Library contains a collection of network models.

4.3.1. All network models can be retrieved.

4.3.2. A network model can be retrieved.

4.3.3. A network model can be added.

4.3.4. A network model can be updated.

4.3.5. A network model can be deleted.

The specification details the design of the middleware and DANNA Library, including the representation of DANNA in software, the functionality required of the middleware, the communication protocol to and from a DANNA device, and the API for the DANNA library. Of the items listed, only 1.2.6, 1.2.7, 1.2.8, and 1.4.4.5 were not implemented and have been left as future work. We are able to communicate with a DANNA device and can store networks and constraints. In the following two chapters (see Chapter 4 and Chapter 5), we layout our design and implementation for the specification.

Chapter 4

Middleware

We have developed a language-agnostic application programming interface (API) that abstracts away low-level communication details with a DANNA and provides a high-level interface for reprogramming and controlling a DANNA. This is achieved using a REST protocol (HTTP/1.1) commonly used for web and business-to-business services. An application can send HTTP requests to control, program, and monitor a DANNA device. The API has been designed in modules in order to adapt to future changes in the design of DANNA, including changes to the DANNA element design, DANNA communication protocol, and connection medium.

4.1 DANNA System

A DANNA system is composed of a host computer, a DANNA device, and a connection between the two (as seen in Figure 4.1). The host computer provides a means for interacting with a DANNA device via the connection. The connection transmits data between the host and DANNA device. The DANNA device contains a programming interface and a DANNA. The programming interface interface receives encoded packets to program and control a DANNA and sends encoded packets that contain DANNA state and charge events (discussed further in 4.8).

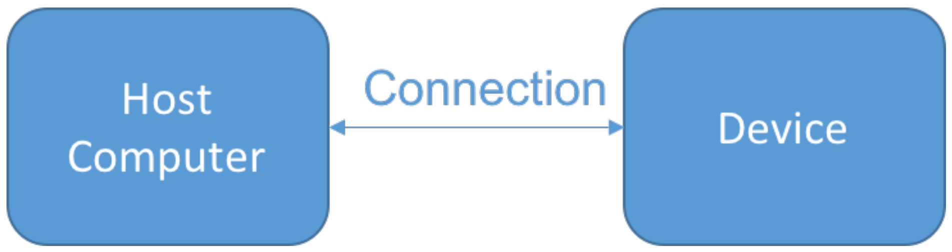


Figure 4.1: DANNA System Overview

A DANNA is a 2-D reprogrammable grid of generic elements, where each element contains parameter values and connections to its nearest neighbors. Our original system implementation comprised a Linux desktop personal computer, or PC, (the host), Peripheral Component Interconnect Express, or PCIe, interfaces on the PC and FPGA (connection), and a Field Programmable Gate Array [34], or FPGA, (device). This design is evolving, but the current stable implementation is the same as the original system.

PCIe is a complex protocol that includes many features that were not used in our system. It takes up valuable space in the FPGA real estate and requires the FPGA to be installed inside a PC. Therefore, we have since adopted USB, which is a much simpler protocol and is much more portable.

We have added two new device designs since our original implementation, a Very-Large Scale Integration (VLSI) integrated circuit design, or "chip", and a multi-threaded CPU simulation. The design for the VLSI chip, by Chris Daffron [8], is a major step forwards towards having an implemented DANNA chip. It is more space efficient than our FPGAs and will allow us to reach grid sizes that are than are possible with the FPGAs. Adam Disney has implemented a multi-threaded CPU simulation that correctly emulates the current FPGA DANNA design. This will allow us to quickly add and test changes to the DANNA design that can then be implemented in the FPGA. We are also discussing a parallel DANNA implementation in GPUs as well as a hardware implementation in memristors.

The DANNA architecture has undergone several minor changes, including modifications to DANNA elements and the programming interface. As discussed in

Chapter 2, DANNA elements have a ring connection scheme. However, the original implementation only had a single ring; elements could only connect to neighbors one-hop away. The second ring was added to allow for elements to jump other elements in order to avoid blocked paths. We have also discussed adding a new element, a Central Pattern Generator (CPG), that fires pulses in a repeated sequence once activated. The load command was split into two packets in the fall of 2014 in order to support the additional element connection ring; it was deprecated in the summer of 2015 when the input packet size was upped to 36 bytes. Research in the monitoring of DANNA state introduced shift and capture commands in the fall of 2014. Before this research, a DANNA device only produced output port event weights; now, DANNA device output consists of both output port event weights and DANNA element state.

We want to see our research in DANNA used by other researchers. Therefore, we have developed a Software Development Kit (SDK) that contains the hardware specification, software libraries, and documentation necessary for a DANNA system. The SDK includes a 3-D printed case we designed that contains an ARM Wandboard [53] (host), USB interface [47] (connection), and a FPGA [34] (device). The Wandboard executes the middlware described in this report. In the future, the DANNA visualization application developed by Drouhard [13] [14] and evolutionary optimization applications will be modified to support the SDK.

