Computational Intelligence in Rate-Based Control for Data Networking

by

C. Hart Poskar

A Thesis

Submitted to the Faculty of Graduate Studies

In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg, Canada, 2002

0-612-79882-8

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES
*****
COPYRIGHT PERMISSION PAGE

COMPUTATIONAL INTELLIGENCE IN RATE-BASED CONTROL FOR DATA NETWORKING

BY

C. HART POSKAR

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University

of Manitoba in partial fulfillment of the requirements of the degree

of

Doctor of Philosophy

C. HART POSKAR © 2002

Abstract

The work presented in this thesis developed a negative acknowledgment based transport layer protocol called the versatile transport protocol (VTP). In addition to this a fuzzy neural network based fuzzy controller for the VTP was evaluated. The controller was developed by extending the Brain Construction Kit neural network simulation software to support fuzzy neural networks. The VTP controller was then used together with a client/server bulk transfer software called Phat (developed at the University of Manitoba), which acted as the emulation environment for VTP.

To implement a generic multi-layered feed-forward fuzzy neural network, the generalized delta rule for back-propagation was extended to include both logic-based as well as arithmetic neural descriptions. To implement training of the network it was also necessary to develop a fuzzy back-propagation algorithm to guarantee the correctness of the connection strength parameters for logic-based neurons. In doing this a different view of the learning rate was required, one that incorporates both a network independent as well as a network sensitive part.

This thesis represents one of the first attempts to use a fuzzy neural network to control or broker bandwidth within a rate-based transport protocol running over IP. As opposed to models based on simulation, a fuzzy neural controller was demonstrated on top of the Phat protocol which was used to emulate a rate-based transport layer protocol.

In addition to contributions made to fuzzy neural network control theory, this work illustrates the viability of fuzzy control as applied to telecommunication applications such as bandwidth brokering within the confines of an IP network. In addition, the study illustrates the role that fuzzy control can play in improving end-to-end performance by adaptation to host and/or network bottlenecks.

Acknowledgements

This thesis, as with all research, is not the sole work of a single author, but the results from collaboration and interaction with many people. I would like to take this opportunity to thank all of the people who have contributed towards this thesis.

I would like to begin by acknowledging the work that led up to this thesis, the development of the FSTP protocol and the Phat client server, by thanking my advisor Bob McLeod, the Internet Innovation Center, and specifically Shawn Silverman.

I would also like to thank the members of my thesis committee, Dr. Witold Pedrycz, Dr. Peter Graham, Dr. Bruce Cockburn, Dr. David Blight and Dr. Ekram Hossain, for taking the time to be on my committee, and for having read my thesis. In particular I would like to thank Drs. Graham and Cockburn for their feedback and recommendations for improving the research, and Dr. Pedrycz for his early impetus into fuzzy neural hardware.

In addition I would like to thank all of my collaborators and friends who have been there for me these many years. In particular I would like to express special thanks to:

Peter Czezowski, for your help and insights, and knowing when to call it a night and see a movie,

Fred Corbett, who was instrumental in making my code work, and for putting up with me even at my worst,

Yair Shachar-Hill, for reading virtually every version of my candidacy and most of my thesis, all those re-writes - I learned a lot,

Howard Jesierski, just one more game of cards - one day,

Eileen, Mithal and Mina Patel, and all those mashed potatos, and

Myrna, because we all need a little support.

Finally, none of this could have happened without the long term support from my family. Their encouragement these many years made it possible for me to finish what I started. Thank you.

This thesis is dedicated to

David Nathan Shachar-Hill

Table of Contents

List of Figures

List of Tables

I. Introduction

Data networking today is founded on packet switched networks. Packet switched data networks work by routing packets of encapsulated data based on the current state of each switching node (called routers) and the destination (or optional routing) information contained in the data packet header. Initial network protocols for packet switching compensated for unreliable networks by including large amounts of overhead and redundancy both in the encapsulated data packet and at each routing point. As the underlying data networks became more reliable, it was recognized that the additional overhead was no longer needed. Step-by-step reliability was pushed back to the end points of a data connection (the hosts). The final result is a network protocol that provides an unreliable service, the Internet Protocol (IP) [1], which has become the dominant network protocol in data networks.

Reliability in IP data networks is provided by the transport layer Transmission Control Protocol (TCP) [2] or by the application (using the User Datagram Protocol, UDP [3], at the transport layer). In essence, reliability encompasses two main issues: first the detection of data loss due to missing or corrupt data, and second the ability to recover from data loss.

TCP provides for reliability through window-based management of each data flow (or connection) together with positive acknowledgements of all data received. TCP recognizes that there are two general causes for data loss: the network and the communicating hosts. To combat this two windows are maintained, one representing network congestion (the usual reason for data loss on the network) and one for the host I/O buffers (for flow control).

Together TCP and IP are the major protocols in a suite of protocols that bears their name TCP/IP. The next section will introduce the idea of network protocol stacks which feature a layered approach to network protocol development, together with the OSI reference model and TCP/IP.

1.1 Network Protocol Stacks

The network protocol stack[1] is the foundation of modern communication networks. The basic idea of the stack is to group together functions to perform a common set of tasks into a module or layer. Each layer in the stack has a well defined function and standardized interface, allowing for the encapsulation of layers. This method of design has several advantages, including the ability to replace the implementation of a layer without needing to redesign or re-implement the entire communication process.

