

University of Nevada, Reno

# Brain Communication Server: A Dynamic Data Transferal System for A Parallel Brain Simulator

A thesis submitted in partial fulfillment of the  
requirements for the degree of Master of Science  
with a major in Computer Science

by

James G. King

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

May, 2005

UNIVERSITY  
OF NEVADA  
RENO

THE GRADUATE SCHOOL

We recommend that the thesis  
prepared under our supervision by

**James G. King**

entitled

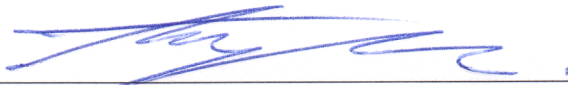
**Brain Communication Server: A Dynamic Data  
Transferral System For A Parallel Brain Simulator**

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

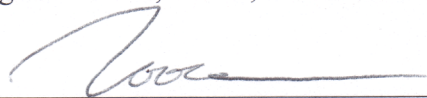
**MASTER OF SCIENCE**



Frederick C. Harris, Jr. Ph. D., Advisor



Angkul Kongmunvattana, Ph. D., Committee Member



Philip Goodman, M.D., Graduate School Representative



Marsha H. Read, Ph. D., Associate Dean, Graduate School

## Abstract

Among the many ongoing research projects at the University of Nevada, Reno, one has been the development of a biologically realistic brain simulator called the NeoCortical Simulator (NCS). As NCS has evolved, it has become desirable to have other applications interact with the simulated brain in order to perform a greater range of experiments. The Brain Communication Server (BCS) allows separate applications to exchange data with NCS. This creates an interactive system where information goes into NCS as stimulus to affect the network of simulated neurons and a response returns from NCS to indicate the reactions generated within the simulation. By using BCS as a bridge to NCS, a variety of applications may be developed to work with the simulator to achieve greater experimentation results and learning.

## Dedication

To my parents: James and Carmen. You have always been there for me and I respect all that you have done.

## Acknowledgements

I would to thank Dr. Frederick C. Harris, Jr. for showing more patience and insight than I could ever have imagined. He is a model professor who could never receive all the adulation that he is due.

I would like to thank Dr. Goodman for providing a wonderful opportunity at the Brain Lab. He shows that technology holds the key to understanding ourselves and the world around us.

Also, thanks to Christine Wilson, Rich Drewes, Rene Doursat, and James Frye for exposing me to the depths of their experience. They are truly a well of knowledge that has enhanced my understanding of not only Computer Science, but critical thinking as well.

Finally, I would like to thank Dr. Frederick C. Harris, Jr., Dr. Angkul Kongmunvattana, and Dr. Philip Goodman for taking the time to serve on my committee.

# Contents

<b>Abstract</b> .....	<b>i</b>
<b>Dedication</b> .....	<b>ii</b>
<b>Acknowledgements</b> .....	<b>iii</b>
<b>List of Figures</b> .....	<b>vi</b>
<b>List of Tables</b> .....	<b>vii</b>
<b>Chapter 1: Introduction</b> .....	<b>1</b>
<b>Chapter 2: Background</b> .....	<b>3</b>
2.1 NCS .....	3
2.1.1 The Biology of NCS .....	3
2.1.2 NCS1 and NCS2.....	7
2.1.3 C++ Version NCS3.....	9
2.1.4 NCS4 and NCS5.....	11
2.2 Applications.....	15
2.2.1 CARL.....	15
2.2.2 IVO .....	18
2.3 Analysis of other Server Packages.....	19
2.3.1 Overview of limitations .....	19
2.3.2 Apache .....	20
2.3.3 Squid .....	20
<b>Chapter 3: Development</b> .....	<b>22</b>
3.1 First Approach: VOservers .....	22
3.1.2 Using VOservers .....	24
3.1.3 Data Format.....	26
3.1.4 VOservers commands .....	27
3.1.5 Difficulties with VOservers.....	28
3.2 Second Approach: Autoport Server .....	30
3.2.1 Behavior .....	30
3.2.2 Identification Header .....	31
3.2.3 Updated and New features .....	33
3.3 Third Approach: Single Socket Server.....	35
3.3.1 Description .....	36
3.3.2 Modified Handling of Stimulus and Reports.....	36
3.3.3 New Reporting using NFS.....	37

<b>Chapter 4: Results .....</b>	<b>42</b>
4.1 Server Metrics.....	42
4.2 Interaction of NCS and IVO .....	44
4.2.1 Expansion of NCS Input File .....	44
4.2.2 Runtime Commands for NCS .....	46
4.2.3 Demonstration of IVO .....	47
<b>Chapter 5: Conclusions and Future Work .....</b>	<b>51</b>
5.1 Contribution.....	51
5.2 Potential Applications .....	52
5.2.1 Visualization.....	52
5.2.2 NCS Tutorial Program .....	53
<b>References .....</b>	<b>55</b>

# List of Figures

Figure 2.1 Compartment reaches threshold and fires an Action Potential (AP) .....	4
Figure 2.2 APs arrive on compartments, and new APs will be generated.....	5
Figure 2.3 Hebbian Learning occurs over 3 stages.....	7
Figure 2.4 Speedup results of attempting to parallelize NCS2 [15].....	8
Figure 2.5 NCS3 communication model centered on a Message Bus object [36] .....	10
Figure 2.6 Simulated response to step current of 150-300 pA [27].....	12
Figure 2.7 With optimizations, NCS achieves speedup closer to ideal [11].....	14
Figure 2.8 Preprocessing of visual and audio data [25].....	17
Figure 3.1 Data enters the server from the writer port, leaves through the reader port.....	23
Figure 3.2 Simple connections scheme for data exchange .....	23
Figure 3.3 NCS nodes are kept separate for security.....	24
Figure 3.4 Input data may convert from ASCII to binary or vice versa. ....	25
Figure 3.5 Example header used when sending ASCII data.....	27
Figure 3.6 Example header used when sending binary data. ....	27
Figure 3.7 With many ports, coordination becomes difficult .....	29
Figure 3.8 Bad configurations were difficult to detect and debug. ....	30
Figure 3.9 Autoport Server manages connections for the user. ....	31
Figure 3.10 Single Socket Server handles larger simulations best.....	38
Figure 3.11 Extract data from multiple reports and converge the data based on time step. ....	39
Figure 4.1 Timing differences of a small simulation.....	43
Figure 4.2 The IVO needed to move from the center towards the right.....	48
Figure 4.3 Initial IVO experiments worked with interpreting reports.....	49
Figure 4.4 Without an explore function, the IVO might not move .....	50
Figure 4.5 As the simulations was fine-tuned, the IVO behaved as desired.....	50
Figure 5.1 Screenshot from early NCS visualization software .....	52
Figure 5.2 A web based interface to create NCS input files .....	53
Figure 5.3 Instinct: a java-based tutorial program for NCS.....	54



# List of Tables

Table 2.1 Distribution of robot tasks over three-layer system [24].....	16
Table 4.1 Average run times for a small and large simulation.....	43
Table 4.2 New and Updated keywords of NCS for VServer .....	44
Table 4.3 New and updated keywords for NCS for Autoport.....	45
Table 4.4 New and updated keywords of NCS for Single Socket Server.....	46
Table 4.5 Commands that may be sent to NCS during runtime.....	47

# Chapter 1: Introduction

There is much to be learned about the way mammalian brains work in controlling the body. Within the complex networks formed by neurons, information cascades and generates reactions that emerge as memory, consciousness, and behavior. Information enters the brain through the body's senses: sight, hearing, touch, smell, taste. These interactions with the environment are processed to become stimuli. They trigger the synaptic paths connecting one group of brain cells with another. Those cells then trigger synapses to more cells, and so on. Once a stimulus has set off a chain reaction, some response will usually happen. An arm raises, legs move forward, or any other possible motor response may occur. Whatever action is taken, it will most likely lead to new information collecting through the senses, new stimuli being created, and the whole process starting once again.

Computers offer a new method to further investigate the inner workings of the brain. As computer hardware drops in price, Beowulf clusters can be built with greater processing power and the ability to simulate the biological, electrical, and chemical properties within the brain. Simulations can separate the stages of neurological processing by having one application focus on replicating brain behavior and another gathering sensory information from interactions in an environment. A method for exchanging data between these processes is required. The Brain Communication Server (BCS) is the bridge linking the high level reasoning of the NeoCortical Simulator (NCS) with an entity, physical or virtual, moving about a real or artificial world. While NCS issues motor commands to an entity, that entity simultaneously gathers sensory data. BCS allows the two programs to exchange their results. BCS must be fast and efficient in order

to keep the rate of communication flowing from one source to another since the stimulus-response loop is continuous.

The rest of this thesis is organized as follows. In Chapter 2, the concepts and underpinnings of NCS are covered. From there, the applications that make use of NCS, CARL (not an abbreviation, but rather the name of a robot) and IVO (Intelligent Virtual Organism), are described. Next is a discussion of server programs already in development, their intended purposes, and why they did not meet our goals, therefore requiring the creation of BCS instead of using an off the shelf server program. Chapter 3 goes into the development of BCS. It covers the methods initially tried in earlier versions and outlines the problems and difficulties encountered that forced changes in those methods until the final version of BCS emerged. Chapter 4 is an analysis of BCS. BCS's performance during data exchange is examined in comparison to using static data files. Also the changes made to NCS so that it could make full use of the server and help the progress of the latest research project, the Intelligent Virtual Organism (IVO). Chapter 5 is the final discussion of how BCS performs at the tasks currently required of it and offers a glimpse into future projects that will make use of BCS to link with NCS and its ability to simulate the brain.

# Chapter 2: Background

## 2.1 NCS

### 2.1.1 The Biology of NCS

As knowledge of brain functions is uncovered, NCS seeks to incorporate them into models. So far, many structures and organizational patterns researched in physical matter [29] have been designed and implemented. The constructs and concepts included in NCS are described in the following paragraphs.

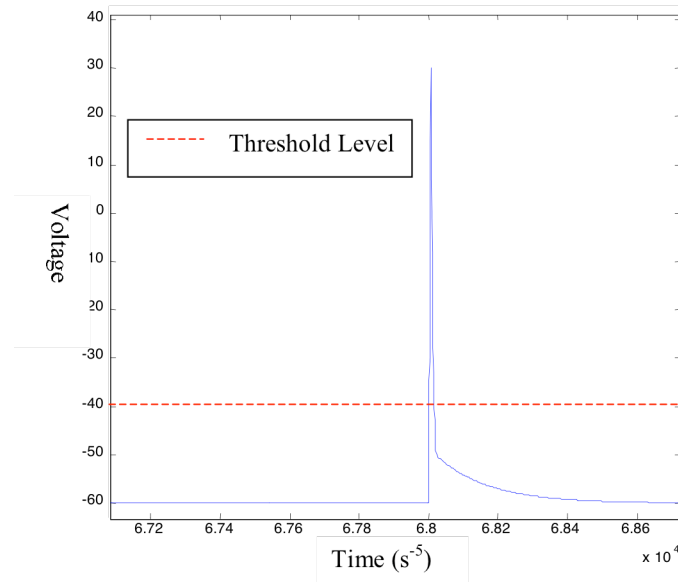
#### **Neurons**

One of the most basic objects, a neuron (denoted as cell from now on) serves as a container for compartments, described later. Collections of cells of the same type are designated as cell groups or cell clusters and are managed as single units for the purposes of synaptic connections, stimulus and report targeting, and load balancing during parallel simulations.

#### **Compartments**

A cell may be divided into multiple compartments. Each compartment keeps track of multiple state variables such as membrane voltage, as well as acting as a container for channels and synapse endings, described later. A cell, regardless of the total number of compartments, must have exactly one compartment that is designated as active. This “soma” compartment pays special attention to its membrane voltage as it adjusts due to incoming current from adjacent compartments, cells connected via synapses, and other sources. If the membrane voltage reaches a threshold value, the compartment generates a spike shaped template and sends an action potential (AP), a sudden burst of current, to all the other cells/compartments to which it is

connected via synapses. Figure 2.1 shows the voltage pattern for a soma compartment that has reached its threshold. When multiple compartments exist in a cell, voltage flows from the compartments where synapses terminate, dendrites and spines, towards the soma. Figure 2.2 shows a possible cell configuration with synapses terminating on dendrite compartments or the soma itself. When the soma fires an AP, it may send the AP itself, or have axon compartments send the APs to other cells/compartments. The process of collecting current and generating an AP is called “integrate-and-fire.” Information propagation through networks of cells is described in [22] and [23].