4.2 Purpose and Design

Visualization applications can examine DANNA element connections, parameters, and functionality, and evolutionary optimization applications can utilize DANNA to speedup neural network simulations. To do this, these applications must be able to interact with a DANNA device. Because interactions with a DANNA device resolves to low-level binary encoded messages, we have developed an API that hides these details. As can be seen in the previous section, the DANNA system is in a constant

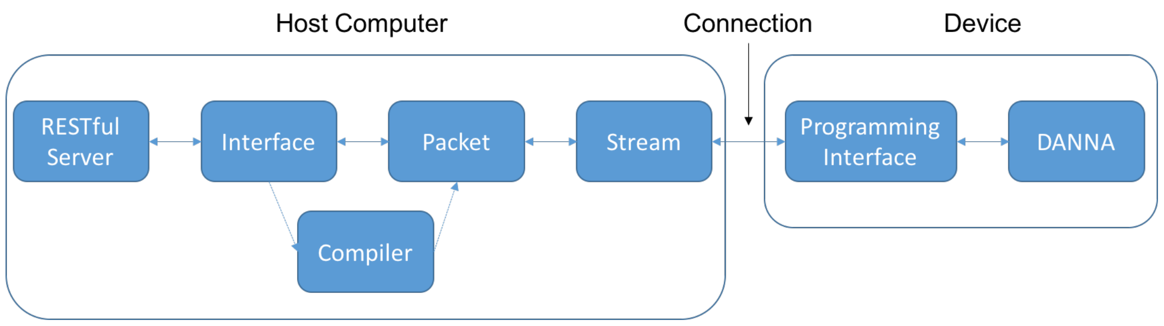


Figure 4.2: Middleware Design Overview

state of change. Therefore, we have modularized our API to accommodate future DANNA system changes.

The middleware, the API that connects applications to a DANNA device, is composed of five modules: a RESTful server, an interface, a compiler, packets, and a stream (see Figure 4.2). The RESTful server allows applications to interact with a DANNA device with HTTP requests that are not specific to any one programming language. The server's resources handle communicating with the DANNA device. A RESTful server also allows an application to be connected to the DANNA device remotely, from a client computer accross the network. The RESTful server directly maps routes, an HTTP action and URL, to interface methods. The interface specifies the versions of the compiler, packets, and stream that it is using. This functionality can be used to adjust the device, programming interface, or connection we are using at compile-time. For example, one interface can be developed for a PCIe connected FPGA, while another could be used for a USB connected ASIC. The compiler handles DANNA architecture changes by putting restrictions on networks and elements according to a device constraint file. It also contains functionality to serialize networks and elements for a programming interface. Packets are encoded packets that map directly to the device programming interface and handle programming interface modifications. Finally, streams transmit packets to and from the device programming interface over the connection. Additional connections can be added by implementing the stream interface.

4.3 Application Integration

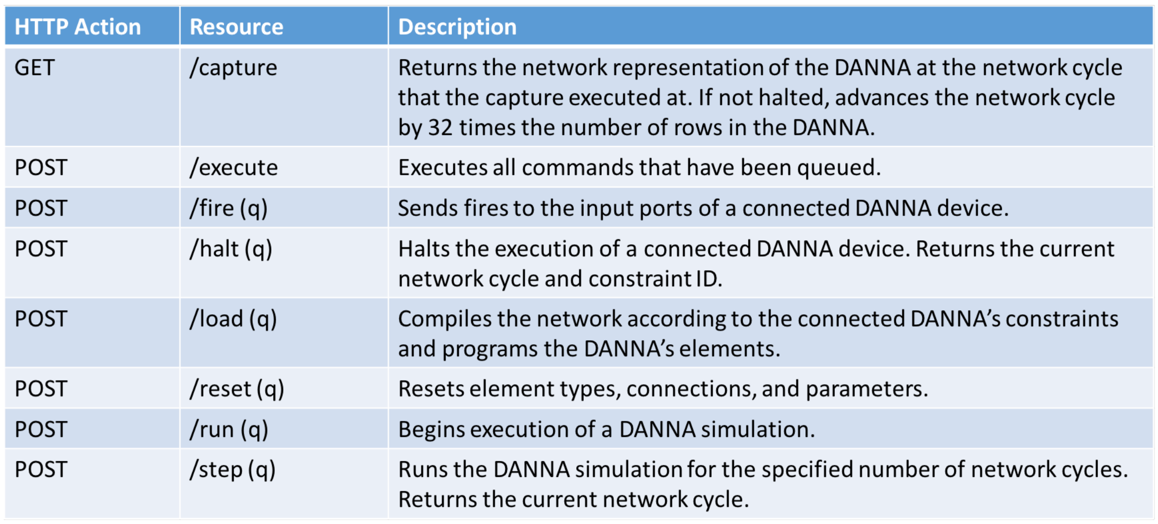
We do not expect applications of a DANNA device to be written in the same language as the software that directly interacts with the device. Therefore, we seek a language-agnostic technique to integrate applications with the middleware, which ultimately communicates with the DANNA device. We considered remote procedure calls [49] (RPCs), Simplified Wrapper and Interface Generator (SWIG) [50], and a REST API [19].

Remote procedure calls allow a client to execute a routine on a server. The client sends an object within the body of a request; the object contains the method to be called and its parameters. The formatting of the body is specific to the RPC system being used and is not standardized. Therefore, the format used needs to be documented.

SWIG is a tool that allows C and C++ code to be used in a variety of languages, primarily scripting languages. The developer defines a SWIG interface file for their C / C++ code. Then, they compile the code with SWIG for the language of their choice. They can then use the code in the selected language as if it was a library created in the selected language.

A REST API uses the RESTful interface introduced by Fielding et ȧl in [20]. Services are available through the use of URLs, the create, read, update, and delete (CRUD) methods of HTTP, and the headers of requests. If designed correctly REST APIs are self-documenting; a user can guess the routes available in the API.

SWIG is useful for integrating a library into a program residing on the same machine; however, we also need to interact with the middleware from a separate computer. Because a REST API is self-documenting and provides the functionality needed for application users, we use it to integrate application software with the middleware. Figure 4.3 lists the routes available in the REST API. Routes with a (q) denote routes that are pushed onto a queue, which can be flushed and sent to the connected device with the execute command. These commands are queued so that



HTTP Action	Resource	Description
GET	/capture	Returns the network representation of the DANNA at the network cycle that the capture executed at. If not halted, advances the network cycle by 32 times the number of rows in the DANNA.
POST	/execute	Executes all commands that have been queued.
POST	/fire (q)	Sends fires to the input ports of a connected DANNA device.
POST	/halt (q)	Halts the execution of a connected DANNA device. Returns the current network cycle and constraint ID.
POST	/load (q)	Compiles the network according to the connected DANNA's constraints and programs the DANNA's elements.
POST	/reset (q)	Resets element types, connections, and parameters.
POST	/run (q)	Begins execution of a DANNA simulation.
POST	/step (q)	Runs the DANNA simulation for the specified number of network cycles. Returns the current network cycle.