The operation of a network stack is illustrated in Figure 1.1, which shows a three-layer protocol stack consisting of an application layer, a transport layer, and a network access layer (the physical layer is implied). As we go down the stack (the application layer is the

1. This may also be referred to as the network stack, or just the stack.

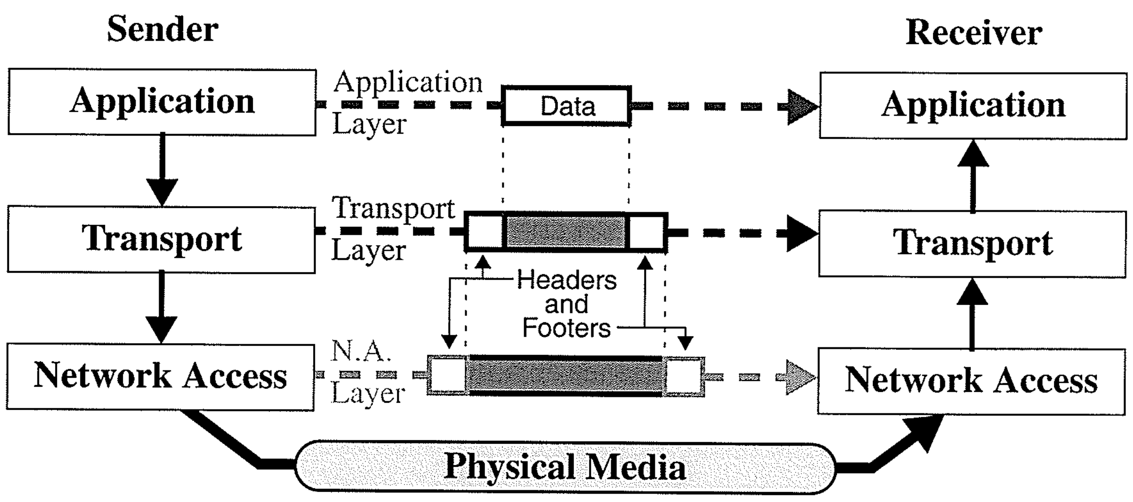


Figure 1.1 : Example of a network protocol stack transaction. The actual data flows from the sender to receiver application following the solid black lines. The dashed lines illustrate the effective flow of information from the perspective of each layer.

top or highest numbered layer, and the Network Access layer the bottom, layer 1 by default) each layer can pass data and parameters regarding the sending of the data. The parameters that are passed between layers are specified by the public layer interface. Each layer then builds a packet which encapsulates the data (as the payload) and optionally adds a header and/or footer containing the actual or calculated parameters passed to it. This packet is then passed down to the next layer which encapsulates the entire packet as its payload[1]. Parameters may also be passed down to be used in the next layer's header or footer, or to be passed down to a still lower layer.When data is passed back up the network stack at the receiver, each layer strips off the header/footer added by the corresponding layer at the sender, and passes the payload up to the next layer, until the original data is delivered to the receiving application.

Although the actual path of the communication is up and down the protocol stacks, the horizontal (dashed) arrows illustrate the apparent communication among corresponding layers in each stack. The information added as header or footer by a specific network protocol layer at the sender is only used by the corresponding layer at the receiver.

For additional information on network protocol architectures or the specific OSI and TCP/IP architectures discussed in the following sections, please see [4] or [5].

1.1.1 The OSI Reference Model

The International Standards Organization (ISO) set out to design a standard network protocol architecture and named it Open Systems Interconnection or OSI. OSI is a seven-layer model which never fully achieved its goal as a practical networking standard, but nevertheless has proven to be an excellent reference model for classifying protocols, and comparing network architectures.

1. Thus the entire higher layer packet becomes payload - including any header, data, or footer.

The seven layers of OSI are:

- ***Application*** - interface to users and distributed information (servers).
- ***Presentation*** - interface for different data representations.
- ***Session*** - establish, manage, and terminate connections between applications.
- ***Transport*** - provides transfer of data between end points.
- ***Network*** - provides control for transmission over networks.
- ***Data Link*** - provides transfer of data (frames) over the physical medium.
- ***Physical*** - provides transmission of unstructured bits over a physical medium.

1.1.2 TCP/IP

While OSI was being designed, an actual suite of protocols was being tested and put into operation on the ARPANET, an experimental packet-switched network and a forerunner of the Internet. These experimental protocols eventually came to define a network architecture called TCP/IP, named for its two main protocols which define the transport and network layers. The TCP/IP architecture uses five layers (the layers being defined after, and based on, the resulting implementation). The five layers of TCP/IP are:

- ***Application*** - interface for user applications.
- ***Transport*** - provides transfer of data between end points.
- ***Network*** - provides control for transmission between different networks.
- ***Network Access*** - interface for a network to the physical layer.
- ***Physical*** - provides transmission of unstructured bits over a physical medium.

In fact the TCP/IP architecture really comprises only the topmost three layers, and never dealt with anything below the Network layer. However the Physical layers are certainly implied and required while the Network Access layer is convenient (although some descriptions leave this out and use a four-layer model). Despite this, TCP/IP implementations have historically been tightly integrated with Ethernet. More recently, IP over other technologies, particularly ATM and SONET, have been progressing steadily.

Figure 1.2 shows roughly how TCP/IP fits against the OSI reference model. All functionality above the transport layer (with the exception of connection-oriented service) is passed up to the application layer. In addition, IP performs some of the functions of the data link layer. However, the ongoing work to provide IP natively over physical layers requires a network access layer to act as an interface.

The next section expands on the transport layer, or layer 4, and the layer 4 protocols that comprise the TCP/IP suite.

1.2 The Transport Layer

The job of the transport layer is to take data from a sending application, and to deliver it to the receiving application (see Figure 1.1). Note that the sending and receiving applications may be running on the same or different hosts.