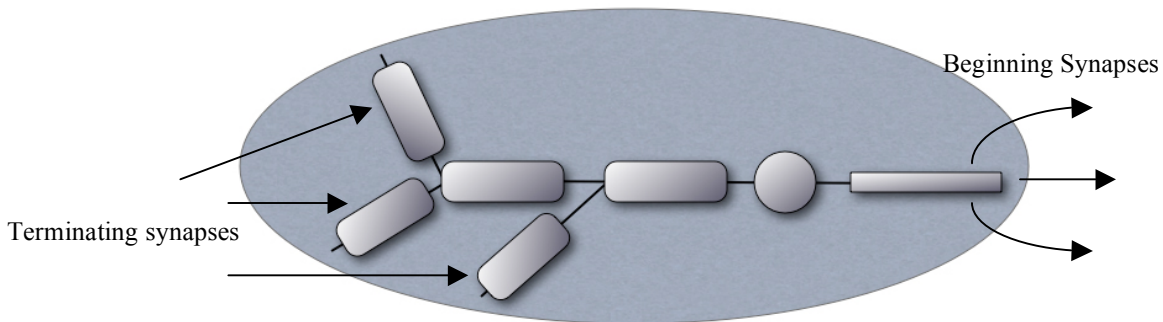


**Figure 2.1** Compartment reaches threshold and fires an Action Potential (AP)

## Channels

Within a compartment, channels regulate the flow of certain types of ions, dynamically impacting how the compartment’s membrane voltage changes either inhibiting or enhancing the generation of an AP from the active compartment. NCS has the capacity to simulate M, A, and

AHP channels and can potentially create more channels types in the future [18, 39]. M channels become activated by an incoming AP. An M channel seeks to inhibit its compartment and prevent it from reaching its threshold. An A channel opens up as a compartment approaches its threshold, but shuts off once threshold has been reached [16]. This has the effect of reducing the impact of background noise. AHP channels are different from the other types in that they are not voltage dependant. Rather, these respond to calcium ( $\text{Ca}^{+2}$ ) concentration within a compartment. As a cell generates its own APs, calcium enters the compartment, opening the channel so that it hyperpolarizes its compartment, possibly suppressing further firing.



**Figure 2.2** APs arrive on compartments, and new APs will be generated.

## Synapses

Synapses connect one cell to another and pass APs from source (pre-synaptic cell) to destination (post-synaptic cell). The spike shaped output of the soma in a pre-synaptic cell is considered a binary event due to the short length of time it lasts. However, the synapse turns the AP into an analog event for post-synaptic compartments since the resulting post-synaptic current (PSC) lasts one to two hundred microseconds, considerably longer than the original AP. Synapses are either excitatory or inhibitory. Excitatory synapses help post-synaptic cells to reach their threshold (depolarize), and inhibitory synapses suppress the ability of post-synaptic

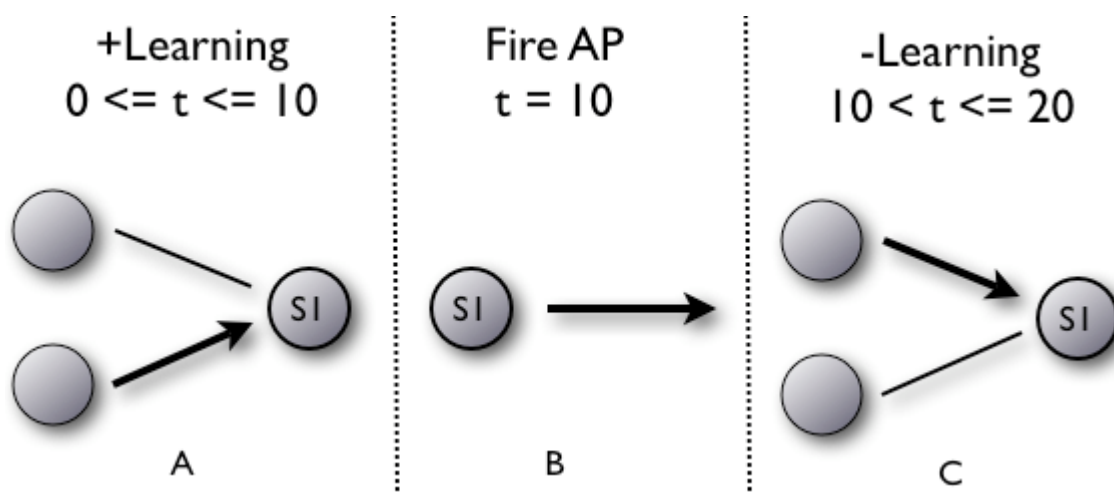
cells from reaching their threshold (hyperpolarize). The effectiveness of a synapse to depolarize or hyperpolarize is controlled by short-term dynamics called facilitation and depression as well as long term dynamics called Hebbian learning [34]. These affect a synapse's Utilization of Synaptic Efficacy (USE) which determines the release of more neurotransmitters (amplifying the signal) or less neurotransmitters (dampening the signal).

Short-term dynamics regulate neurotransmitters within a specified window of time. Two primary values dictate whether a synapse will be more or less effective as successive APs arrive. The depression time constant seeks to reduce effectiveness while the facilitation time constant seeks to promote it. A synapse adjusts its USE based on the relative values of its depression time constant and facilitation time constant. USE is not allowed to drop below 0.0 (no efficacy) or rise above 1.0 (full efficacy). The effectiveness of a synapse is not permanently affected by these properties; during periods in which APs are not arriving, a synapse can recover its USE back to its original state.

Hebbian learning does have a permanent effect on the USE of a synapse. This depends on the pre- and post-synaptic AP timings. A post-synaptic AP that arrives and helps cause a pre-synaptic AP will strengthen the synapse that delivered the AP. Otherwise, if the post-synaptic AP arrives too late, after the pre-synaptic AP has been generated, then that synapse is weakened. Figure 2.3 illustrates the timing involved where two pre-synaptic cells fire at different times. Cell A's AP arrives in a positive learning window and the synapse connecting Cell A and Cell C is strengthened. Cell B's AP arrives during a negative learning window and the synapse connecting Cell B and Cell C is weakened.

## Columns and Layers

Cells are organized into other container objects. Within a layer, the quantity of each cell type determines the size of the final cell groups or clusters. Layers are organized on top of each other to form columns. Once cells have been placed within these structures, synaptic connections are made using a probabilistic model. Cell groups within the same layer use the highest probabilities of connecting with each other. Connections between layers use the next highest probabilities, and connections across columns use the lowest probabilities.



**Figure 2.3** Hebbian Learning occurs over 3 stages.

- A. During a positive learning window, synapses that fire to S1 are strengthened
- B. When S1 fires, the positive learning window closes, and negative learning window begins
- C. During a negative learning window, synapses that fire to S1 are weakened

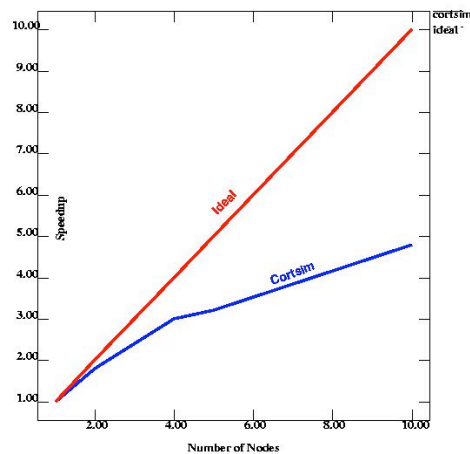
### 2.1.2 NCS1 and NCS2

The initial version of the NeoCortical Simulator (NCS) explored the potential of simulating experimental results obtained from rat brain slices [14]. Matlab was the chosen platform for development of the pilot program. Implementations of calcium dependant AHP channels, voltage dependant A and M potassium channels, synaptic delay, and membrane impedance were



tested and calibrated against the biological experiment recorded observations. Short-term synaptic dynamics and long-term refinements due to Hebbian regulation of synaptic efficacy were incorporated using recently published results from Markram and Colleagues [34]. A model comprised of 160 point-neurons (single compartment) in a 2-column architecture was used for testing and observing associations of input-output pairing.

To increase the scale of simulations and maintain reasonable performance, NCS2 was developed by converting the existing program into C code. In addition, parallelization of the code was developed and experimentation was carried out on a 20 node Beowulf cluster comprised of 450 MHz Pentium CPUs. Experiments were performed by increasing the size of the network from 100 cells to 1000 cells and increasing the number of CPUs from 1-10. Increasing the number of CPUs beyond 10 did not provide any more benefit to the runtime of NCS. Figure 2.4 shows the speedup results of increasing the number of nodes on a 1000 neuron simulation.



**Figure 2.4** Speedup results of attempting to parallelize NCS2 [15]

The disappointing speedup is due to slow switching across Ethernet. A proposed Beowulf for the next incremental version of NCS planned to incorporate high-speed Myrinet switching

along with faster CPUs. This would allow for greater complexity in the design of biological neural networks, in terms of both compartmental design and the number of cells and synapses that could be processed.

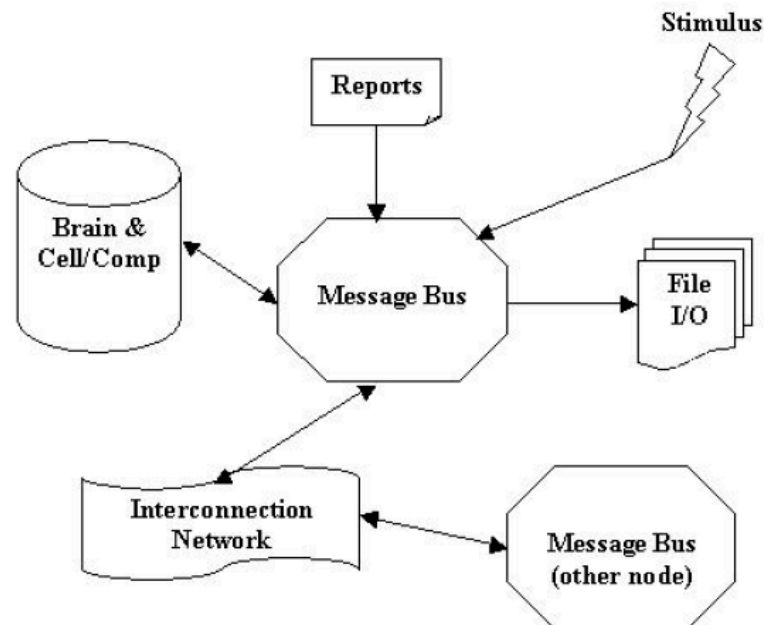
### 2.1.3 C++ Version NCS3

The third incarnation of NCS sought to utilize object-oriented designs to improve code performance and organization [36, 37, 38]. The relationships among neuronal properties were broken down into corresponding objects to allow for the completion of two goals. First, with objects fully encapsulated, different implementations could be swapped out without needing to rewrite large portions of code. This allowed for easier experimentation with algorithms to find the most efficient means of accomplishing biological simulation. The second goal was to allow the code to be better organized by having objects serve as containers and/or state machines. As a container, instances of other objects are stored, usually in lists. For example, a column object acts as a container for layer objects, keeping them separate from unrelated objects. As a state machine, biological tasks are performed that impact a system's behavior. A synapse object is an example of a state machine since it keeps track of its USE parameter as it changes due to short and long-term learning.

Since NCS3 was a complete rewrite of the NeoCortical simulator, steps were taken to make it more general for future improvements as well as capable of executing on a parallel system more efficiently. This version of NCS split up cells evenly across the nodes with no notion of keeping cell groups together, although load-balancing algorithms were anticipated to be included in the future. A Message Bus object was created to coordinate the flow of communication between objects on and off node. This managed the timing and synchronization of simulations to prevent

deadlock and starvation [8]. Figure 2.5 shows the interaction of the Message Bus with various other components of NCS3.

In addition to code improvements, new hardware was obtained to improve system performance and increase simulation size [14]. The Cortex cluster was built using 30 dual-Pentium III 1 GHz processor nodes with 4 GB of RAM per node. A new networking system built with Myrinet [4] was purchased and installed for this system. As connectivity was a main issue with previous versions, Myrinet's high-performance packet-switching technology allowed for high rate data transfers without flooding the network, as occurred with Ethernet.