Figure 4.3: The available REST routes available in the middleware. Notice: Any routes with a (q) denote functions that are not immediately sent to the connected device, but queued instead.

a client can send commands on a network cycle basis. Without the queue, network latency could affect commands executing at a specific network cycle.

4.3.1 Persistent Connections

It is important to note that, as of HTTP/1.1, an HTTP request to a server automatically sets up a persistent TCP connection to that server [19]. A persistent connection is desirable because it alleviates the overhead of setting up and tearing down a TCP connection every time an HTTP request is made. The persistent connection can be terminated through headers residing in the HTTP packets. But, we are seeking maximum efficiency and destroy a persistent TCP connection only when the middleware exits.

4.4 Interface

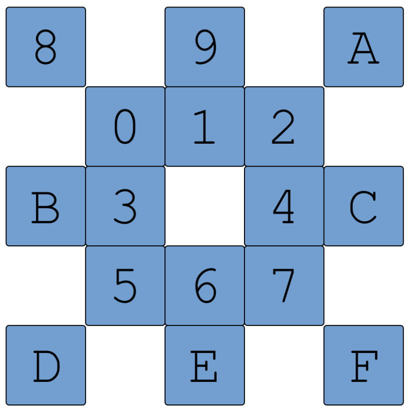
The interface correlates to a DANNA system and contains the modules that map to a DANNA system. It includes a stream (connection), packets (programming interface), and constraints and compiler (DANNA). It has functionality for capturing the state of a DANNA, firing input ports, loading a network, halting a simulation, resetting a DANNA, running a simulation, steping a simulation. It can be implemented to provide specific functionality for a device by allowing for compile-time adjustments to the compiler, stream, and packets used.

4.5 Networks and Elements

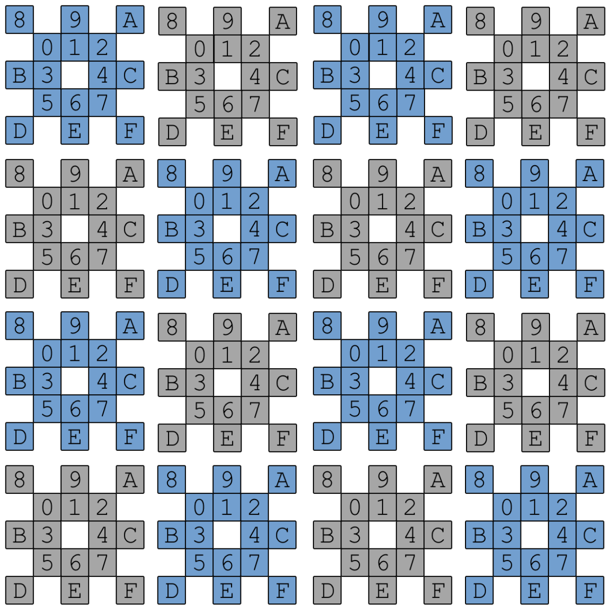
A network is a grid space of elements meant to emulate a DANNA grid. It can be viewed as a bidirectional graph, where network elements represent graph nodes and network element inputs and outputs represent graph edges.

In addition to connections to other elements, elements have a kind and properties. The element kind is a string that can be dynamically changed and is validated by the compiler according to the constraints specified by the DANNA device. For example, current DANNA device constraints limit the element kind to a neuron, synapse, or pass-thru. Properties are stored as key-value pairs, are specific to the element kind, and are verified by the compiler according to the constraints of the DANNA device. For instance, a neuron has a threshold property with a maximum value of 127, a minimum value of -128, and a default value of 0. By storing the element kind as a string and the properties of an element as key-value pairs, we can quickly and dynamically change what are elements look like and adapt to future element changes in DANNA.

In the element, edges are maintained in a single orientation scheme (as seen in Figure 4.4a. The single orientation scheme allows for the placement of an element anywhere within a network (see Figure 4.4b), unlike the DANNA orientation scheme, where it is confined to other positions in the grid space with the same orientation unless its enabled inputs and outputs are translated. Notice that because the single orientation scheme is not the same as the DANNA orientation scheme, it has to be translated at compile time; however, this is seen as more efficient than changing the ports at run-time when an element is moved around the grid.



(a) Single orientation element scheme



(b) Elements can be placed anywhere in the grid without enabled input and output translation, when the single orientation element scheme is applied to the grid.

Figure 4.4: Figure (a) shows the orientation scheme for a single element. Figure (b) shows that inputs and outputs do not have to be translated in a grid where all elements have the single orientation scheme.

Network methods have been included to modify and retrieve information from the graph of elements, including methods to retrieve elements, connect and disconnect elements, retrieve paths, add a subnetwork to a network, "cut" a subnetwork from a network, resize a network, and set and retrieve metadata. Elements are stored as pointers and are returned as such; therefore, a user can directly manipulate them inside the network. The connection methods are merely for convenience; elements have methods for enabling and disabling inputs and outputs. By examining element connections, a shortest hop path between a source and sink element can be retrieved using breadth-first search. Similarly, all paths between a source and sink element can be retrieved using depth-first search. The add subnetwork method places a subnetwork within a network at a starting coordinate; it ensures that every element in the subnetwork will not overwrite any elements in the network. When a subnetwork is added, the network updates each subnetwork elements parent network to point to the subnetwork; this allows us to retrieve the network an element belongs to by looking at the element's parent. The cut network acts like an Excel cut; it spans a rectangular grid, removes all element values, stores them in a new network, nullifies the old element values, and returns the new network. The resize method expands and contracts the grid of the network. If the grid is contracted, any elements not within the new network bounds are destructed. Finally, network metadata includes a network ID, name, description, author, and group.