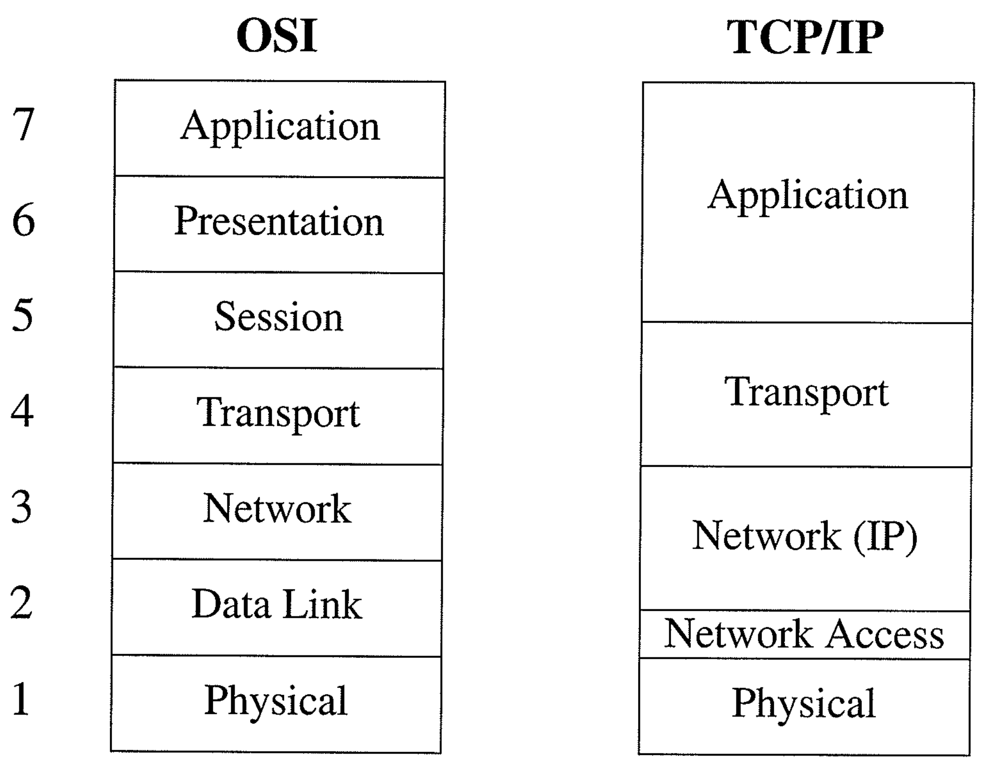


Figure 1.2 : A side by side comparison of TCP/IP to the OSI seven-layer network reference model. In particular the transport layer in TCP/IP may subsume some of the functionality of the OSI Session layer. The remaining functionality of the Presentation and/or Session layers are left to the Application Layer. Similarly, the Network layer provides some of the functionality attributed to the Data Link layer as well as the Network layer.

The simplest transport protocol would provide an interface between applications and the protocol in the layer below, the network (IP) layer in TCP/IP, and would provide a means to identify the source and target application. It would also pass on additional parameters provided by the application, such as the destination host address. At the receiving side the transport protocol would accept data from the network layer, extract the payload and pass it to the application identified in the header (i.e. the target application), along with parameters about the sender of the information (the IP address of the sending host passed as a parameter from the network layer) and the sending application identification.

At the transport layer, the means of identifying an application is called a port number, which is an address that is used to distinguish between different applications that are using the same transport protocol (on the same host) at the same time.

TCP/IP defines two standard transport layer protocols, TCP and UDP. We will look at each of these, starting with UDP.

1.2.1 User Datagram Protocol (UDP)

UDP is a connectionless transport protocol which provides unreliable transport of data between applications as defined in RFC[1] 768 [3]. By unreliable, we mean that there is no mechanism to either guarantee that a transmitted packet is received, or to determine that the destination is unreachable (and hence stop transmitting). It is the job of the application to monitor the situation and take any action, such as requesting feedback from the application with which it is communicating.

As a simple transport protocol, UDP adds very little information to the data payload passed by the application. The format of the UDP header is illustrated in Figure 1.3. Note that beyond the source and destination port numbers, an additional 32 bits of overhead are added for two fields. The first specifies the amount of data encapsulated in the frame (the UDP Length), and the second provides a checksum to verify the correctness of the data before passing it (or not) to the application.

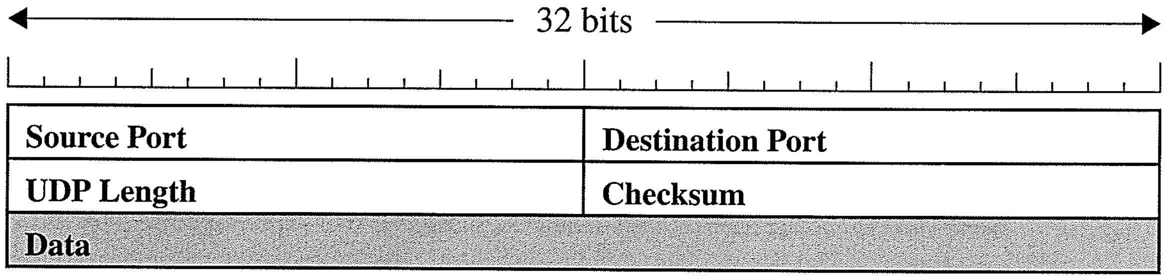


Figure 1.3 : UDP header format. The UDP header contains four 16-bit fields, the source and destination ports (application address), the length of the data (in bytes), and a checksum of the data.

Historically, UDP was intended as a simple way to transport data over reliable networks (i.e. local area networks, or LANs), or where it was more economical to implement minimal reliability in the application, rather than use the greater overhead of a reliable TCP connection (e.g. when transmitting small network messages) [4]. Applications using UDP include DNS [6], RIP [7], DHCP [8], TFTP [9], and NFS [10].

More recently a number of applications for delivering higher bandwidth content over wide area networks (WANs) have used UDP to better deliver real time data where the reliability overhead of TCP may actually be detrimental to achieving acceptable performance.

1.2.2 Transmission Control Protocol (TCP)

TCP is a connection-oriented protocol that provides for the reliable transfer of data between applications as defined in [2]. "Connection-oriented" means that a logical or virtual connection, rather than a physical one, is established.

An application's port number (16 bits) together with a network (IP) address (32 bits) define a unique 48-bit TSAP (TCP Service Access Port). A TCP connection is a logical

1. RFC - Request For Comments

full duplex connection formed between a unique pair of TSAPs.

The basic function of TCP is to provide a reliable connection over an unreliable service. Operations required for reliable transport include[1]:

- establishing and ending connections.
- re-ordering of packets that arrive out of order.
- flow control (to minimize packet loss).
- error detection, and
- the retransmission of lost or damaged packets.

To provide such a service, TCP uses additional header information to establish and monitor the connection (see Figure 1.4). After the port numbers come the sequence and acknowledgement numbers, used to keep track of each byte transmitted and received. This is followed by the header length field (as the header may include optional fields), flag bits to indicate the current status of the connection or to pass on control information regarding the current frame (such as the use of the Urgent Pointer). Then comes the window size, a number that represents how much data can be sent before waiting for an acknowledgement - used for TCP's flow and congestion control.