**Figure 2.5 NCS3 communication model centered on a Message Bus object [36]**

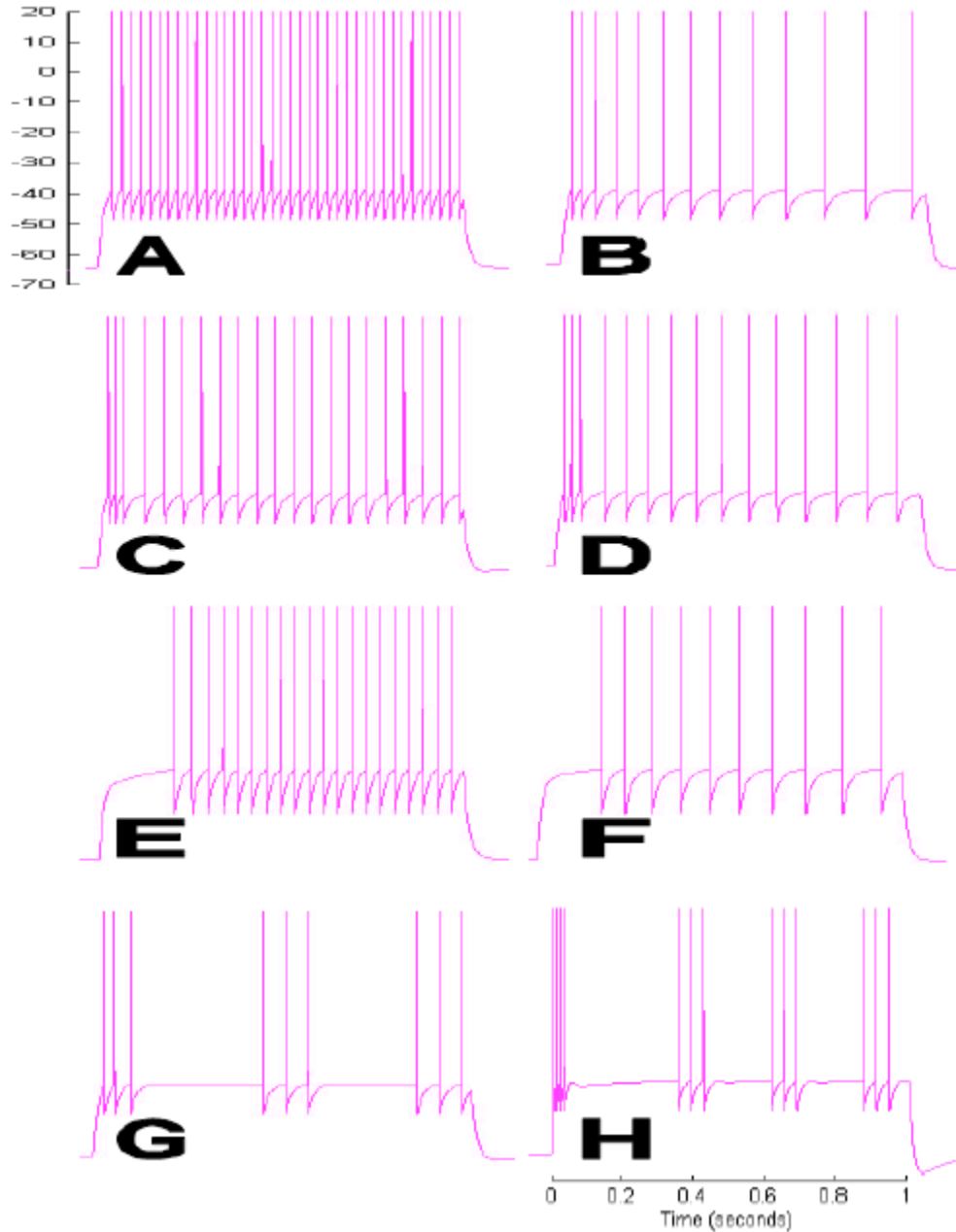
Using NCS3, experiments were performed to utilize channel structures within cellular compartments in order to reflect accurate dynamics of interneuronal GABAergic system [27]. NCS3 supplied M-, A-, and AHP-channels that could be tuned using parameters. By combining

these channels in a proper way, previously published *in vitro* responses taken after applying 1-second current steps ranging from 50 to 350 pA were replicated.

Biological research has shown that neocortical interneuron membrane responsiveness is more varied than previous neuronal classifications took into account [10, 16]. Studies have looked into the GABAergic system to understand its role in neuronal anatomy, synaptic dynamics, and membrane physiology [7, 16]. Using a point-neuron model, based on published data [16, 17], channels were developed and fine-tuned to reflect biological data [18, 20, 39]. Figure 2.6 shows the results achieved by NCS3 in replicating the behavior of various neurons: classic non-accommodating (cNAC), classic accommodating (cAC), bursting non-accommodating (bNAC), bursting accommodating (bAC), delayed non-accommodating (dNAC), delayed accommodating (dAC), classic stuttering (cSTUT), and bursting stuttering (bSTUT). Under NCS, the behaviors, except for “stuttering” types, were relatively robust under a two-fold variation in strength for a single somatic cell.

#### 2.1.4 NCS4 and NCS5

The next revisions of NCS sought to expand the functionality available to simulations as well as offer further optimizations to speed and performance [13]. Previous versions of NCS had been designed to work with point neurons (i.e. only a soma compartment). The latest revisions of NCS allowed for complex cells to be created consisting of numerous compartmental structures such as dendrites, spines, and axons. Additionally, geometric information could be assigned to the structures. For example, synapses would have a higher probability of formation among closely spaced objects and action potentials would travel faster to nearby destinations. The capability to save and load the state of a constructed brain was added. This allowed synaptic dynamics influenced by Hebbian learning to be preserved and reused for future simulations.



**Figure 2.6** Simulated response to step current of 150-300 pA [27].

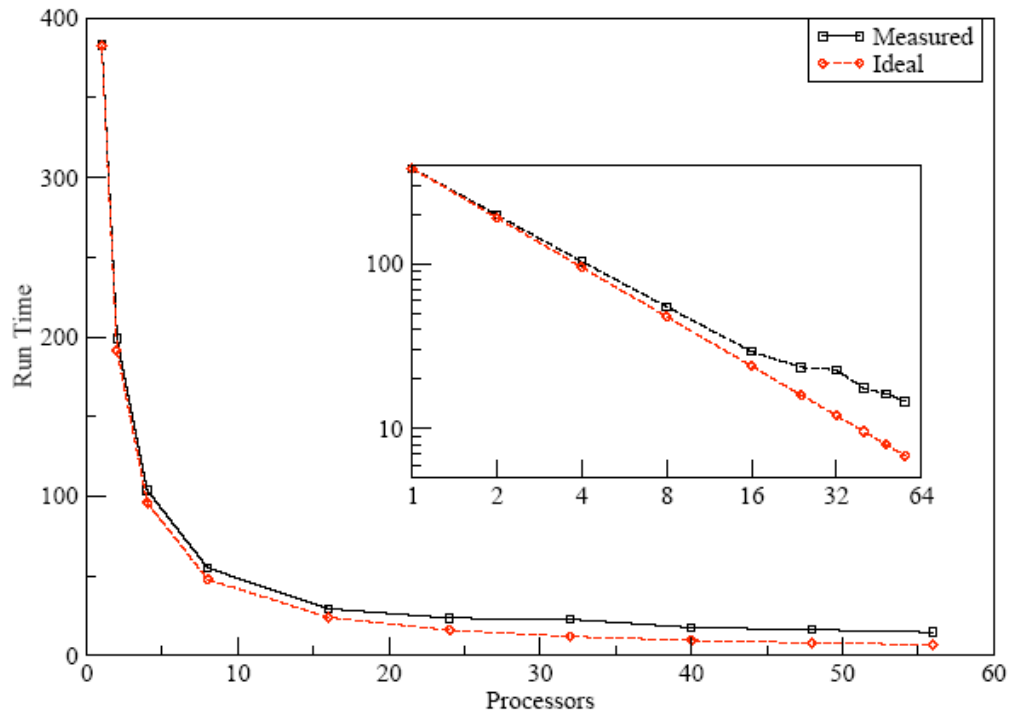
A. cNAC B. cAC C. bNAC D. bAC E. dNAC F. dAC G. cSTUT H. bSTUT

Numerous optimizations were made to NCS during the latest revisions. Improved load balancing was needed to ensure that some processors in the cluster weren't being overworked while others stayed idle [11, 12]. The method of load balancing in NCS3 had focused on

dividing cells evenly among nodes whereas the latest methods focused on the number of synapses in order to properly lay out the possibilities of “work” balance. Synapses comprised the largest memory usage in the simulation; thus, the need to evenly distribute total synapse count became more important. The connection process was also optimized so that less time was spent initializing the brain network. Older versions would visit every possible connection during the construction of the brain and check the probability of that connection being created. This yielded an  $O(n^2)$  process where the number of cells in the network ( $n$ ) could potentially be connected with each other. The new version looks at the number of connections desired, and selects available cells randomly to create a connection. This algorithm more closely approaches  $O(n)$  in execution. One of the other major optimizations was to tone down the Message Bus so that it produced less overhead. Message sizes were reduced and were therefore able to get to their destinations faster. Some objects no longer needed to use the Message Bus as they were made more intelligent in working with the brain. Figure 2.7 shows the results of the optimizations on NCS in the parallel environment. The inset graph shows the same data, but on a logarithmic scale.

More biological experiments were performed with the latest versions of NCS. One such experiment took a look at how spike-timing and membrane dynamics of biological neurons may encode information [26, 27]. It sought to make comparisons with artificial neural networks (ANNs) to show how the biologically influenced NCS was more flexible and robust and could utilize multiple sensory modalities as humans do in situations requiring pattern recognition, speech comprehension, and path planning. Audiovisual recordings were decomposed and presented to the synaptic and membrane dynamics obtained from laboratory-derived parameters.

The results of these experimentations would be used in the development of the CARL application described in Section 2.2.1.



**Figure 2.7 With optimizations, NCS achieves speedup closer to ideal [11]**

Another experiment examined intracortical and cortical-subcortical “networks of networks” involved in attention, sensory association, motor planning, memory, reward, theory of mind, and emotion [30]. It attempted to understand how these modules interact using NCS as a platform for rapid experiment development and processing. After performing literary synthesis of existing research and incorporating the designs into virtual brains, models ranging from 1,000 to 1,000,000 neurons were constructed to sustain auditory patterns with stability. Synapses used in the simulation approached 600 million creating a massively interconnected network.

A look at entropy and its association with data distribution in a neural network was looked at in further experimentation. First, a look at entropy in biological models was created to establish

a foundation [26, 27]. Further experiments looked at the impact that AHP channels have in the transmission of information; that is, determining the amount of entropy in the system [28]. By knocking out the AHP channel, entropy substantially lowered, which indicated a reduction in the information carrying capacity of the network. Entropy measurements were taken using methods developed by quantifying spike codes recorded from a fly's visual system [6, 33].

Experimentation continues in both real-world biology as well as NCS's silicon environment. As new discoveries are made, both fields can benefit. NCS can expand its simulation capabilities by including new structures and biology. Biologists can perform experiments that might not be feasible under real-world circumstances.

## 2.2 Applications

Once NCS reached a threshold level of biological realism, projects were developed to make use of the simulator. These applications created the stimulus that initiated dynamics within NCS. In turn, report files from NCS initiated reactions on applications. The first major application created to work in this manner was a mechanical robot dubbed CARL (note that CARL is not an abbreviation). An ongoing project focuses on creating an Intelligent Virtual Organism, or IVO for short.

### 2.2.1 CARL

In 2002, work was done to create a system that mimicked interactions between the brain, the body, and the environment using a robot body controlled by a small on-board computer, a local computer or laptop, and a Beowulf cluster of CPUs [9, 24, 25]. Juan Carlos Macera, who worked on this project, sought to meet the timing demands of complex thinking while maintaining a sleek, light robot body capable of interacting with the environment. Complex



thinking in a reasonable amount of time often meant carrying heavy, multiprocessor systems which would reduce the dexterity of a robot body. By keeping two systems separate as described by Inaba et al, a robot used wireless communication to gain the benefits of faster, more complex processing from the multiprocessor system without compromising maneuverability [19]. Juan Carlos went even further by expanding the two-system model into a three-layer hierarchal robotic control system. These layers were labeled the Body, the Brainstem, and the Cortex to associate their behaviors with corresponding biological structures.

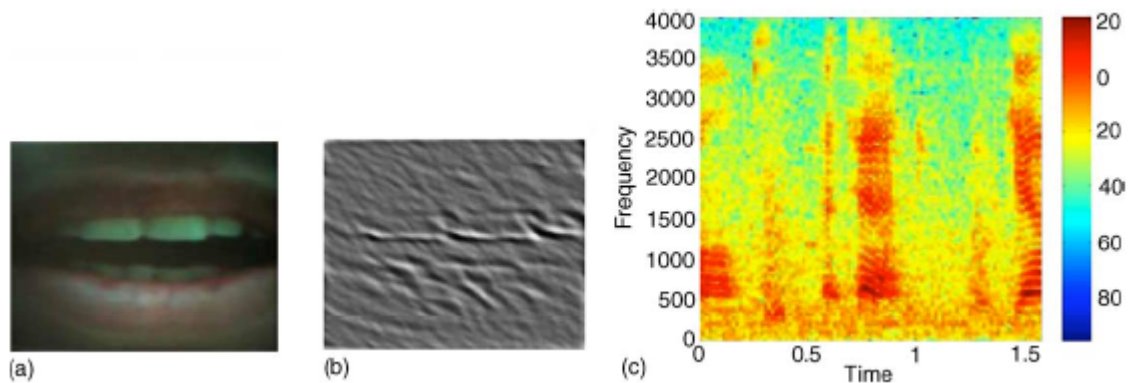
The Body layer corresponds with the robot body, which directly interacts with the environment, gathering stimulus. The Brainstem corresponds to a local machine where “instinctive” behavior allows fast responses. It also relays information to the Cortex and back. The Cortex corresponds to NCS which handles higher-level cognition to formulate responses to complex situations. Simple tasks are handled by the lowest layer and incorporate other layers as needed. Table 2.1 shows the breakdown of certain tasks across the three-layer system.

**Table 2.1** Distribution of robot tasks over three-layer system [24]

<b>Application</b>	<b>Body</b>	<b>Brainstem</b>	<b>Cortex</b>
Obstacle avoidance & navigation routines	x		
Binaural sound localization	x	x	
Navigation to sound target	x	x	
Robotic control over the Internet	x	x	x
Bimodal speech recognition (ANN)	x	x	x
Bimodal speech perception (SNN)	x	x	x

One of the primary experiments performed using CARL was to incrementally triangulate and navigate towards a speaking target. By giving audio and visual speech information to NCS, the target would then be classified as a threat or non-threat based upon any potential danger so that CARL could take an appropriate tactical response [25]. The audio processing is handled by short time Fourier transforms that return sets of energies for 129 frequency bands. These energy

values are written to a file that NCS can open and read, then convert the information into probabilities of synaptic firings. The probabilities are applied to a cell group designated as the auditory cortex as stimulus. Video data, meanwhile, is processed by a Gabor filter where sub-regions of a frame are converted once again to spike probabilities. These are likewise output to a file to be processed and fed to a cell group designated as the visual cortex [26]. Figure 2.8 illustrates the conversion of audio and visual data into spike probabilities for NCS. Using these tactics, the behavior of CARL was fine tuned to achieve the project goals.