4.5.1 Network Generation

The ability to create random neural network models is beneficial for testing and, in the future, evolutionary optimization. Thus, we created a generation module that builds DANNA models with random placement of input and output ports, networks, neurons, synapses, element parameters, and connections and path. Currently, the generation module is not compatible with our most up-to-date modules, the network, element, constraint, and compiler modules.

We begin with a parameter to specify the number of ports, N_p and a parameter to specify the number of elements, N_e. We randomly select N_p unique neurons along the left-most column of the grid, specify them as input ports, and allocate them. Similarly, we allocate N_p unique synapses as output ports along the right-most column of the grid. Then, we select N_e unique elements, excluding elements that were allocated as ports, and randomly allocate their element types.

We then iterate through all of allocated elements and find a path, a set of positions in the grid, using depth-first search (DFS) from the current element to a randomly selected element in the grid; the randomly selected element cannot be the current element. When we look for a path, we only use elements that still have input and output ports available; we do not reuse paths that already exists. This could be seen as a future improvement.

Once we have found a path, we allocate elements along it. For every position along the path, we determine what element to allocate based on the last element we looked at. If the previous element was a neuron, then we randomly choose to either create a pass-thru or a synapse. If the previous element was a synapse, the we randomly choose to either create a neuron or synapse. If the previous element was a pass-thru, the randomly choose to either create a synapse or pass-thru. We allocate elements in this grammatical fashion to ensure a logical chain of elements.

4.6 Constraints

The constraints module represents the restrictions of the connected DANNA device. For example, it contains the size of the network, the number of connection rings surrounding an element, the element kinds and their properties, and the external port locations. A more detailed description can be found in Chapter 3.

The constraints module is serialzed and deserialized using JSON. The intent of the JSON constraints representation is for DANNA device designers to specify the constraints of their design. From there it is readable via the middleware constraints module and can be stored in the DANNA Library discussed in Chapter 5.

4.7 Compiler

The compiler validates and serializes a network and its contained elements. Before serialization, it performs syntactic and semantic checks for verfication and places the network within the grid constraints of the DANNA device.

In the syntactic stage, the compiler ensures that the network can fit within the DANNA device grid, that it has edges that meet the maximum number of element rings for the device, and that elements with non-null kinds have values for their required properties. If an element does not have its element kind properties, a warning is issued and the default values are applied to the properties.

Then, in the placement stage, a network model is created that is the size of the DANNA on the connected DANNA device. Using the network model's subnetworking functionality, the network passed in to be compiled can be placed within the device network as a subnetwork. The compiler places the subnetwork starting at coordinate (0, 0) of the device network. In the future, we want to place networks within the device network to optimize space consumption, route availability, and device port availability.

Finally, in the semantic stage, the compiler confirms that element edges are valid, that edges are mapped to the DANNA device edge scheme, that element property values are within ranges specified in the constraints, and that device ports are enabled. An element's edges are considered valid if its element kind number of inputs and number of outputs is less than or equal to the maximum allowed in the constraints. Network edges are mapped to the DANNA device edge scheme using a port map, which translates from the orientation free edge representation to the device edge orientation scheme. Property values are checked to be between the exclusive minimum and maximum values according to the constraints.

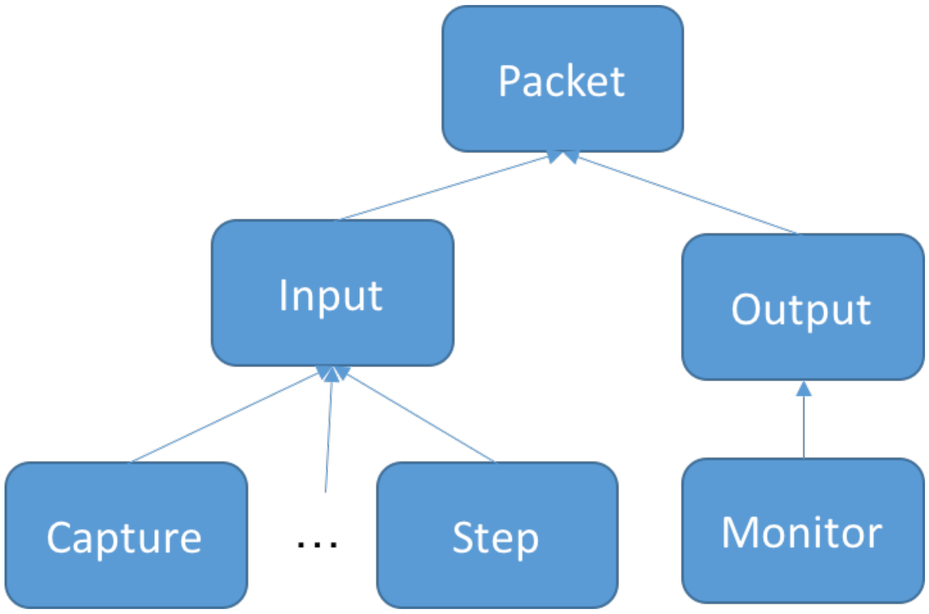


Figure 4.5: Inheritance diagram for Packet

The compiler directly depends on the data residing in the constraints module. Because constraints can be represented in a JSON file they can be modified to change the verifications of the compiler and, in the case of elements, change the functionality of the compiler.

4.8 Packets

Packets provide a base functionality for storing data being tranferred to and from a DANNA device. Data are stored in a signed 8 bit integers array. Packets can be extended through inheritence; the current inheritance structure can be seen in Figure 4.5.