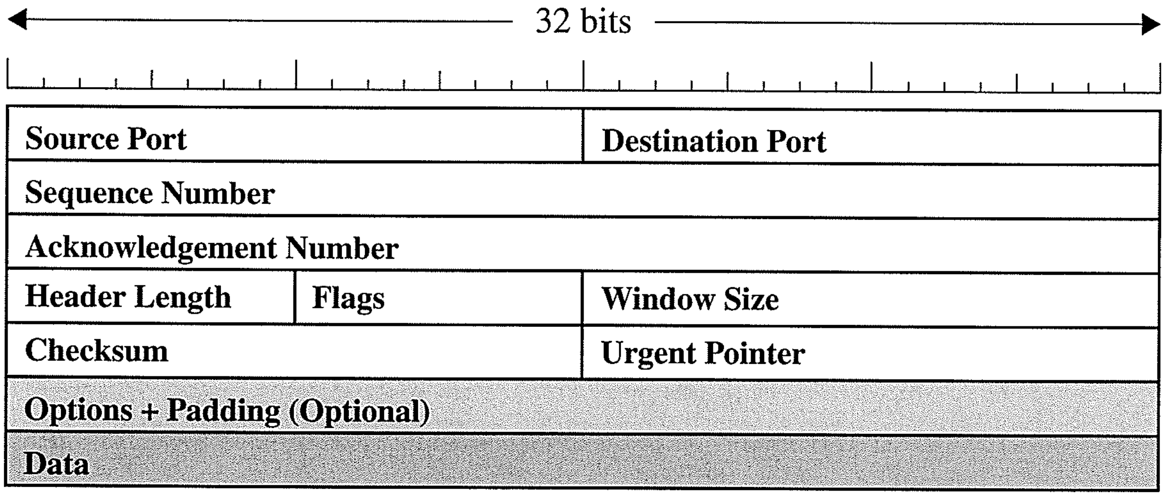


Figure 1.4 : TCP header format. The TCP header contains nine fields plus allows for additional options and padding (to even the word size). These fields are used to provide the reliable connaction oriented service (Sequence, Acknowledgement, window size, flags) and the optional fields require a standard header length field.

To address the strengths and limitations of TCP, its key features - connection-oriented service, reliability, flow control and congestion control - are discussed further in the remainder of this section.

1. Initially security was also included as part of a reliable transport, but was later dropped.

Connection-Oriented Service

TCP connection management requires that there be at least one host which will accept connection requests, and one that can initiate connection requests[1]. The process of establishing a connection is called the "Three-Way Handshake", and is illustrated in Figure 1.5. This uses the sequence (Seq) and acknowledgement (Ack) fields to initialize the flow control, and the two flags **SYN** and **ACK** to request and acknowledge, respectively, the establishment of the connection. This handshake is three-way because it actually sets up two simplex connections (one for each direction) with the intermediate acknowledge and request combined into a single transmission.

When data is sent in either direction, the sequence number indicates the last byte sent. The starting byte is typically chosen by an incrementing algorithm, rather than always starting at zero. This was thought to have added a secure element to an established connection, but this has been shown to offer a false sense of security [11][12][13].

As illustrated in Figure 1.5, TCP requires that the acknowledgement that comes back be actually for the next byte in the sequence that can be sent, i.e. one byte more than the byte sent. If no data is sent the acknowledgement does not need to be updated, except when initiating and terminating a connection.

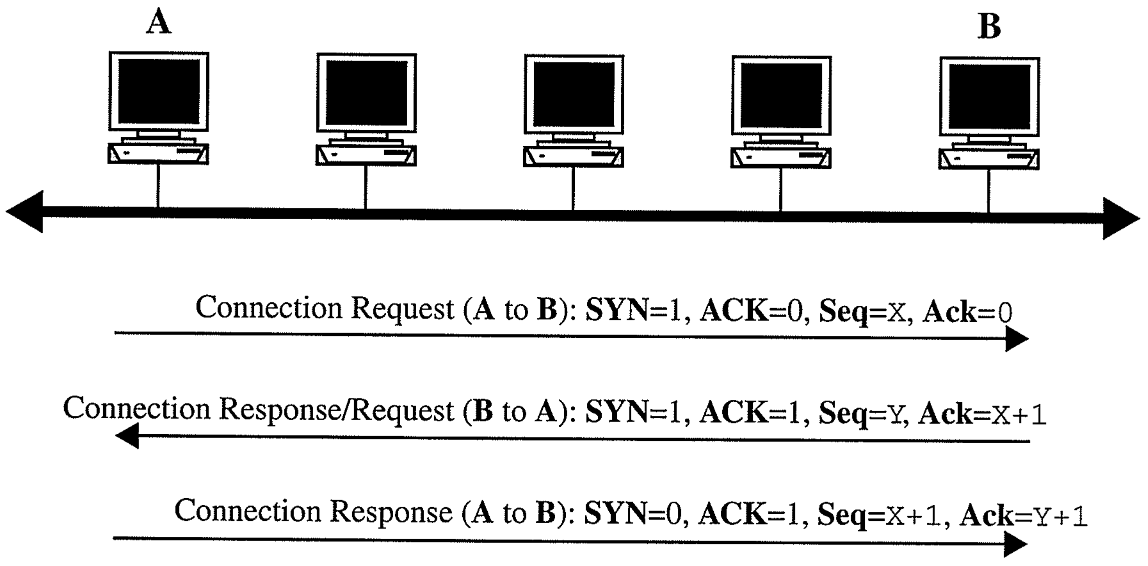

A
B
Connection Request (A to B): SYN=1, ACK=0, Seq=X, Ack=0
Connection Response/Request (B to A): SYN=1, ACK=1, Seq=Y, Ack=X+1
Connection Response (A to B): SYN=0, ACK=1, Seq=X+1, Ack=Y+1

Figure 1.5 : TCP Three Way Handshake

When the connection is no longer needed (all of the data has been transferred), the connection is torn down in a similar fashion. In this case the **FIN** flag is used in conjunction with the **ACK** flag to signal and acknowledge the closing of the connection.

Timers are used when establishing, maintaining and terminating a connection. If one of these connection timers expires without receiving a proper response, then the connection is terminated.

1. These may actually be the same host.

Once a connection has been established, the two main features of TCP connection management are flow control and congestion control. These mechanisms are used to maintain reliability and to provide for retransmission when required.

Reliability

The sequence, acknowledgement, window size, and checksum fields together with a variety of timers are used to implement reliability in TCP. The checksum is used to determine if the packet has been corrupted during transmission, while the sequence and acknowledgement numbers handle out-of-order and duplicate packets as well as allowing for the retransmission of missing packets. The use of the window size is presented in more detail in the following sections on flow and congestion control.