**Figure 2.8** Preprocessing of visual and audio data [25].

- (a) Video frame of the speaking mouth target
- (b) Gabor analysis of the mouth frame
- (c) Spectrogram of the speech captured.

The communication method uses a wireless link between CARL and the local machine, and a wired link from the local machine to the cluster running NCS. A simple, coarse grain communication pattern was implemented. The robot can move, stop and gather stimulus for a period, and transmit to the local machine. The local machine compresses the data to improve the speed of transfer to the cluster. When the local machine transferred data, it would send it as one large file, and via ssh, start a new NCS job using the chunk of stimulus data. NCS would go

through all its initializations to construct the brain and all its components, work with the data file, and produce a report for the complete simulation, then shut down. The final report is interpreted in order to issue a command to CARL.

### 2.2.2 IVO

CARL demonstrated that an external entity could make appropriate motor responses after exchanging information with NCS. The next goal was to fine tune the simulator so that it could exhibit greater memory and learning. A virtual organism was selected as the platform for experimentation. Using Matlab, an Intelligent Virtual Organism (IVO) was designed so that it could send and receive data from NCS. This holds key advantages over CARL to facilitate the research process.

One advantage is that using a virtual organism would simplify maintenance. As a physical machine, CARL had numerous mechanical pieces that needed to be maintained. Effort was spent repairing and replacing worn components. Sometimes, the failing components might not be so easily determined and numerous attempts were made to find out which part was faulty before future simulations could be run. These components were sometimes expensive as well since CARL was designed with small, delicate parts. By moving to a virtual environment, no physical piece of the IVO could fail and research time could be spent working with NCS.

Additionally, as a physical object, CARL stayed in the research lab. In order to perform experiments, researchers needed to be near him to observe behavioral responses. As inspiration may come at anytime, being forced to travel to the lab to make every change seemed less productive. An IVO has the advantage of being highly portable. Anyone can run the IVO program remotely from a home computer and see its actions of IVO on his or her monitor. Also,

for demonstration purposes, an IVO can be shown off more easily since IVO can be downloaded from the Internet as opposed to transporting a system like CARL.

By using IVO, the focus of research was clearly in fine-tuning NCS to achieve the level of learning and responses we needed. The goals of IVO were to work out the model of the brain first, then make that model available to any platform. CARL had shown a glimpse of what was possible with NCS. IVO provided a faster, more efficient way to work with NCS. IVO was also designed from the start to use fine-grained communication implemented with BCS so that it could send and receive data from NCS as it became available.

## 2.3 Analysis of other Server Packages

### 2.3.1 Overview of limitations

Many of the following server tools provide the ability to customize to fit user needs. Unfortunately, the amount of retooling needed to change any of these projects to fit the needs of IVO and NCS would have required a greater amount of work than building a specialized server. In addition, documentation for many of the available tools is written for individuals whose work stays close to the mainstream purposes of Internet activity whereas the work done at the Brain Lab is far from mainstream. Without documentation specially written to assist in meeting the goals of NCS, time would need to be devoted to use available tools in non-conventional ways. The worst-case scenario being a retooling of source code to meet exclusive needs.

These projects are great at what they do; they are very robust and full featured and should be used for projects that have use for their abilities. However, all these features are unnecessary for this project and result in excess overhead. With a goal slated towards real-time operation, as much overhead as possible needs to be eliminated; otherwise, communication overhead adds up

rapidly when for example, a single second of simulated brain-time is divided into 10,000 time increments.

### 2.3.2 Apache

Apache is an open source HTTP server [2]. It seeks to provide a secure, efficient, and extensible server for current HTTP standards. A commercial-grade HTTP server, Apache is provided by means of a collaborative development project working towards robust, featureful, and freely available source code. By accepting work from dedicated programmers world-wide, Apache has grown into one of the most widely used server applications, existing on approximately 64% of the servers on the web. Apache provides full access to HTTP standards, helping to maintain the integrity of the protocol and ensuring that the Internet serves as a "level playing field" for any entities seeking to develop for the web.

Apache is designed for expansion through the use of modules. This allows new features to be added with the durability of the underlying server. In an unpredictable environment such as the Internet, such robustness is essential. Users bombard a system and mishaps, both intentional and unintentional, occur constantly. However, such heavy-duty security is not needed, and only hinders our activity.

### 2.3.3 Squid

Another product recommended during an initial look for existing servers was Squid. A proxy-caching server for multiple services such as FTP, gopher, and HTTP, Squid offers high performance for web clients [5]. It allows Internet objects to be stored on a closer system to speed access from clients.

There is no speed of efficiency advantage gained by using Squid as opposed to creating BCS. It is best used in an environment where data is repeatedly accessed at varying times by one or more clients. For example, a business might use a proxy server to create a local copy of a web page so that employees within the company can access it more quickly without consuming as much bandwidth. With the communication scheme used between IVO and NCS, data is needed as soon as it is generated, and only used once. Storing copies of the data that will never be needed again accomplishes nothing.

# Chapter 3: Development

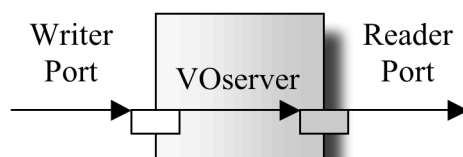
In developing the Brain Communication Server, various methods were implemented and tested. As flaws and unexpected cases emerged, modifications were made so that a more refined communication server would emerge and satisfy all requirements. In this chapter, the history of developing BCS is presented.

## 3.1 First Approach: VOserver

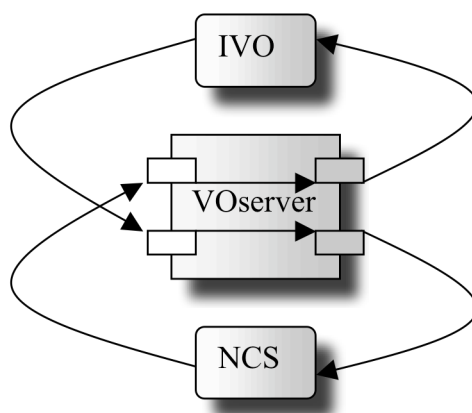
Initially, the server was intended to work only with one other program. After NCS achieved a level of biological realism, an Intelligent Virtual Organism, or IVO for short, was created to further test the capabilities of NCS after the CARL project wrapped up. The server was therefore designed with the specific purpose of coordinating data between IVO and NCS more fluidly than had occurred with CARL when it was forced to accumulate data over an extended period of time and then allow NCS to start from scratch on the collected data. This virtual organism server, or VOserver, accepted data from IVO and sent it to NCS as stimulus dynamically as it was received. NCS in turn sent data back to the server as a motor response for the IVO to execute continuously rather than in large, discrete blocks. During execution, the VOserver need not differentiate between the two types of data streams. As shown in Figure 3.1, VOserver merely accepted whatever was sent to a “writer port” and directed it to a “reader port”. Any client connected to the reader port received the pending data. So long as NCS and IVO connected to the correct ports, each would receive the data sent from the other application.

Figure 3.2 helps to clarify by showing a simple connection scheme. Once data pathways had been set up, the IVO could connect to one of the writer ports while NCS connected to the reader

port corresponding to that writer. Meanwhile, NCS connected to a different writer port and IVO connected to the corresponding reader port.



**Figure 3.1** Data enters the server from the writer port, leaves through the reader port

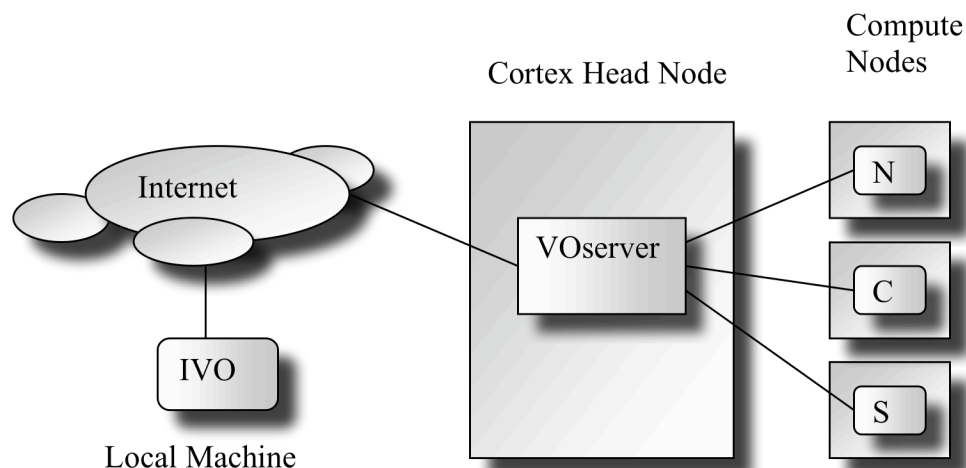


**Figure 3.2** Simple connections scheme for data exchange

Additionally, the change in communication model used by CARL had another effect that facilitated the need for VOserver. Because NCS was always restarted with CARL, the simple communication program it employed needed only contact the head node of the Cortex cluster and initiate a new NCS run. With NCS always running, an outside application could not connect to the specific nodes where NCS actually ran since only the head node was accessible from the outside world due to security restrictions. Figure 3.3 shows the organization of the Cortex system when NCS is running and how VOserver fit into the scheme. Users working from home or from another lab could connect to the VOserver application since it ran on the head node of



the Cortex cluster, then VOserver would have access to the individual nodes.



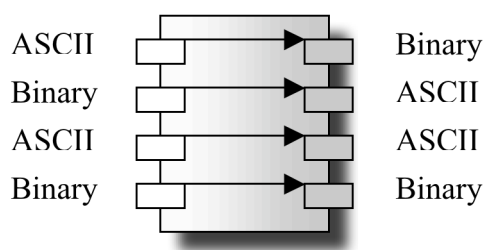
**Figure 3.3** NCS nodes are kept separate for security.

### 3.1.2 Using VOserver

VOserver used two ports for each one-way flow of data. This meant that each port used by VOserver was responsible for only reading or for writing and was linked with only one other port. Thus, no header information needed to be sent by any clients connecting to VOserver. Ports were assigned the responsibility of reading or writing using a configuration file passed to VOserver as a command line argument. The file had a very simple initial design where the first number in the file was the number of writer/reader pairs and the following lines contained the pairs of port numbers, with the writer port followed by the reader port.

In addition to the port numbers in the configuration, the user needed to specify whether the reader and the writer would handle data in ASCII mode or binary mode. The two formats were made available because each had advantages and disadvantages that could affect a user choosing one over the other. ASCII was easy for a user to work with and quickly debug, but consumed more bandwidth to transmit data. Floating-point numbers transmitted in ASCII were often

truncated to only a few places after the decimal, reducing their accuracy. On the other hand, binary would maintain full precision for floating point values and used the same amount of space for each value, saving bandwidth. Users wanting to view or debug binary values needed to know how to view them using the appropriate tools. In order to help accommodate the user, a conversion function was added to the server so that data could be changed from ASCII to binary or binary to ASCII. Figure 3.4 illustrates data coming into the server as one form, and if the output data format is not the same, VOserver will handle the conversion to the other format.



**Figure 3.4** Input data may convert from ASCII to binary or vice versa.

This conversion was especially useful for sending stimulus to NCS since it had been designed to receive only in binary (NCS sent reports in either binary or ASCII). In addition, in working with Matlab to build the IVO, it was difficult to understand how to make Matlab use the same binary format as NCS, so from the user's perspective she/he could just use ASCII on IVO's end within Matlab, and let the server convert the data to a binary representation that NCS could understand. Although this put more work on the server to handle, the conversions would not be used as long as data remained in the same format going from one port to the next.

The last function handled in the configuration file allowed the user to display verbose output for a given port so that it could be monitored in a terminal during runtime. In order to keep the configuration file from becoming cluttered, this feature was merged with the ASCII/binary mode

indicator. If a port was to be verbose, a capital letter was used, 'A' or 'B', whereas a lowercase letter, 'a' or 'b', would leave verbose output turned off.

As the server program developed, a means of giving commands directly to the server during runtime became necessary. The user could designate a separate port for issuing server commands on the command line along with the configuration file. The commands available to the user are described in a later section.

The final command to start the server appeared as:

```
VOserver <configuration file> <control port>
```

### 3.1.3 Data Format

In order to decrease overhead in sending data, a minimal header was introduced for all transmissions. It aided in converting data whenever that function became necessary. The header required the sender to specify the total number of bytes in the main message. This value was expected to be in ASCII followed by a white space character (' ', '\n', '\t', '\0'). By sending the length value in ASCII even if the stream was using binary, this simplified the parse function on the server that would extract the data. In addition, by using ASCII, it would be understandable by any users of the server. When data from a writer arrived, the header would be removed and the extracted data stored until it was sent to the reader. Figure 3.5 shows an example ASCII transmission. Figure 3.6 shows an example of a binary transmission. Note that the length value does not include the space taken up by itself or the separating white space.