4.8.1 Output Packets

Output packets extend the base functionality of packets by sizing the array to 36 bytes and filling their data appropriately for each programming interface command. Every type of output packet is broken into two pieces: the opcode and the payload.

Table 4.1: Output packet types, their opcodes, payload parameters, and device actions.

Type	**Opcode**	**Payload**	**Device actions**
Capture	0100 0000, 0x40	Empty	Snapshots the state of every element
Fire	0001 0000, 0x10	32 one byte signed weights	Sends fire events with payload weights to external input ports
Halt	0000 0010, 0x0	Empty	Stops the execution of the DANNA and sends an output packet with EOF set.
Load element	0000 0001, 0x1	Element address, type, parameters, enabled inputs and outputs	Programs the appropriate DANNA element
Reset	0010 0000, 0x20	Empty	Clears type, parameters, and enabled inputs and outputs. Notice, does not clear synapse fire queue.
Run	0000 0100, 0x4	Empty	Begins execution of the DANNA
Shift	1000 0000, 0x80	Empty	Outputs a packet with one bit from every DANNA column
Step	0000 1000, 0x8	8 byte unsigned network cycle	Executes the DANNA for the payload specified cycle count

The opcode is a one byte quantity that follows a one-hop encoding and identifies the operation that a DANNA device client is requesting. The payload includes parameters specific to an operation. Table 4.1 lists the current output packet types. Output packets that have parameters in their payload have an initialize method that fills in the payload for a user.

4.8.2 Input Packets

Input packets expand the operation of a packet by sizing the array to 64 bytes and filling in the data appropriately with data received from a DANNA device. In the original implementation, packets received from a device only contained a timestamped set of events (weights from external output ports). However, our newest programming interface contains these events, the captured DANNA state, a shift flag, and an end-of-file (EOF) flag.

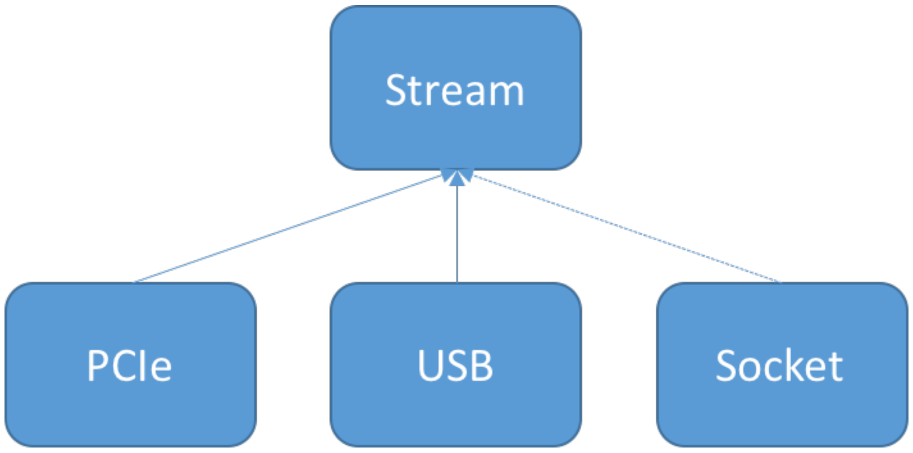


Figure 4.6: Inheritance diagram for Stream. Notice: The socket stream has not been implemented and has been left for future work.

4.8.3 Validation of Packets

Every input command has been sent to the FPGA programming interface. All commands are accepted by the programming interface and perform the appropriate action. Similarly, all data received from the programming interface are correct. Debugging took several days of joint effort between the software developer and hardware designer. It is, compared to the hardware, quicker to debug the software; debugging the hardware is slow because it requires waiting fifteen minutes to a hour to implement a new design on the FPGA for a small DANNA.

4.9 Streams

A stream base class provides an interface with a read and write method for packets. The stream base is then implemented for each new connection type. We have implemented streams for both PCIe and USB device connections (see Figure 4.6).

DANNA devices accepts input at any time and sends output in an asynchronous fashion. Therefore, our stream implementations (PCIe and USB) are asynchronous. To do this, we spawned read and write threads. Each thread has a packet buffer and a lock. When writing, we call the stream write method. The write method acquires the write lock and pushes a packet onto the write buffer. Then, at some later point the write thread acquires the write lock, pulls a packet from the write buffer, and writes to the appropriate destination. The read thread reads from the destination, acquires the read lock, pushes a packet onto the buffer, and then reads again. When the stream read method is called, it acquires the read lock and pops a packet from the buffer.

4.10 Adapting to Future Changes

The middleware has been designed to be flexible to future changes in various parts of the DANNA system. The following sections describe how to add additional functionality to the middleware.

4.10.1 Adding or Modifiying an Input/Output Packet

An input/output packet can be added by extending the input/output packet interface. An existing input/output packet can be modified by extending the current packet and modifiying its factory method to return the newest version.

4.10.2 Adding a new Element

A new element can be added by including its information in a contraint file. Once added to the contraint file, a compiler that reads the constraint file will be able to accept and validate the new element. Notice that while the compiler can validate it, it cannot serialize it at run-time because load packets cannot be adjusted at run-time.

4.10.3 Serializing a new Element

Serializing a new element type requires that the load packet be adjusted. If the old load functionality is to persist, the the old load packet can be extended. If not, a new load packet can be created by extending the input packet interface. In either case, the load packet factory method should be updated to return the newest version of the load packet. The compiler will then use this new packet version when it calls the factory method.

4.10.4 Adding a new Element Parameter

Element parameters can be added at run-time by adding them to a constraint file that is read by a compiler. Simply add the additional parameter and specify a minimum, maximum, and default value.