TCP uses positive acknowledgements as the main feature of reliability. Every byte of data transmitted in one direction is identified via a consecutive sequence number. On the receiving side, TCP acknowledges data as the data is delivered to the application. If a sender does not get an acknowledgement after a set amount of time, it retransmits the data. This requires the use of additional timers for each packet sent. There are four main reasons for the sender to fail to receive an acknowledgement:

- the transmitted packet is dropped at a routing node or by the receiver,
- the transmitted packet is received but is corrupted,
- the acknowledgement packet is dropped at a routing node or by the sender, and
- the acknowledgement packet is received but is corrupted.

TCP does not acknowledge packets that arrive out-of-order, which may lead to a fifth cause for packets to be retransmitted: if the sender's timer for those correctly received packets expires before receiving an acknowledgement.

Flow Control

Flow control deals with the reliable end-to-end transmission of packets. Specifically, flow control is responsible for making sure the sender is not flooding the receiver with too much information. If this occurs, then the receiver's incoming buffer will overflow and packets will be lost.

TCP uses a sliding window algorithm together with positive acknowledgment for flow control. The size of the window specifies how much data may be sent before waiting for an acknowledgment from the receiver. The window size (specified in each packet header) is dynamic and may change over the period of a connection.

The flow control mechanism ensures that no packet or sequence of packets exceeds the receiver's buffer capacity. This is done by transmitting the lesser of the receiver's window size and the size of the sender's transmission buffer. The sequence and acknowledgement numbers form the current view of the window, where the acknowledgement number indicates the start of the window, and the sequence number indicates the current position of data that has been sent. As each packet is sent a timer is started. If the timer expires without receiving an acknowledgment, the unacknowledged data is re-transmitted (as described in the previous section).

Congestion Control

While flow control deals with data at the end points of transmission, congestion control deals with data in the intermediate network. Congestion occurs when a router is unable to process packets as quickly as it receives them. Congestion can cause packets to be delayed or discarded.

TCP uses a second window, called the congestion window, to track congestion. The real flow rate is determined by the minimum of this window size and the flow window.

Congestion is detected at the transmitter either through feedback from the intermediate routers or the lack of feedback (acknowledgements) from the receiver. Since the flow window size ensures that the receiver is not being flooded, lost packets (data or acknowledgements) can be attributed to congestion.

The congestion window starts at some small size. Each time the transmission is successfully acknowledged, the congestion buffer is doubled (i.e. the window increases in size exponentially). There is a threshold (initially 64 Kbytes) on the size of the congestion buffer. Once the congestion window size reaches the threshold, the congestion buffer size will only increase linearly. When congestion is detected, the affected data is retransmitted and the congestion window size is set to it's minimum value while the linear growth threshold is reduced by half.

This TCP congestion control mechanism is called slow start because the window size always starts at a minimum size after each congestion detection, and cuts the threshold at which exponential growth becomes linear growth.

1.3 Problems with TCP

TCP provides a reliable connection-oriented service over an unreliable network. However the method used to do this has several drawbacks, in particular given the increased network reliability and bandwidth provided by today's Internet.

Over relatively short distances or low-bandwidth networks, TCP works well because the delay time waiting for an acknowledgement is small compared to either the buffer processing time (short distances) or data transmission time (low-bandwidth).

However where these two factors are increasing, the use of a window-based rate control acts to impose an artificial maximum data rate based on round trip delay time, and not the actual capacity of the hosts and the network.

This latency is due to two metrics of distance. First, physical distance is ultimately constrained by the maximum propagation rate[1]; no amount of infrastructure upgrades or bandwidth management can improve upon this. Second, the number of intermediary nodes (also called hops) that the data must traverse adds a variable amount of processing latency. Networks that suffer from this type of usage are characterized by having a bandwidth delay product of greater than 10^5 bits [14][15]. These networks are referred to in the literature as **high bandwidth*delay product networks**.

1. <= 300,000 Km/s - the speed of light.

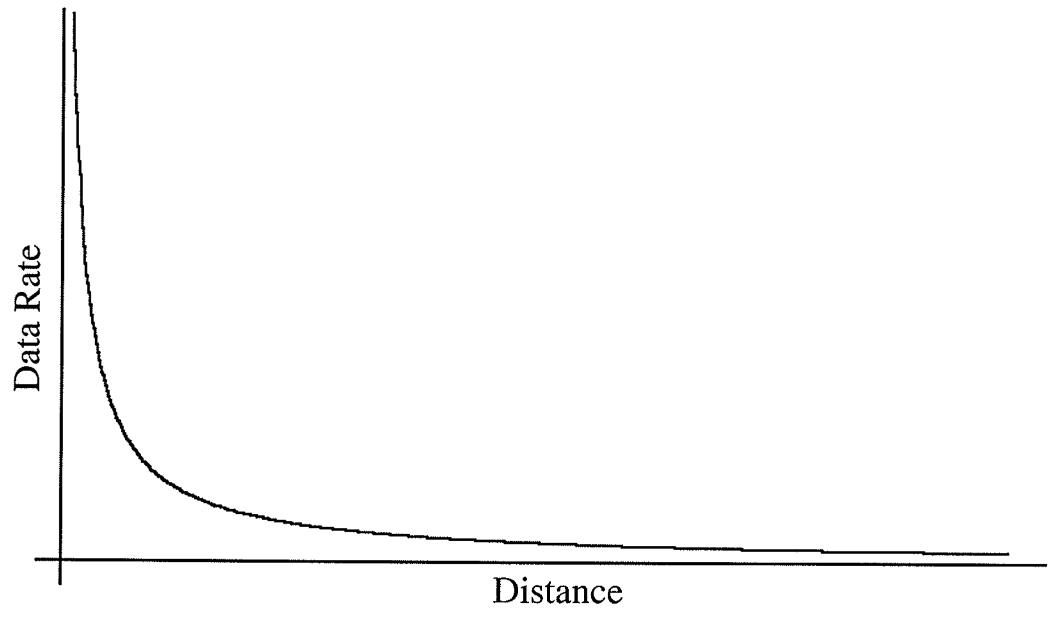


Figure 1.6 : The affect of distance on the maximum theoretical data rate using TCP. The data rate (in a simplified form) is inversely proportional to the physical distance that the data must propogate. The actual scale of the data rate is dependent on the TCP window size which specifies the actual ammount of data that can be transmitted before requiring an acknowledgement.