Any additional formatting needed by the reading application would need to exist outside of the header used by VOserver. For example, when NCS reads stimulus, it reads a constant size, so no additional formatting is needed. When NCS writes an ASCII report, it ends each line with a new line since each report line could be of variable length. This new line character remained in

the data stream and was sent on to any client applications even after VOs server had removed its own header from the data. VOs server's header was included so the server could distinguish a whole block of data and properly queue it for the reader. By knowing when the whole block of data had arrived, VOs server could correctly convert data without mistakenly breaking apart values that might have only partially arrived on the port.

9	\0	-	6	5	.	4	3	2	1	\n
---	----	---	---	---	---	---	---	---	---	----

**Figure 3.5** Example header used when sending ASCII data.

4	\0	#	#	#	#
---	----	---	---	---	---

**Figure 3.6** Example header used when sending binary data.

### 3.1.4 VOs server commands

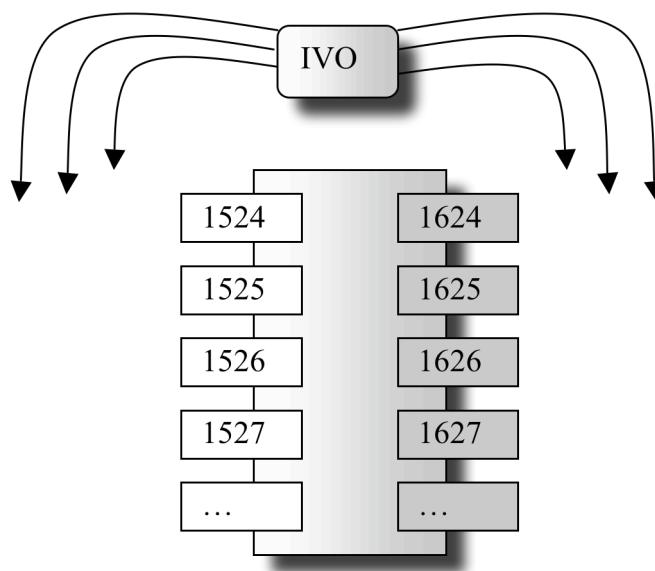
As the server was used, the need to control the server as it ran emerged so that various commands were created. The first server command conceived was a reset command to keep the server from needing to be shut down every time the user wanted to perform a new simulation. The reason behind this reset command is addressed in Section 3.1.5. With the server always active, additional commands needed to be added to help with the verbose output. For example, if a port was set to have verbose output, and the user had become satisfied that communication was working properly, she/he should not need to shut down the server in order to change system the configuration. Therefore, another command was added so that a user could specify a port's verbose output be turned on or off during runtime. These commands, reset and verbose control, were the initial attempts to control the server during runtime. Later versions of BCS would expand the number of commands a user could use to manage the server.

### 3.1.5 Difficulties with VOserver

VOserver had been designed to be simple so that it could transfer data with as little overhead as possible. However, that simplicity led to difficulties in getting the server to work the way the users wanted it to and work around solutions were not always forthcoming.

One of the initial problems that came up occurred whenever the server was shut down. The ports it had been using would remain reserved for a period of time set by the operating system in order to assure that new applications would not use those ports and read potentially left over data[32]. This was a great inconvenience when a user wanted to make minor changes to the interaction between the IVO and NCS. To try and solve this problem, a reset command was added to the server so that it would close any open connections with other programs (i.e. NCS and IVO) and clear any pending data from its buffers. This would allow the server to continue running and keep the ports it had already claimed. The user would make any changes and then reconnect to the ports without worrying about data from the previous run corrupting the current run. Although this new feature helped make consecutive runs easier, other difficulties still existed.

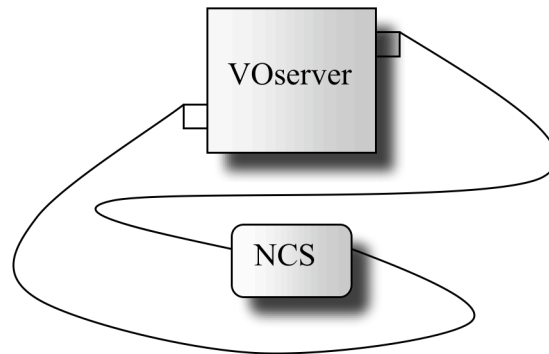
Numerous data streams could be added to VOserver since multiple stimulus and reports are needed for NCS to better simulate a biological organism. As the number of data stream increased, working with the port numbers became increasingly difficult since a number does not describe the purpose of a port. Figure 3.7 illustrates that connecting an IVO to the appropriate ports could easily become confusing since the port numbers are not descriptive enough to distinguish one from another.



**Figure 3.7** With many ports, coordination becomes difficult

Additionally, if the IVO or NCS connected to the wrong ports, the mistake might be difficult to discover. Figure 3.8 shows NCS connecting a report writer to one of VOserver's reader ports while NCS also connects a stimulus reader to one of VOserver's writer ports. Such mistakes might allow the simulation to run fine while it reads erroneous data or it might lock up while it waits for data that will never arrive. Even when a user realized that a mistake existed in the connections, debugging these connections was a tedious chore. The user would be forced to spend much time reading through text files making sure NCS and IVO connections were all properly directed to the correct port numbers when a simulation was started. If changes to the connection scheme needed to be made or if multiple simulations were to be run at the same time, the user would need to repeat the process of reconfiguring port connections all over again. For the case of multiple simulations running at once, each simulation needed its own set of ports to work with since a given set of ports could not be shared. When these situations arose, the user

would spend time coordinating three separate files: IVO's Matlab file, VOserver's configuration file, and the NCS input file.



**Figure 3.8** Bad configurations were difficult to detect and debug.

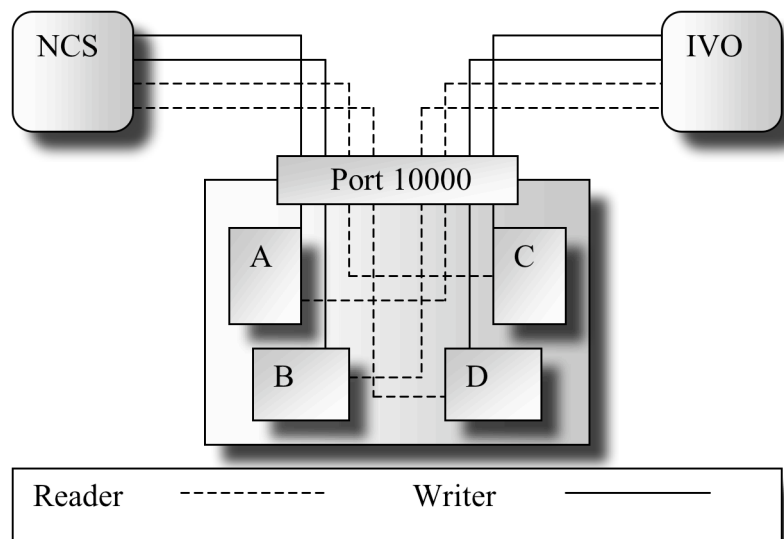
## 3.2 Second Approach: Autoport Server

Because the problem with port organization was becoming unbearable, a better solution was needed. Instead of forcing the user to select ports beforehand, the server should determine if it needs to create a new port while it runs. As the “autoport” system was being developed, it became apparent that creating individual ports was not necessary. Rather, the server would merely manage the sockets created during the connection process. Despite the slight misnomer, the name “Autoport Server” was used for simplicity.

### 3.2.1 Behavior

The idea behind the Autoport Server was to have the IVO and NCS tell the server what they intended to read or write. For instance, if NCS wanted to find a stimulus named ‘reward’, it would request the Autoport Server connect NCS as a reader to a writer for the ‘reward’ stimulus. The server would look through the already existing connections to see if any sockets were named ‘reward.’ If it was found, the reader would be linked with that socket so that as data was written

there, it would be transferred to NCS. If no socket was found with the name ‘reward,’ the Autoport Server would create a new object with that name and associate the reader with that object. When a writer then arrived with that name, it could be associated with that same object to create the full connection. Figure 3.9 illustrates this operation. The IVO and NCS connect only to one port on the Autoport Server. Internally, the server locates objects by name and creates an association with the socket.



**Figure 3.9** Autoport Server manages connections for the user.

Sockets from clients link with data objects (A, B, C, D) when communicating with main port.

### 3.2.2 Identification Header

In order for the clients such as the IVO and NCS to use the new Autoport Server, changes needed to be made to the way these and any other programs communicated with the server. There would only be one actual port opened by the server where connections are made. The command port created in the original VOserver for receiving reset and verbose commands would now be the only port. A new command, “request,” was created to handle when clients contacted



the server with a request to be connected to another client for transferring data. Along with the request tag, an identification header would be sent in order to establish the name of the connection, the format of the data, and the purpose of the connection. These fields were allowed to have variable length, so a new line character separated them, allowing the server to know where one field ends and another begins.

The Autoport Server does not place any restrictions on what a connection is named. Instead, the clients are expected to use a consistent standard so that each will be able to recreate the names on their own. Since NCS is the driving force behind the development of the server and IVO, the method of building a name came from pieces of information contained within the simulation input file. Specifically, a root name is built from fields describing the brain type and active brain job that normally help to identify the simulation. Then the individual reports and stimuli have a field named Filename to append to the root name. The IVO would be given the same pieces so that it could construct the name in a similar manner.

After the name of the connection came the data format. Data formats included the ASCII and binary formats from the VOserver along with a new format called “constant binary.” With both ASCII and binary, each time step that NCS or IVO sent data to the server, the line was first written with the total number of bytes contained in the data chunk of the transmission. This would add a small amount of overhead the beginning of each transmission. Although such a small amount of extra data seems insignificant, NCS breaks each second of simulated time into 10,000 time steps. Even though typical simulations lasted only two seconds, with an eventual goal in mind that simulations should last beyond a mere couple of seconds, this overhead would further interfere with trying to achieve near real-time simulation. The constant binary option was devised based around the fact that most binary transmissions would be the same length

throughout the simulation. This length could be determined at the beginning of the simulation and used whenever the data was written or read.

The last component of the header was the purpose of the connection. This purpose was simply whether the connecting application intended to read or write data. With the original VOserver, the writer had always connected to the port listed in the first column. Under the Autoport Server, the reader or writer could connect at anytime and in any order. By specifying the intended purpose of the connection in the header, a reader could connect first and wait for the writer, or a writer could connect first and begin sending data.

### 3.2.3 Updated and New features

Under the Autoport Server, it was still necessary for the user to have some method of viewing activity in order to ensure data transferred correctly. The user had the ability to turn on and off verbose output for the server's main port, but finer control was needed to manipulate verbose output on each of the named connections. In this way, the user could view only the connections she or he was interested in viewing and keep the remaining connections silently at work. The original verbose command was therefore updated so that it could accept a connection name in determining which connection needed to be turned on or off.

The server reset command also needed an update. Upon receiving the reset command, the server destroyed each of the connection objects containing the name of the connection and the socket numbers of the clients communicating with the server. If the same names were given to the server, they would need to be rebuilt, but if a new simulation with new names was used instead, then keeping the old objects would have been a drain on resources. Destroying the connection objects did not have the same impact as closing a port since working with sockets did not result in a time delay while waiting for the operating system to verify port security.

A new command was also created for users of the Autoport Server. During development of the Autoport Server, the specific nodes and CPUs used by the simulations were predetermined by the user. Although Autoport Server's naming convention allowed multiple similar jobs to run by changing the 'job' field of the input file, these simulations would try to use the same computing node in the cluster. This was because the helper script used to launch the simulations needed to read a file containing the desired nodes. Thus, multiple launches of NCS resulted in each reading this same file. In an attempt to remedy this problem, a new server command was created to start the NCS simulations through the server itself.

The "launch" command was designed to take a set of arguments from the user so that it could launch the simulation on any set of nodes. This was even available if the intended NCS simulation did not make use of run-time communication, but merely executed by itself until completion. The arguments included an input file to use, an NCS executable to use, the nodes to use, and the files where standard output and standard error should be written. Although, the user could use this information to launch NCS separately, the process of repeatedly editing the helper launch script was tedious, especially for new users unfamiliar with scripting languages.

The input file could be given to the 'launch' command as either a filename directly accessible by the Autoport Server, or the entire contents of the file would be streamed to the server from a remote host and then stored locally for NCS to use. The nodes were handled in a similar way such that a file listing the nodes could be specified by name or created on the fly. The NCS executable was handled differently, however. A new file was created for the server to read at the time of start up. The 'server.apps' file contained the name of all the executables the server was allowed to run using MPI. Each executable was then given an id number. The client would then send this id number instead of a name, with id 0 being the default executable.

Obtaining the list of executables is described shortly. The final arguments were merely filenames where the standard output and standard error files were written as the simulation ran.

To assist the launch command, the “applist” command allowed a client to receive the list of available applications. When the server received the applist command, it sent the contents of the ‘server.apps’ file to the requesting client. The client could determine which id to assign each application by the order the names were received. The first application had id 0, making it the default. The next had id 1, then id 2, and so on. Each application name was of variable length, so new lines were used to separate one from another.