4.10.5 Connecting to a new Device

A new connection type can be added by extending the Stream interface for that connection type. For example, a socket stream could be added by deriving from the stream interface and implementing the read and write methods of the stream.

4.11 Documentation and Testing

Documentation was generated from code comments using Doxygen [12]. Design decisions and installation are in separate markdown pages.

We used the Catch unit testing framework [6] to ensure that the specification laid out in Chapter 3 is fulfilled. Elements and networks have been minimally tested (element creation, element methods, network creation, and network methods); a comprehensive test of the implementation against the specification (see Chapter 3) is left for future work.

We have begun to test that the multi-threaded simulator emulates the FPGA design and have had successful tests for several hand-designed 10x10 networks while using either PCIe or USB interface to the DANNA device.

4.12 Summary

We have developed a language-agnostic API that abstracts away low-level communication details with a DANNA and provides a high-level interface for reprogramming and controlling a DANNA. An application can send HTTP requests to control, program, and monitor a DANNA device. The API has also been designed in modules in order to adapt to future changes in the design of DANNA, including changes to the DANNA element design, DANNA communication protocol, and connection medium.

Chapter 5

DANNA Library

It is beneficial for applications and researchers to store and view networks with known functionality. It is also advantageous to be be able to load device constraints dynamically; so, constraints will also be stored. Therefore, a Representational State Transfer (REST) API with a MongoDB database back-end has been developed to encourage the collection and exploration of networks and device constraints.

5.1 MongoDB and Git

When considering how to store networks, we looked at using MongoDB [27], a NoSQL database, and Git [21], a versioning system. MongoDB is a NoSQL database that stores Binary JSON (BSON), provides standard database functionality, and is deployable in a distributed environment using sharding. Data are stored in a document (analogous to a SQL row/tuple) inside a collection (analogous to a SQL table/relation). Git is a versioning system that would allow us to store versions of networks very easily. In its simplest form, Git is essentialy a standard folder with a hidden Git folder that contains version information inside of it. Therefore, it is easy for a human to clone a Git repository and view networks.

In a small database setup, one computer would be used to host a database. If the computer begins to run low on disk space or the CPU is consistenly maxed out because of a high number of queries, then further resources are added to the computer. In constrast, NoSQL has adopted a distributed philosophy. A database is broken into pieces and spread accross a cluster of computers. MongoDB refers to the distribution of its database as sharding [28].

While Git is usable by a human, it is more difficult for a computer than a database. For example, MongoDB has searching functionality that is not available with Git. Git does not scale as well as MongoDB because it does not have distributed storage. Because MongoDB scales well and naturally stores JSON, we will be using it to store our collection of networks and constraints.

5.2 Tools

Node.js [35] is a JavaScript server framework running on the Chromimum V8 JavaScript engine. It provides the tools to quickly set up a lightweight server. It also contains a rich module system and package manager to further facilitate swift and clean code development. Express.js [17] is a Node.js module that provides a router that handles routes, which are a combination of a HTTP request action and the requested resource. Mongoose.js [30] is a Node.js module that connects and interacts with a MongoDB. Object-relational mapping (ORM) [36] binds objects in Node.js to MongoDB collections and is used to create and modify documents in the collections.

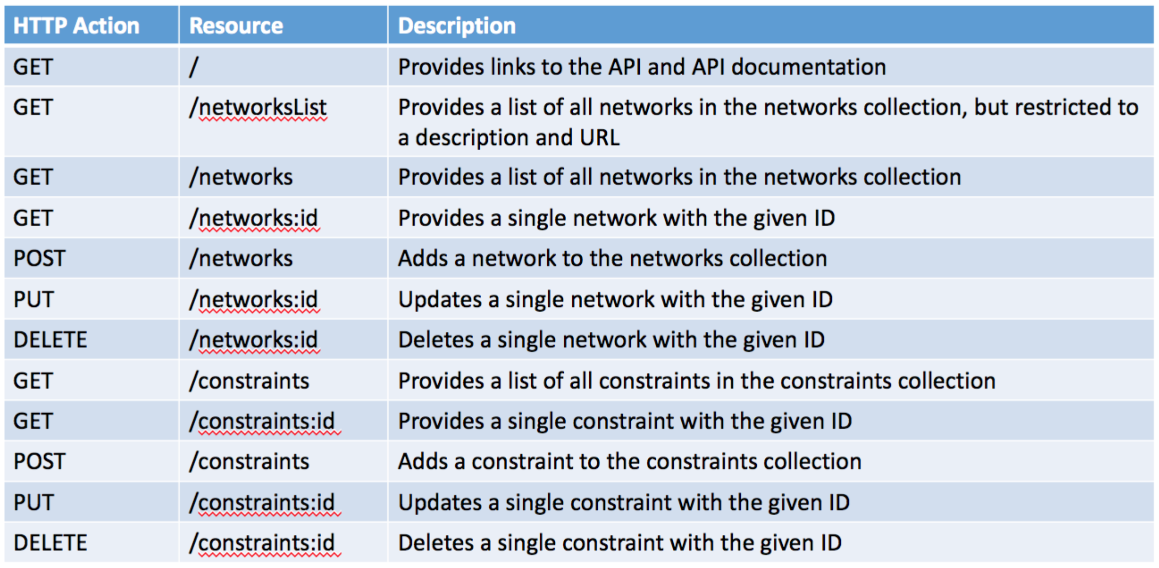
5.3 RESTful API

Using Express, we create a router, an object that registers routes. An Express route is defined by the HTTP action used, the URL, and a callback function. When the client requests a route, the router looks through its action types for the URL and executes the appropriate callback. The callback performs an operation with the database and then returns a response.