The affect of propagation latency due to physical distance and the maximum data rate is shown in Figure 1.6. This illustrates the inverse proportional relationship between data rate and distance from the simplified[1] data rate as expressed in:

Data Rate = (Maximum Unacknowledged Data) / 2*(Round Trip Time)

Where the maximum unacknowledged data is dependent on the TCP windows and round trip time is a function of distance and propagation speed.

This limitation in TCP has been known since the earliest implementations [16], and has inspired many potential solutions. These solutions include modifying TCP, and replacing TCP with specialized protocols for high bandwidth*delay product networks [17][18][19]. The most common and successful approach is to use intermediate applications to reduce the required bandwidth (e.g. via data compression [21]), buffer or cache the data at physically closer points [22] and, most recently, to split single TCP flows into multiple smaller parallel flows [20].

A readily available alternative to TCP already exists, the user datagram protocol, which provides a connectionless and unreliable service. Many applications currently use UDP, but mostly in its traditional role [4], where the underlying network is reliable, for real time applications, and for the transfer of very small amounts of data.

1. By simplified it is meant that this does not include transmission, processing or buffering latencies.

Although any application could use UDP instead of TCP, using UDP requires the application developer to implement any or all of the reliability or connection features already present in TCP. This would seem to violate the intention, not to mention the obvious advantages, of designing network protocols in a layered or stack architecture [4].

Several fixes for the limitation imposed by window-based flow and congestion control have also been proposed including modifying the window algorithms [32][33][34][39][40], adding extensions to TCP [26][29][30][41][44][45] and/or optimizing TCP or TCP parameters for the underlying network [36][38][42][43], as well as replacing TCP altogether [23]. In general, improvements to the window algorithm can increase throughput, but don't affect the primary limitation (the window size). Other extensions have been met with criticism [27][28].

The following section will introduce another proposed solution, the Versatile Transport Protocol, which replaces window-based control with an active rate based control using computational intelligence to determine when and how much to throttle back or forward the flow of data based on individual flow characteristics.

1.4 Versatile Transport Protocol

A new transport layer protocol, the Versatile Transport Protocol (or VTP), is developed as a part of this thesis. This new protocol recognizes the increasing reliability of even wide area data networks together with the increasing availability of high bandwidth access, and has a goal to maximize the utilization of available bandwidth. In addition, as the name implies, VTP strives to be versatile while providing a range of service classes, including the reliable transport of data, through different combinations of specific service features.

To accomplish these goals VTP makes use of negative acknowledgments[1] together with a global bandwidth broker for managing data flows. Eliminating positive acknowledgements yields a second benefit - reduced processing overhead. While waiting for positive acknowledgements, each transmitted packet is associated with a corresponding timer. The use of negative acknowledgements can greatly reduce the number of required timers. By designing a simpler transport protocol, and one that can be more easily optimized for both software and hardware implementations, it is hoped that VTP can alleviate the second drawback to TCP, namely, the processing bottleneck in the network stack. In the case of short distance networks (LANS/MANS) it is this processing and, in particular, the clearing of buffers that often limits the overall data rate.

VTP is described in greater detail in Chapter 2 which focusses on rate-based control. The next section introduces the area of computational intelligence, which will be used in implementing the rate-based control mechanisms described in this thesis.

1. A negative acknowledgement is sent to indicate that data was not properly received, rather than a positive one that indicates it has been properly received.

1.5 Computational Intelligence

The area of Computational Intelligence (CI) has gained much acceptance in recent years, and yielded many techniques and algorithms. These techniques and algorithms can be characterized into three categories: Genetic Algorithms (also called evolutionary computing), Fuzzy Systems (granular computing), and Artificial Neural Networks (ANNs). In fact these have been described as "... the three dominant information technologies ..." forming a "Triumvirate of Information Technologies" [48]. Figure 1.7 illustrates how these categories of CI can compliment rather than compete with one another.

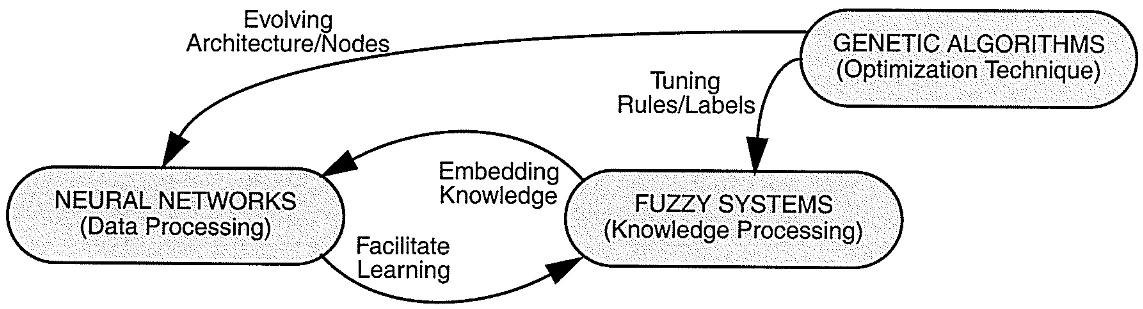

Evolving Architecture/Nodes
GENETIC ALGORITHMS (Optimization Technique)
Tuning Rules/Labels
Embedding Knowledge
NEURAL NETWORKS (Data Processing)
FUZZY SYSTEMS (Knowledge Processing)
Facilitate Learning

Figure 1.7 : Relationships of the three categories of CI techniques.[1]

Artificial Neural Networks or ANNs are based on a model of human neural processes and use large numbers of simple computational elements (called neurons - see Figure 1.8) in a highly interconnected fashion [59]. Each interconnection is represented by a parameter or weight that can be altered to increase or decrease the degree to which an input influences the output of each neuron. The change in the weights is accomplished through learning. Learning typically occurs either through the repeated exposure to training sets of input and output data, or during the processing of real data. ANNs are typically classified by their structure and the learning algorithm - the mechanism for implementing learning.