The Autoport Server handled simulations of moderate size very well. With the Autoport server managing where the writers and the readers sent and received their data, the user had less maintenance to deal with. However, the Autoport Server had limitations that revealed themselves as more complex simulations were developed.

### 3.3 Third Approach: Single Socket Server

As research with NCS expanded beyond just the IVO, a greater variety of simulations were being developed, among which were ones causing the Autoport Server to create over a thousand sockets. The UNIX system used by NCS allowed for a single process to have a limit of 1024 sockets open at once [32]. When this threshold was reached, no new sockets could be created, preventing the simulation from being executed properly. For these large simulations, a new method of communicating with the server was needed that would keep the server program from reaching this limit.

### 3.3.1 Description

The Single Socket Server was created so that a client would need to use only one socket when communicating with the server. The client would use this single socket for reading and writing all the various data being exchanged with NCS. Additionally, more commands intended to manage the server itself were created so that the user could better manage an NCS simulation remotely. These new features prepared the server to deal with a greater variety of applications for communication with NCS.

The Single Socket Server opened one port for communications like the Autoport Server. This was actually the same port since the Single Socket Server could still accept Autoport connections. This was done for backwards compatibility and because in some instances, Autoport connections would be a better choice in terms of efficiency. The efficiency aspect is described later. When a client such as an IVO or NCS connects to the Single Socket Server, more descriptive headers than the ones originally used must be sent to the server since a socket must be able to give multiple commands at arbitrary times. The server cannot make any assumptions about the connection; otherwise, it risks corrupting the data being passed over the socket. The new headers affect the handling of traditional stimulus and reports, as well as the commands issued to the server.

### 3.3.2 Modified Handling of Stimulus and Reports

Since a single socket was used in the transfer of data, the clients needed to use a complete header for each transfer between itself and the server. For example, an IVO wishing to send a stimulus and read a report from the Single Socket Server would need to properly label its requests since assumptions could no longer be made based on the socket number. Instead, when

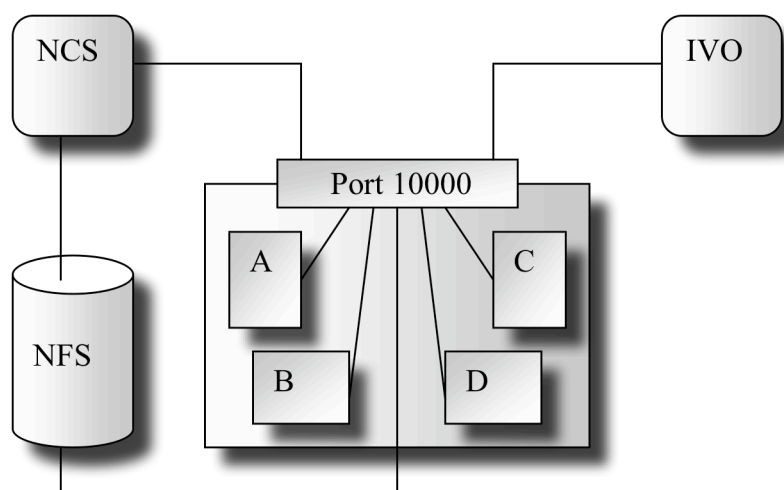
the IVO was ready to send the stimulus, it would initiate communication, sending the name of the stimulus, its intent to write, and the stimulus contents. When the IVO was ready to read a report, it sent the name of the report and its intent to read. At that point, it began listening to the socket, waiting for the server to send the data. During a stimulus event, a server locates the place in memory where it stores data for a named object (or creates a new location if it does not exist) and queues newly arriving data until another client requests that data. During a report event, a server accepts a name and locates where data for that name is stored. If there is no data currently available, the server continues handling other requests. When the data arrives, it will send it to the requesting client. If the data already exists, then the server can remove the first available data and send that.

The names for stimuli and reports are built the same way they were with the Autoport Server. The brain type and brain job form the root of all names with the filename of the stimulus and reports appended to the end.

### 3.3.3 New Reporting using NFS

In expanding the behavior of the server, a new method of handling reports was also developed. This allowed a user to let NCS write reports to the Network File System (NFS) as it normally does for non-dynamic communication. Larger reports need not need be kept in memory, and the contents of the report files could be manipulated and arranged to suit the user's needs. Figure 3.10 shows how the Single Socket Server requires only the single connection from client applications, and accesses NFS to acquire larger reports. The clients NCS and IVO connect to single port provided by the Single Socket Server. Internally, the server manages the movement and storage of data into objects based on naming conventions.

One goal of using the traditional reporting to NFS and then accessing those reports through the Single Socket Server was to look at data across reports for a specific block of time. Using direct communication from NCS to the server, reports only give data sequentially from the first data sent to the last data sent. If the user wished to look back at the middle of the simulation, the client needed to have saved the data and be able to access the desired locations. Now, with the option of using NFS to store reports but still access them with the server, new commands were introduced to let the user manage and manipulate report files.

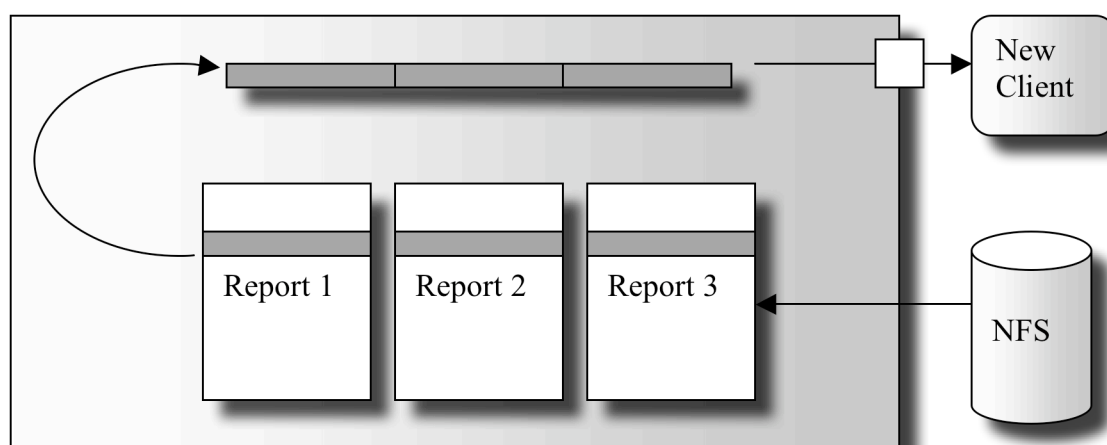


**Figure 3.10** Single Socket Server handles larger simulations best.

Clients connect through a single port and the server manages the data objects (A, B, C, D).

The primary command used to collect data from the NFS reports was the ‘getdata’ command. After the server received parameters to indicate which files and which time steps the user was interested in, the Single Socket Server would open those files and combine the same time steps together. Figure 3.11 shows how the time steps were extracted, combined, and sent rather than having one report send all its data as one big block before moving on to the next report.

In order to specify which reports the user wanted, six properties of a report combined to form a pattern. These properties included the report name, what values were being reported, and identifiers for a particular cell cluster: column name, layer name, cell name, and compartment name. These properties were embedded in the header of a binary report file. The server was given the layout of this header so it could extract and interpret the values. Multiple values for a given property could be specified, potentially making patterns long. A “setpattern” command was introduced to allow a user to predefine a pattern and receive an integer ID for future reference to that pattern. Finally, any property could be given a wildcard (“\*”) so that it would match any report.



**Figure 3.11** Extract data from multiple reports and converge the data based on time step.

To help the user manage a high volume of reports, the ability to create subdirectories and switch between them was introduced by adding the “mkdir” and “setpath” commands. By using these commands before launching an NCS simulation, the resulting report files would be placed in the current directory so that they could be kept separate, from the reports of previous simulations.



As the Single Socket Server was being developed, the Sun Grid Engine (SGE) and Portable Batch System (PBS) queuing system on Cortex were made available [1, 3]. These systems would allow parallel jobs to be queued up one after another and distributed to available nodes in the cluster without a user specifying them. The launch command was updated so that it would use the queue instead of starting the simulation itself. With the queue, the node file specifying which CPUs would no longer be needed, so that option was removed. Although the Sun Grid Engine included a graphical interface, it expected users to write a shell script to be submitted to the queue. This was rather absurd since writing the shell script would be a difficult task for our expected user base of medical and psychology students with little UNIX experience. The Single Socket Server created and submitted this script on behalf of the user after getting the same parameters it had before: simulation input file (name or contents), NCS application index, number of CPUs, and where to send standard output and standard error.

A command similar to launch, called “exec,” was created to allow specified script programs to run. The file “server.scripts” was added so that a list of available scripts would be read in at the server’s start up. Each script was assigned an ID based on the order of appearance in the file. The user could then request that a script be started with a parameter set passed to the server. The main goal of this new feature was to allow large simulation input files with repetitive structures to be created locally on the server’s system. A user would send the unique features of the input file along with parameters that dictated how they repeated. The script then took the parameters and created a large and complete input file. This method saved the bandwidth that was consumed sending a very large input file had it been constructed on the user’s machine and then sent to the cluster.

The current version of BCS focuses primarily on the Autoport and Single Socket Server portions. Although the original VOServer is still available for backwards compatibility, it is not used. Autoport is primarily useful for jobs with a small number of connections (less than 1024), whereas the Single Socket Server is most useful for large jobs that require much interaction with outside clients.

## Chapter 4: Results

With BCS streaming communication back and forth between client programs such as IVO with NCS, experimentation results can be accumulated and analyzed. This chapter first looks at assessing the original goal of replacing the file-based stimulus with the port-based stimulus. Next is an examination of BCS and its role in linking NCS and IVO together. The changes made to NCS so that it could communicate with the server are outlined. The development of IVO in working with the brain simulator is also covered.

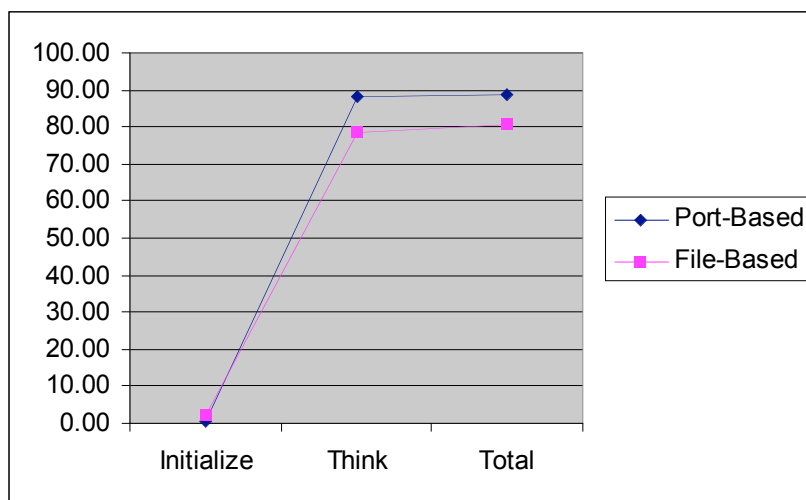
### 4.1 Server Metrics

In interpreting the effectiveness of BCS, a comparison with the file-based stimulus generation is made. This is not a definitive means of evaluating the performance of BCS, since subtle nuances need to be accounted for. Figure 4.1 shows timing results for two brain models run on four 1.0 GHz Pentium III CPUs. The first model has seven columns, each with five layers. Each layer has a total of twenty excitatory cells and six inhibitory cells. The layers are named L2, L3, L4, L5, and L6. Stimulus is injected into L4 every 2.5 milliseconds. The simulation lasts for three seconds, with each second divided into 10,000 time steps (30,000 time steps total). The second simulation has the same column and layer setup, but the number of cells are increased by ten-fold. Table 4.1 shows timing results from the two simulations. The timings are broken down into two situations. The initialization time is when the objects that make up a brain are constructed and configured. The thinking time is when actual simulation takes place, stimulus entering the system and action potentials moving through the system. The initialization times show that a file-based system takes longer to start because it reads in the entire stimulus

file at the start. In the first simulation, the thinking time is greater for the port-based stimulus since it incurs some overhead from the data transfer happening throughout the simulation. For the larger simulation, the thinking times fluctuate enough that after some instances, the port-based simulation finishes faster, other instances have the file-based finish faster. The resulting average ultimately ended in favor of port-based, but that is not necessarily always the case. Figure 4.1 shows a graph of the timing results of the first simulation to illustrate the differences.

**Table 4.1** Average run times for a small and large simulation

	Initialization	Thinking	Total
<b>Simulation 1</b>			
Port-Based	0.32	88.32	88.64
File-Based	2.04	78.36	80.41
<b>Simulation 2</b>			
Port-Based	2.45	2007.60	2010.05
File-Based	4.21	2011.08	2015.29



**Figure 4.1** Timing differences of a small simulation

There are several things to take note of when analyzing the table and graph. When using a file-based system, NCS must terminate and restart in order to take on new stimulus. Meanwhile, the port-based input can be fed continuously to a simulation. This means that for finer control of

the simulation, a file-based system would need to shut down more often and be restarted. This can be seen in experiments done with CARL, where new data collected would have no effect until the next start and stop cycle. Using the port-based system means no delay in interpreting new information.