Mongoose uses object-relational mapping (ORM) to create collections (tables in SQL terminology) of documents (tuples or rows in SQL terminology). A Mongoose schema is created that defines the structure of the documents in a collection. After the schema has been defined it can be instantiated to create a connection to the database. The instantiated object then has built-in database functionality for adding, creating, updating, and deleting documents of the instantiated type. Therefore, you can use the instantiated schema object to interact with the database in callback functions.

5.4 Routes and Filters

In RESTful API terminology, a route is a path from the server URL and a HTTP request action. When a request is made for a route, a registered callback function is executed in the server and a response is generated. For example, a server might return a welcome message or home page with the route, "GET /". The list of available routes in our library can be seen in Figure 5.1. All response bodies are JSON representations of the requested resource.



HTTP Action	Resource	Description
GET	/	Provides links to the API and API documentation
GET	/networksList	Provides a list of all networks in the networks collection, but restricted to a description and URL
GET	/networks	Provides a list of all networks in the networks collection
GET	/networks:id	Provides a single network with the given ID
POST	/networks	Adds a network to the networks collection
PUT	/networks:id	Updates a single network with the given ID
DELETE	/networks:id	Deletes a single network with the given ID
GET	/constraints	Provides a list of all constraints in the constraints collection
GET	/constraints:id	Provides a single constraint with the given ID
POST	/constraints	Adds a constraint to the constraints collection
PUT	/constraints:id	Updates a single constraint with the given ID
DELETE	/constraints:id	Deletes a single constraint with the given ID

Figure 5.1: The routes available in the DANNA Library

5.5 Summary

We have implemented a collection of networks and constraints with a Mongo database. Users can interact with the database through the RESTful API developed in Node.js.

Chapter 6

Summary

We have developed a middleware for applications of Dynamic Adaptive Neural Network Array systems and a library service for storage of DANNA structures. We presented a language-agnostic API designed to provide high-level functionality for controlling, programming, and monitoring a DANNA. The modular design of the middleware will accommodate DANNA system and DANNA architecture changes. We are able to successfully communicate with a DANNA device via HTTP requests. Future middleware work includes keeping a element change log, generating random networks with a percentage utilization measurement, automatically placing networks within other networks using simulated annealing, and communicating with a DANNA device using a socket stream.

We introduced a library for storage of DANNA structures that could be used by future researches for exploration of tasks using DANNA. The Node.js server was quick to develop and is maintainable. Object-relational mapping allowed for the rapid development of a database that stored binary JSON documents (BSON). Networks and constraints can be retrieved, added, updated, and deleted from the database. Further work on the library includes compressing responses in the server before they are sent, adding filtering, adding searching, adding a more advanced authorization method, dividing long responses into pages, implementing sharding, and developing a website for quick access to the database.

Bibliography

[1] J. Douglas Birdwell, Mark E. Dean, and Catherine Schuman. Method and apparatus for constructing a dynamic adaptive neural network array (DANNA). U.S. Patent Application 14/513,297, filed October 14, 2014. 1, 4

[2] J. Douglas Birdwell, Mark E. Dean, and Catherine Schuman. Method and apparatus for providing random selection and long-term potentiation and depression in an artificial network. U.S. Patent Application 14/513,334, filed October 14, 2014. 3

[3] J. Douglas Birdwell and Catherine Schuman. Method and apparatus for constructing a neuroscience-inspired artificial neural network. U.S. Patent Application 14/513,280, filed October 14, 2014. 3

[4] Brian. Brian. http://briansimulator.org/, 2015. Accessed: 25-July-2015. 9

[5] Daniel Brüderle, Mihai A Petrovici, Bernhard Vogginger, Matthias Ehrlich, Thomas Pfeil, Sebastian Millner, Andreas Grübl, Karsten Wendt, Eric Müller, Marc-Olivier Schwartz, et al. A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems. *Biological cybernetics*, 104(4-5):263–296, 2011. 9

[6] Catch. Catch. https://github.com/philsquared/Catch, 2015. Accessed: July-26-2015. 14, 42

[7] World Wide Web Consortium et al. Extensible markup language (xml) 1.1. 2006. 7

[8] Christopher Paul Daffron. Visualization techniques for neuroscience-inspired dynamic architectures. Master's thesis, University of Tennessee at Knoxville, 2015. 28

[9] Andrew Davison, Eilif Muller, Daniel Brüderle, and Jens Kremkow. A common language for neuronal networks in software and hardware. *Neuromorph. Eng*, 2010. 9

[10] Andrew P Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. Pynn: a common interface for neuronal network simulators. *Frontiers in neuroinformatics*, 2, 2008. 8

[11] Mark E Dean, Catherine D Schuman, and J Douglas Birdwell. Dynamic adaptive neural network array. In *Unconventional Computation and Natural Computation*, pages 129–141. Springer, 2014. 4

[12] Doxygen. Doxygen. `http://www.stack.nl/~dimitri/doxygen/`, 2015. Accessed: 23-May-2015. 14, 42

[13] Margaret Drouhard, Catherine D Schuman, J Douglas Birdwell, and Mark E Dean. Visual analytics for neuroscience-inspired dynamic architectures. In *Foundations of Computational Intelligence (FOCI), 2014 IEEE Symposium on*, pages 106–113. IEEE, 2014. 29

[14] Margaret Grace Drouhard. Visualization techniques for neuroscience-inspired dynamic architectures. Master's thesis, University of Tennessee at Knoxville, 2015. 29

[15] Eclipse. Eclipse. 11

[16] Eclipse. Eclipse. `http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.cdt.doc.user%2Fconcepts%2Fcdt_c_over_cdt.htm&cp=5_2_0`, 2015. Accessed: 28-May-2015. 11

[17] Express.js. Express.js. `http://expressjs.com/`, 2015. Accessed: 25-July-2015. 16, 45

[18] G. Fairhurst. Advice to link designers on link automatic repeat request (arq), 2002. 9