The structure of a neuron, illustrated in Figure 1.8, consists of two basic processes: aggregation of inputs and a non-linear transformation that produces an output. The first process combines together all of the inputs weighted by their respective interconnection strengths (w_i). This aggregated value is then mapped into a continuous bounded domain, typically [0,1] or [-1,1], through a transformation function(s).

The power of a highly-connected network of these simple structures is in their distributed processing which is used to develop or learn related data patterns stored in the weight space. However there are a few drawbacks to traditional ANN's. First, while the computational elements are typically simple in an analog sense, they can be quite complex to implement using traditional digital hardware and often require large amounts of computing resources in software. A second disadvantage is that there is no known relationship that can be used to map an arbitrary problem into a specific neural structure.This makes the design of traditional ANNs arbitrary and often complex, and heavily relies on the representation of the input data.

1. This figure is reprinted from [49] with permission.

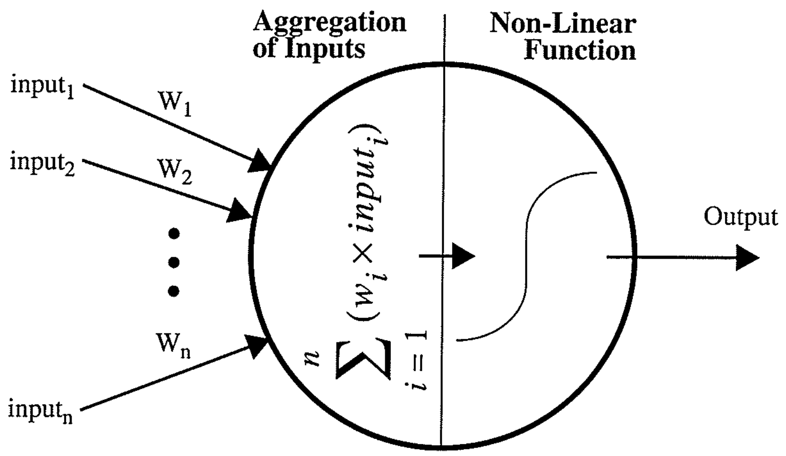


Figure 1.8 : Structure of a neuron - the ANN processing element. Each input is multiplied by a weighting factor. The sum of the weighted inputs is taken, and passed through a non-linear mapping to provide a bounded output value. One neuron's output can be an input to another neuron, or even to itself.

Fuzzy systems (FS) are based on fuzzy logic [52] which is a combination of traditional two-valued logic and set theory [53][54][55]. In traditional logic and set theory, a variable can either belong to a set or not belong to a set, i.e. it can have two values. Fuzzy logic employs multi-valued variables that describe the degree of belonging to a set which is expressed in a membership function. This degree of belonging can take any value between the two extremes of belonging and not belonging. Fuzzy systems are then easily specified linguistically, where each membership function is given a label, and each fuzzy variable can be combined with these labels in simple if-then rule sets using fuzzy logic operators which are analogous to the familiar logic operators. For example, a simple thermostat controller might have a fuzzy input variable representing temperature (called `Temp`), and a fuzzy output variable representing the control reaction (`Control`). These fuzzy variables might have a membership function labelled "Hot" (for `Temp`) and "Large" (for `Control`). One rule in the control system might then be:

```
if Temp is Hot then Control is Large
```

Fuzzy systems derive their strength from this simple linguistic specification, which gives them the ability to quickly and easily map domain knowledge into a structural representation. A further strength is their ease of implementation in digital hardware using simple low precision logical and arithmetic operations.

Genetic algorithms or GAs are used primarily for optimization [60]. One key strength is a feature called mutation, which has the potential to find new, possibly better, minima when traditional learning algorithms become stuck within a specific local minimum. In other words, GAs performs a broad analysis of the given data space, which for example can be used to determine the starting point for a more traditional minimization algorithm, an ANN learning algorithm, or the parameters of membership functions with respect to the universe of discourse of a specific fuzzy system [48].

The transport bandwidth controller combines the knowledge processing of fuzzy systems with the learning facilities of ANNs, to combine their strengths in the form of Fuzzy Neural Networks (FNNs). The structure of the network will be based on the linguistic labels and rule sets, with learning performed by a fuzzy version of the back-propagations algorithm as proposed in [51] developed in [56] and [57] and completed in this thesis.

Genetic algorithms for parametric optimization will not be incorporated as the initial control design is to be kept as general as possible, and therefore does not need any initial optimization. GAs could be utilized to determine a more optimal starting point, but that is not the focus of this work. For further information on genetic algorithms see [61], on ANNs see [59], and on fuzzy systems see [53].

This will require the specification of a general rule set, the key to which is the identification of pertinent variables, and training data (pairs of input variables, and target outputs).

1.6 Novelty

The primary novelty in this thesis is the specification of a new transport layer protocol, VTP, and the derivation of an extended generalized delta rule for back-propagation which encompasses logic-based as well as arithmetic neurons.

The bandwidth limitations of TCP were well known since the introduction of the protocol, as has been the potential benefits of using negative rather than positive acknowledgement. VTP encompasses many of the ideas that have been proposed for replacing/improving TCP.

There are several unique features in VTP, including the use of computational intelligence, via fuzzy neural networks, to try to classify and respond according to changing network conditions. A specific derivation of the fuzzy back-propagation algorithm is presented for general referential neurons using hardware-friendly triangular norms, together with a scalable digital hardware realization (incorporating in-situ learning).

Regardless of the control scheme used, VTP offers additional unique features including a built in level of acceptable packet loss towards the goal of maximizing bandwidth utilization, a separate control plane, and versatile connection and reliability settings.

Part of the versatility of VTP comes from the encapsulation of session and presentation layers, moving some aspects of reliability and connection management out of the core transport, and reducing the processing overhead at the transport layer. In addition, the session and presentation layers form the basis of re-admitting security as an aspect of reliable transport[1], through native support for authentication and encryption.

1. This may be a redundant feature with the spread of IP-Sec [46] and IPv6 [47].

II. The Versatile Transport Protocol

In this section the proposed rate-based transport protocol, Versatile Transport Protocol or VTP, will be briefly discussed, highlighting the salient features and their relationship to the control system. This is followed by a more detailed description of the control system itself, which is comprised of local connection management and a global bandwidth broker. The local controller manages each data flow or connection, while the global controller oversees all of the flows.