## 4.2 Interaction of NCS and IVO

### 4.2.1 Expansion of NCS Input File

As BCS evolved, the input file that was used to construct an NCS simulation adapted to allow a user access and control over the new features. Table 4.2 shows keywords added to NCS for use with VOserver. These keywords formed the basis for future expansions so that as new modifications were made, users would not need to add new keywords to the NCS vocabulary. Rather, the values given to the keywords changed to reflect new capabilities of the latest server program.

**Table 4.2** New and Updated keywords of NCS for VOserver

<b>Object</b>	<b>Keyword</b>	<b>Value</b>	<b>Purpose</b>
Brain	Port	Integer	Specify the port where commands for NCS can be read
	Server	Hostname	Specify the IP address of Voserver
Report	Port	Integer	Specify the port where this report sends data
	Filename	Hostname	Specify the IP address of Voserver
Stimulus	Port	Integer	Specify the port where this stimulus reads data
	Filename	Hostname	Specify the IP address of Voserver

With the VOserver, the user needed to specify only a few parameters. The Hostname was required to be specified in any object that used the server, but never changed. Port values could range in value, but as described in Chapter 3, became difficult to configure properly. Table 4.3

shows the changes made to the NCS input file to accommodate the Autoport server. The primary brain object was expanded to contain the master server address and port value. The normal Port keyword could now accept the value AUTO so that it would know to use the Autoport's automatic configuration capabilities. The Job and Type fields that already served purposes in the original NCS input file had their duties expanded so that they would make report and stimulus streams have descriptive names.

**Table 4.3** New and updated keywords for NCS for Autoport

<b>Object</b>	<b>Keyword</b>	<b>Value</b>	<b>Purpose</b>
Brain	Port	AUTO	Request that Autoport server manage the Brain's command stream.
	Server	Hostname	Specify the IP address of Autoport server
	Server_Port	Integer	Specify the port of Autoport server for all objects (Brain, Reports, Stimulus).
	Job	Name	Descriptive name to attach to streams
	Type	Name	Descriptive name to attach to streams
Report	Port	AUTO	Request that Autoport server manage this report's data
	Filename	Report Name	The name sent to Autoport for reference of this report
Stimulus	Port	AUTO	Request that Autoport server manage this stimulus's data stream.
	Filename	Stimulus Name	The name sent to Autoport for reference of this stimulus

With the development of the Single Socket server, only minor refinement needed to be made to the NCS input file. Table 4.4 shows the final revision to the input file such that the Port keywords were again updated to allow for the value SINGLE to be given by the user. This

would dictate that a common socket be created by the brain and shared among any sending or receiving objects.

**Table 4.4** New and updated keywords of NCS for Single Socket Server

<b>Object</b>	<b>Keyword</b>	<b>Value</b>	<b>Purpose</b>
Brain	Port	SINGLE	Request the Brain's command stream use the single socket.
	Server	Hostname	Specify the IP address of Single Socket server
	Server_Port	Integer	Specify the port of Single Socket server for all objects (Brain, Reports, Stimulus). Also used for Autoport connections.
	Job	Name	Descriptive name to attach to streams
	Type	Name	Descriptive name to attach to streams
Report	Port	SINGLE	Request that Single Socket server manage this report's data stream.
	Filename	Report Name	The name sent to Single Socket server for reference of this report
Stimulus	Port	SINGLE	Request that Single Socket server manage this stimulus's data stream.
	Filename	Stimulus Name	The name sent to Single Socket server for reference of this stimulus

#### 4.2.2 Runtime Commands for NCS

As experimentation occurred with the IVO, greater control was needed over the simulation. NCS was extended to allow it to accept certain commands which affected how the brain responded and behaved as it received stimulus and engaged in neural network activity.

The ability to activate and shut off long-term learning for specific synapses allowed greater observation into the impact of Hebbian learning on the virtual organism. Another command allowed a predefined stimulus object to be injected at arbitrary times. Normally, stimulus was applied at predefined times, using regular intervals. By allowing stimulus to be applied only

under certain circumstances, the effect of reward and punishment on specified regions of cells could be created. Corresponding to the ability of arbitrary stimulus was reporting at arbitrary times. This allowed the user to look at cells more closely when the simulation's activity picked up and seemed interesting, but keep the reports small and manageable if certain regions of cells did not need to be viewed. A Save command was created so that as experiments showed promising results, the state of the brain could be preserved and reused under different conditions. Finally, when the simulation fulfilled some goal, an Exit command allowed NCS to shut down and close open connections in a stable manner. Table 4.5 outlines these commands and the parameters passed into NCS so that it could execute them.

**Table 4.5** Commands that may be sent to NCS during runtime

Command	Parameters	Description
SetHebbian	<synapse name> <setting>	Specify the name of the Synapse to change. The settings are: NONE, BOTH, +Hebbian, -Hebbian.
AppendStim	[flags] <stimulus name> <TimeStart> [TimeStop]	Specify any flags: a = automatic TimeStop after 2.5 ms, s = times specified in seconds, t = times specified in timesteps. The Stimulus name. The new start time. The new stop time (if not automatic).
AppendReport	[flags] <report name> <TimeStart> [TimeStop]	Specify any flags: a = automatic TimeStop after 2.5 ms, s = times specified in seconds, t = times specified in timesteps. The Report name. The new start time. The new stop time (if not automatic).
Save	<filename>	Specify a filename to store brain state.
Exit		Program stops simulation and exits

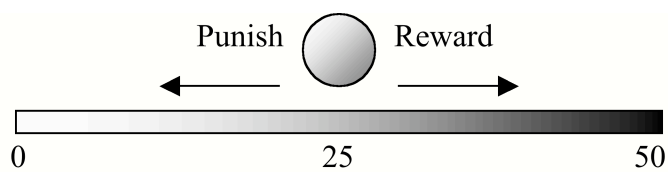
#### 4.2.3 Demonstration of IVO

The IVO was developed using Matlab with the aid of the TCP/UDP/IP Toolbox 2.0.5 [31]. This library allowed for TCP/IP communication with BCS using provided function calls to



establish connections as well as send and receive data.

The goal of the IVO was to move along a linear gradient during training. Figure 4.2 shows how the IVO was positioned in the center of this gradient and through movement with corresponding reward and punishment, the IVO was to learn to move toward the side of greater intensity. The gradient was calculated as a function along the x-axis and used as the stimulus that was sent to NCS.

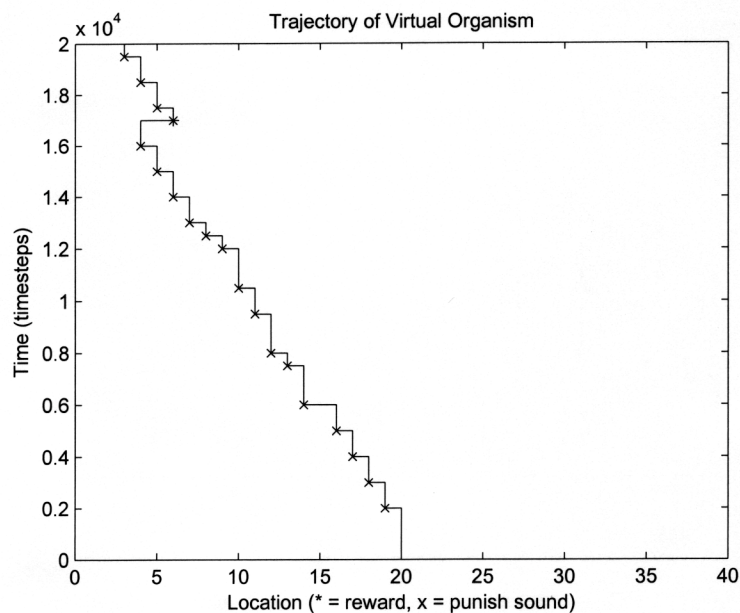


**Figure 4.2** The IVO needed to move from the center towards the right

The earliest IVO experiments attempted to interpret NCS reports received through BCS in a variety of ways to determine what motor response to make depending on the output for a series of time steps called the “motor interval” from selected cells. For example, by looking at the number of action potentials fired in a report, a response of “left”, “right”, or “stand still” could be interpreted. Another method looked at the average voltage to determine appropriate action. Figure 4.3 shows that initial attempts to get IVO to behave in the expected way were not successful. In the figure, the x-axis shows the IVO's current position with respect to time elapsed in the simulation (y-axis). Periodically, the IVO would not use the calculated motor response, but instead would 'explore' such that it would move randomly one way or the other. This would ensure that it did not stand still for the entire simulation as shown in Figure 4.4 of an early IVO run.

As the IVO moved along the line, commands were sent to NCS to control aspects of the

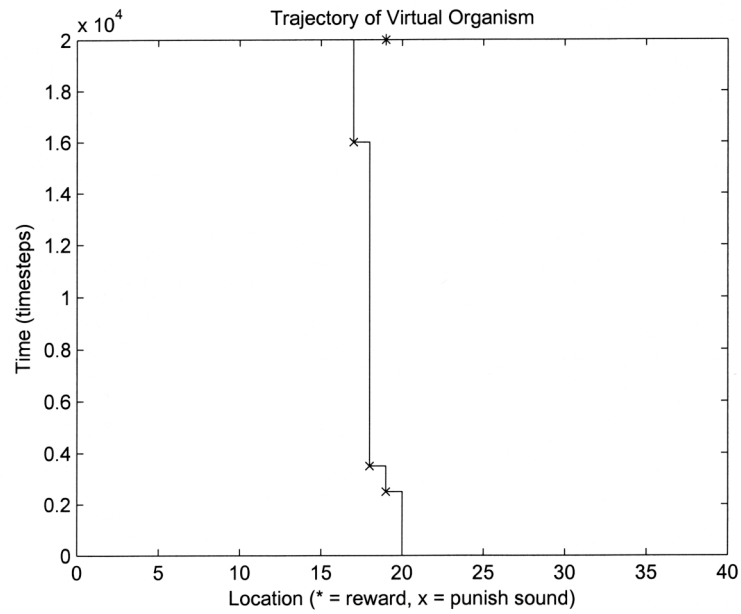
simulated brain. When punishments and rewards were to be used, the Hebbian learning of synapses could be turned on. Then, based on the IVO's action, a command to initiate the rewarding or punishing stimulus was sent to NCS. A reward occurred when the IVO moved in the correct direction and involved a current injection that depolarized a group of cells, increasing their membrane voltage so that it would be easier to reach threshold. A punishment occurred when the IVO moved in the wrong direction and involved a current to hyperpolarize a group of cells, lowering their membrane voltage and suppressing attempts to reach threshold.



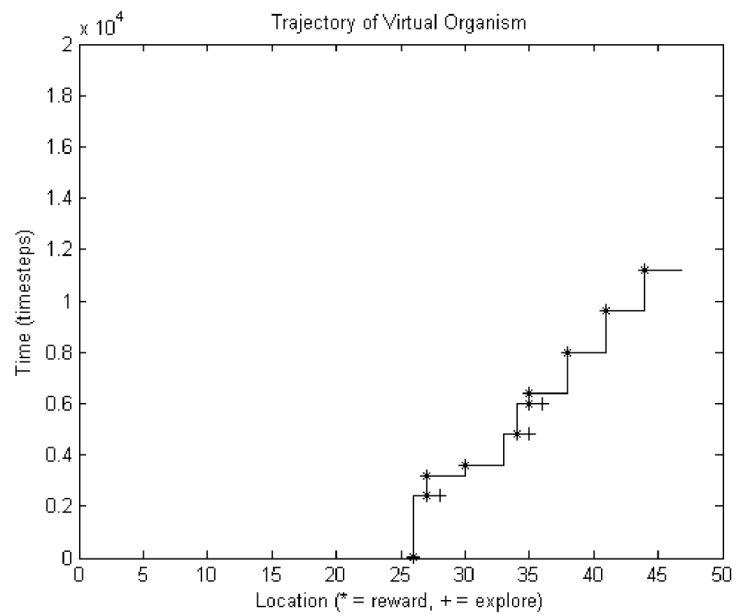
**Figure 4.3** Initial IVO experiments worked with interpreting reports

As the IVO experiments progressed, the stimulus and report data exchanged was better interpreted, and the synaptic learning accumulated as brain states were saved and restored. Figure 4.5 shows that the later experiments successfully had the IVO move toward the goal as the stimulus generated from the IVO's current position on the gradient induces a motor response

of right movement from NCS.



**Figure 4.4** Without an explore function, the IVO might not move



**Figure 4.5** As the simulations was fine-tuned, the IVO behaved as desired

# Chapter 5: Conclusions and Future Work

## 5.1 Contribution

With the current incarnation of BCS, communication between NCS and another client can occur dynamically. No longer does a mass of stimulus need to be collected before it can be sent as one large clump to NCS. This makes NCS more responsive since it can react to events as they happen instead of giving a motor response for events in the past. Motor responses carried out from older information could have become invalid during the delay. Also, NCS does not need to shut down in preparation for the arrival of stimulus, only to rebuild the same structures it had created moments before. Needing to redo initializations repeatedly wastes valuable time. Under the file-based system, the duration of the simulation could be shortened to try and keep stimulus from becoming too old, but this would only exacerbate the accumulating delay caused from initialization.