[19] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1, 1999. 31, 32

[20] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002. 1, 15, 31

[21] Git-Scm. Git. http://git-scm.com/, 2015. Accessed: 25-July-2015. 2, 44

[22] Google. Google Trends xml api vs json api. http://www.google.com/trends/explore?q=xml+api#q=xml%20api%2C%20json%20api&cmpt=q, 2015. Accessed: 28-May-2015. 12

[23] Atif Hashmi, Andrew Nere, James Jamal Thomas, and Mikko Lipasti. A case for neuromorphic isas. *ACM SIGPLAN Notices*, 47(4):145–158, 2012. 6

[24] Xin Jin, Francesco Galluppi, Cameron Patterson, Alexander Rast, Sergio Davies, Steve Temple, and Steve Furber. Algorithm and software for simulation of spiking neural networks on the multi-chip spinnaker system. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8. IEEE, 2010. 10

[25] Misha Mahowald. *VLSI analogs of neuronal visual processing: a synthesis of form and function*. PhD thesis, California Institute of Technology, 1992. 7

[26] Peter Miller. Recursive make considered harmful. *AUUGN Journal of AUUG Inc*, 19(1):14–25, 1998. 10

[27] MongoDB. MongoDB. https://www.mongodb.org/, 2015. Accessed: 26-July-2015. 2, 16, 44

[28] MongoDB. MongoDB sharding introduction. http://docs.mongodb.org/manual/core/sharding-introduction/, 2015. Accessed: 26-July-2015. 17, 45

[29] Mongoose-Cpp. Mongoose-Cpp. https://github.com/Gregwar/mongoose-cpp, 2015. Accessed: 26-July-2015. 15

[30] Mongoose.js. Mongoose.js. http://mongoosejs.com/, 2015. Accessed: 26-July-2015. 16, 45

[31] NEST. NEST. http://www.nest-initiative.org/?page=Software, 2015. Accessed: 25-July-2015. 9

[32] NetBeans. NetBeans c++ setup instructions. https://netbeans.org/community/releases/80/cpp-setup-instructions.html, 2015. Accessed: 28-May-2015. 11

[33] Neuron. Neuron. http://www.neuron.yale.edu/neuron/, 2015. Accessed: 25-July-2015. 9

[34] Bernard J New. Field programmable gate array with distributed gate-array functionality, February 23 1999. US Patent 5,874,834. 28, 29

[35] Node.js. Node.js. https://nodejs.org/, 2015. Accessed: 25-July-2015. 15, 45

[36] ORM. ORM. http://hibernate.org/orm/what-is-an-orm/, 2015. Accessed: 25-July-2015. 45

[37] PCSIM. PCSIM. http://sourceforge.net/projects/pcsim/, 2015. Accessed: 25-July-2015. 9

[38] Premake. Premake 4. https://github.com/premake/premake-4.x/wiki, 2015. Accessed: 23-May-2015. 11

[39] PyNCS. PyNCS backends. http://inincs.github.io/pyNCS/general/introduction.html, 2015. Accessed: 23-May-2015. 9

[40] PyNN. PyNN backends. http://neuralensemble.org/docs/PyNN/backends.html, 2015. Accessed: 23-May-2015. 9

[41] Python. Python. https://www.python.org/, 2015. Accessed: 25-July-2015. 8

[42] Catherine D Schuman and J Douglas Birdwell. Dynamic artificial neural networks with affective systems. *PloS one*, 8(11):e80455, 2013. 1, 3

[43] Catherine D Schuman and J Douglas Birdwell. Variable structure dynamic artificial neural networks. *Biologically Inspired Cognitive Architectures*, 6:126–130, 2013. 3

[44] Catherine D. Schuman, J. Douglas Birdwell, and Mark E. Dean. Neuroscience-inspired inspired dynamic architectures. In *Biomedical Science and Engineering Center Conference (BSEC), 2014 Annual Oak Ridge National Laboratory*, pages 1–4, May 2014. 3

[45] Catherine D Schuman, J Douglas Birdwell, and Mark E Dean. Spatiotemporal classification using neuroscience-inspired dynamic architectures. *Procedia Computer Science*, 41:89–97, 2014. 3

[46] Catherine Dorothy Schuman. *Neuroscience-Inspired Dynamic Architectures*. PhD thesis, University of Tennessee at Knoxville, 5 2015. 3

[47] Universal Serial Bus Specification. Apr. 27, 2000. *XP002474828*, pages 239–274. 29

[48] Fabio Stefanini, Emre O Neftci, Sadique Sheik, and Giacomo Indiveri. Pyncs: a microkernel for high-level definition and configuration of neuromorphic electronic systems. *Frontiers in neuroinformatics*, 8, 2014. 8

[49] Inc. Sun Microsystems. Rpc: Remote procedure call protocol specification, 1988. 1, 31

[50] Swig. Swig. http://www.swig.org/, 2015. Accessed: 23-May-2015. 1, 31

[51] Linus Torvalds. Xml. https://plus.google.com/+LinusTorvalds/posts/X2XVf9Q7MfV, 2014. Accessed: 23-May-2015. 12

[52] Tup. Tup. http://gittup.org/tup/, 2015. Accessed: 23-May-2015. 10

[53] Wandboard. Wandboard. http://www.wandboard.org/, 2015. Accessed: 26-July-2015. 29

Vita

Joshua Caleb Willis was born in Nashville, Tennessee, to Jerry and Patricia Willis. He has two half-siblings and seven nieces and nephews. Upon graduation from White House High School, he enrolled at the University of Tennessee at Knoxville and soon began studying computer science. He received his Bachelor's of Science in Computer Science in May of 2014 from UTK and accepted a teaching assistantship with Dr. J. Douglas Birdwell for the following 2014-2015 academic year. Willis graduated with his Master's Degree in Computer Science in August of 2015.