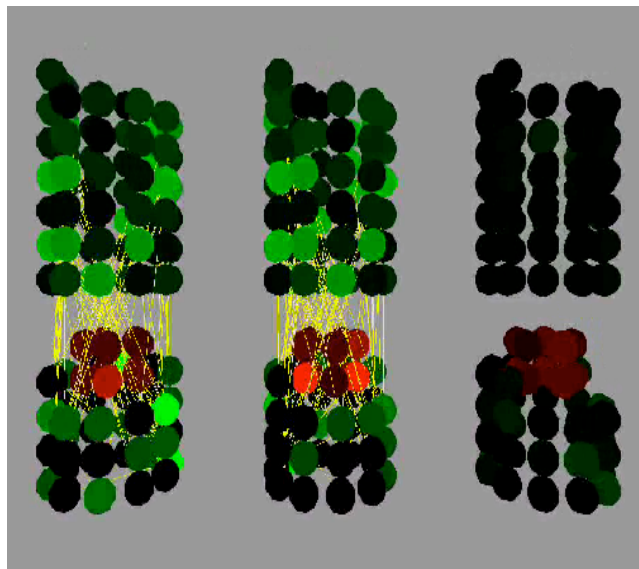
BCS has been designed specifically for NCS. The overhead associated with transferring data has been limited. This will help to prevent communication from becoming a major bottleneck and slow the simulation. Methods for connecting to the server have been implemented in Matlab and C/C++ so that they can easily be reused in applications beyond IVO. Additionally, the code structure and functions developed have been documented for extensibility if the needs of NCS change.

Although there is still much research to be done with IVO, the existence of a server prepares for the development of new applications that could aid in making NCS better, or in making NCS more usable in other industries.

## 5.2 Potential Applications

### 5.2.1 Visualization

With extensions made to NCS that allow it to assign realistic coordinates to neurons and substructures, software able to better visualize the simulation can be constructed. These visualization tools could be designed to work with the server to allow a user to have greater control during a simulation. Figure 5.1 shows an image from an older attempt at designing a visualization program. Although it allowed the user to see the events happening within a brain simulation, the viewing occurred after the simulation had been completed and closed. With the ability to control the simulation through BCS, a user making observations could rapidly alter connections or internal dynamics in order to shift the simulation's performance in one direction or another.

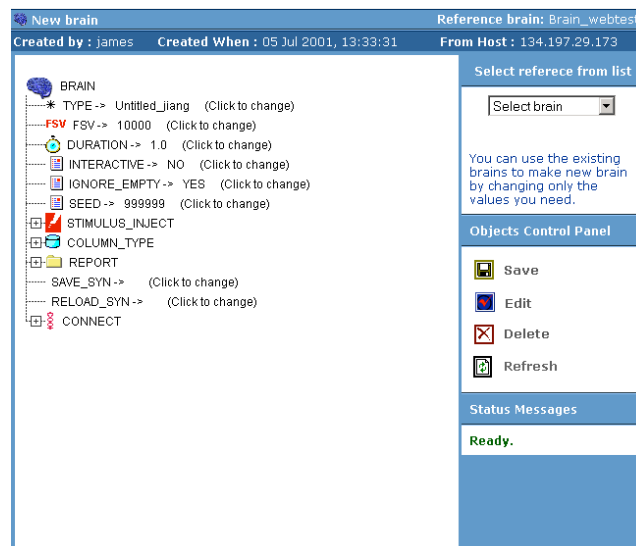


**Figure 5.1** Screenshot from early NCS visualization software

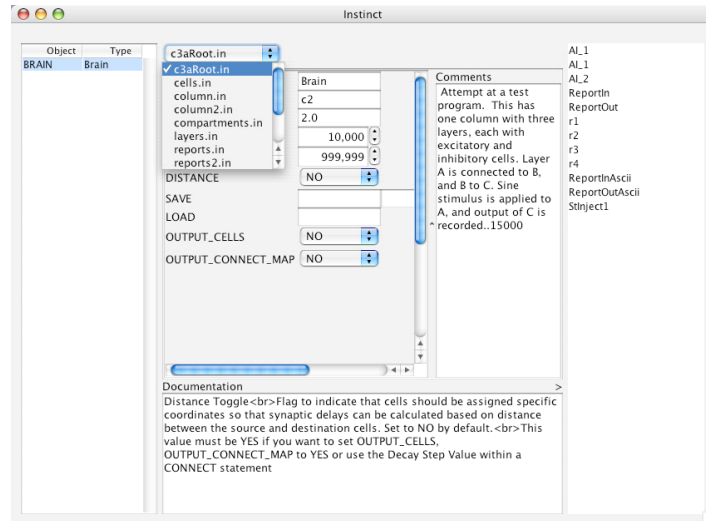
## 5.2.2 NCS Tutorial Program

As a simulation tool, NCS is intended for use by people from various fields such as medicine and psychology. As mentioned in previous sections of this thesis, students have already used NCS for research projects. However, some of these students had never been exposed to a UNIX environment and faced a difficult learning curve not only in understanding the usage of NCS, but also how to work in the text-based operating environment.

Some work has been done in the past to try and develop a more intuitive software system to help users with NCS through both web-based systems [35] and java-based systems [21]. Figures 5.2 and 5.3 show screenshots of these applications. By integrating communication with BCS into these projects, users will have a more interactive experience while learning NCS to help speed their familiarity with the simulator. Access to the cluster's queuing software will be easier when used through a graphical user interface. In addition, these applications help to ensure that the simulation objects are created correctly without dealing with minor syntax issues. Users can then start the simulation and collect the reports as they are written, or request them later.



**Figure 5.2** A web based interface to create NCS input files



**Figure 5.3** Instinct: a java-based tutorial program for NCS

# References

- [1] *Altair PBS Professional*. [last accessed on April 4, 2005]; Available from: <http://www.altair.com/software/pbspro.htm>.
- [2] *Apache HTTP Server project*. [last accessed on April 4, 2005]; Available from: <http://httpd.apache.org/>.
- [3] *Grid Engine sources and documentation*. [last accessed on April 4, 2005]; Available from: <http://gridengine.sunsource.net>.
- [4] *Myricom Inc. Creators of Myrinet*. [last accessed on April 4, 2005]; Available from: <http://www.myrinet.com>.
- [5] *Squid Web Proxy Cache*. [last accessed on April 4, 2005]; Available from: <http://www.squid-cache.org/>.
- [6] N. Brenner, S. P. Strong, R. Koberle, W. Bialek, and R. R. de Ruyter van Steveninck, "Synergy in a Neural Code," *Neural Computation*, vol. 12, pp. 1531-1552, 2000.
- [7] R. S. Cajal, *Histology Due Systeme Nerveux de l'Homme et des Vertebres*, vol. 2. Paris, 1911.
- [8] J. Campos, S. Donatelli, and M. Silva, "Structured solution of asynchronously communicating stochastic modules," *IEEE Trans. on Soft. Engr.*, vol. 25, pp. 147-165, March-April 1999.
- [9] H. J. Chiel and R. D. Beer, "The brain has a body: Adaptive behavior emerges from interactions of nervous system, body and environment," *Trends in Neurosciences*, 1997.
- [10] B. W. Connors and M. J. Gutnick, "Intrinsic firing patterns of diverse neocortical neurons.," *Trends Neurosci*, vol. 13, pp. 99-104, 1990.
- [11] J. Frye. Parallel Optimization of a NeoCortical Simulation Program [Master's Thesis]. University of Nevada: Reno, NV. December 2003.
- [12] J. Frye, J. G. King, J. C. Wilson, and F. C. Harris Jr., "QQ: Nanoscale Timing and Profiling," in the *Proceeding of the 19th IEEE International Parallel & Distributed Processing Symposium*, 2005.
- [13] P. Goodman and F. C. Harris Jr., "Durip04: Parallel Beowulf Computing/Brain/Robotics, Phase III," Research Proposal to ONR. Submitted 2004.
- [14] P. Goodman and F. C. Harris Jr., "Parallel Beowulf Brain-Robotics Simulation," Research Proposal to ONR. Submitted 2001.



- [15] P. Goodman, S. Louis, and H. Markram, "Parallel Beowulf Brain Simulation," Research Proposal to ONR. Submitted 1999.
- [16] W. Y. Gupta A., and Markram H., "Organizing Principles for a Diversity of GABAergic interneurons and Synapses in the Neocortex.," *Science*, vol. 287, pp. 273-8, 2000.
- [17] I. K. Helmchen F., and Sakmann B., "Ca<sup>2+</sup> buffering and action potential-evoked Ca<sup>2+</sup> signaling in dendrites of pyramidal neurons," *Biophys J*, vol. 70, pp. 1069-81, 1996.
- [18] M. J. C. Hoffman D.A., Colbert C.M., "K<sup>+</sup> channel regulation of signal propagation in dendrites of hippocampal pyramidal neurons," *Nature*, vol. 387, pp. 869-75, 1997.
- [19] M. Inaba, S. Kagami, F. Kanehiro, Y. Hoshino, and H. Inoue, "A platform for robotics research based on the remote-brained robot approach," *The International Journal of Robotics Research*, vol. 19, pp. 933-954, October 2000.
- [20] J. Kang, J. R. Huguenard, and A. A. Prince, "Voltage-gated potassium channels activated during action potentials in layer V neocortical pyramidal neurons.," *J Neurophysiol*, vol. 83, pp. 70-80, 2000.
- [21] J. G. King, "Instinct: A Learning Tool for New Users of the NeoCortical Simulator." Technical Paper. University of Nevada, Reno. 2004.
- [22] C. Koch, *Biophysics of Computation*. New York, NY: Oxford Univ. Pres, 1999.
- [23] C. Koch and I. Segev, *Methods of Neuronal Modeling*, 2nd ed. Cambridge, MA: MIT Press, 1998.
- [24] J. C. Macera. Design and implementation of a hierarchical robotic systems; A platform for artificial intelligence investigation. [Master's Thesis]. University of Nevada: Reno. 2003.
- [25] J. C. Macera, P. H. Goodman, F. C. Harris Jr., R. Drewes, and J. B. Maciokas, "Remote-neocortex control of robotic search and threat identification," *Robotics and Autonomous Systems*, vol. 46, pp. 97-110, 2004.
- [26] J. Maciokas, P. Goodman, and F. Harris, "Large-scale spike-timing-dependant-plasticity model of bimodal (audio-visual) processing." Technical Paper. Brain Computation Lab, University of Nevada, Reno, NV. 2002.
- [27] J. B. Maciokas, P. H. Goodman, and J. L. Kenyon, "Accurate Dynamical Model of Interneuronal GABAergic Channel Physiologies." Technical Paper. University of Nevada, Reno. 2004.
- [28] B. Opitz, "A Balanced Knock-Out Computer Model of Neuronal Ca<sup>++</sup>-Dependant K<sup>+</sup> Channels." Technical Paper. University of Nevada, Reno. 2004.

- [29] D. Purves, G. J. Augustine, D. Ditzpatrick, L. C. Katz, A.-S. LaMantia, and J. O. McNamera, "Neuroscience." Sutherland, MA: Sinauer Associates, Inc., 1997.
- [30] M. C. Ripplinger, C. J. Wilson, J. G. King, J. Frye, R. Drewes, F. C. Harris, and P. H. Goodman, "Computational model of interacting brain networks," *Journal Of Investigative Medicine*, vol. 52, pp. S155-S155, 2004.
- [31] P. Rydesäter. *TCP/UDP/IP Toolbox 2.0.5*. [last accessed on April 2005]; Available from: <http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=345&objectType=File>.
- [32] W. R. Stevens, *Advanced Programming in the UNIX Environment*, 1st ed: Addison-Wesley Professional, 1992.
- [33] P. S. Strong and R. Koberle, "Entropy and Information in Neural Spike Trains," *Physical Reveiw Letters*, vol. 80, pp. 197-200, 1998.
- [34] M. V. Tsodyks and H. Markram, "The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability," *Proc. Natl. Acad. Sci.*, vol. 94, pp. 719-723, January 1997.
- [35] K. Waikul, L. Jiang, E. C. Wilson, F. C. Harris Jr., and P. Goodman, "Design and Implementation of a Web Portal for a NeoCortical Simulator," in the *Proceedings of the 17th International Conference on Computers and Their Applications*, April 2002.
- [36] E. C. Wilson. Parallel implementation of a large-scale biologically realistic parallel neocortical-neural network simulator [Master's Thesis]. University of Nevada: Reno. 2001.
- [37] E. C. Wilson, P. H. Goodman, and F. C. Harris Jr., "Implementation of a Biologically Realistic Parallel Neocortical-Neural Network Simulator," in the *Proc. of the 10th SIAM conference on Parallel Process for Sci. Comput.*, March 2001.
- [38] E. C. Wilson, P. H. Goodman, and F. C. Harris Jr., "A Large-Scale Biologically Realistic Cortical Simulator," in the *Proceedings of SC 2001*, Denver, Colorado, 2001.
- [39] K. C. Yamada W.M., and Adams P.R., "Multiple channels and calcium dynamics," in *Methods of Neuronal Modeling: From Ions to Networks*, C. K. a. I. Segev, Ed. Cambridge London: The MIT Press, 1998, pp. 137-170.