An application, called Phat, has been modified and used to emulate the rate-based control of VTP. Phat is a bulk transfer application using the FSTP protocol developed in [63]. The author's role in developing FSTP and the Phat application, based on UDP and using negative acknowledgement provided the initial inspiration for developing a general transport layer protocol using rate-based control to maximize bandwidth utilization.

Being an application, Phat runs over UDP, which allows Phat to implement its own rate-based control structure. After outlining the VTP protocol, the Phat application will be described together with the relevant extensions needed to support multiple connections with independent control.

2.1 Overview

The Versatile Transport Protocol, is a transport layer (layer 4) protocol whose main goal is to maximize the use of available bandwidth for a general data network. As the name implies, VTP strives to provide a flexible range of services to support different classes of applications. Included in these services is a reliable transport service through the use of negative rather than the positive acknowledgements that are used in TCP.

Essentially, negative acknowledgements are retransmission requests, which can be thought of as an acknowledgement of missing data. An important note here is that VTP does not actually propose to reduce the volume of network traffic, although it does try to limit retransmission through active requests rather than passively as with TCP.

As illustrated in Figure 2.1, VTP has two levels of control. A global control unit manages all connections, as well as interactions from the network, applications and the operating system. The second type of control is at the level of individual connections, with each connection having a separate local controller that is responsible only for that connection and which interacts with the global controller (when and if necessary).

Figure 2.1 also illustrates another important addition of VTP: it incorporates an extensible framework for the presentation (OSI layer 6) and session (OSI layer 5) layers. TCP/IP was never designed to fit into the OSI framework, nor was it intended to provide additional service above the transport layer. The result is that many useful and standardized features, such as authentication, data compression, and encryption have had to be added on to all TCP-based applications that desire these functions, much like reliability needs to be added to applications using UDP.

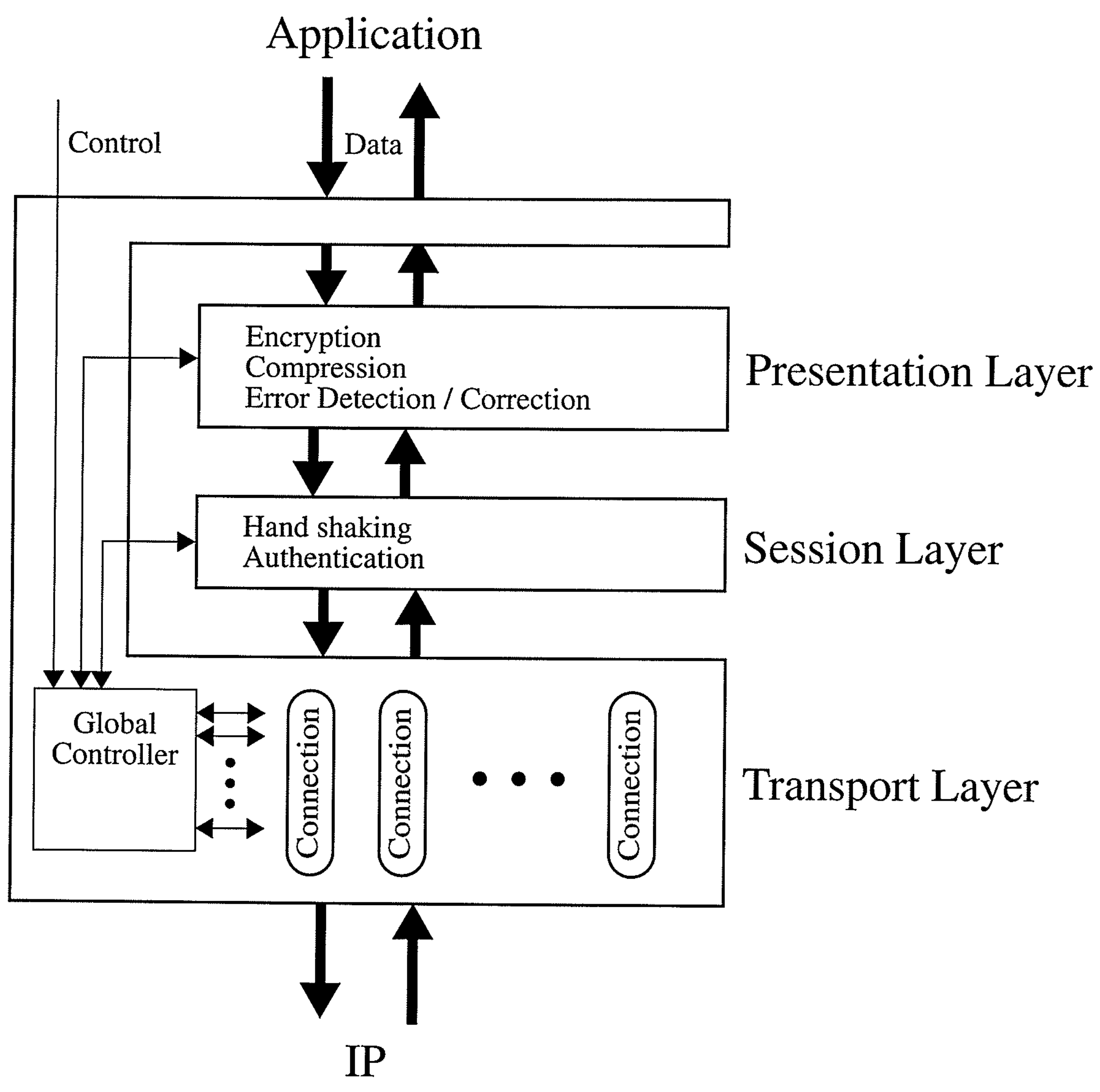


Figure 2.1 : The flow of data through the VTP architecture is shown by the heavy arrows, and control by the lighter ones. The global controller receives connection requests, allocates a new connection setting up the presentation and session layers as requested, and passes control to each connection's local controller. An application can request one or more presentation or session layer functions, or bypass them altogether and go directly through the VTP transport layer.

Any session and presentation headers need only be used during initial connection setup. After setup the header information is associated with each flow within the local flow controllers at each endpoint.

In addition VTP follows the design goal of supporting a range of services. In particular different connection schemes could be used to provide a desired level of service, while simplifying the actual transport mechanism and data packet headers.

The presentation layer is the perfect place to standardize many functions which are typically left entirely to the user, and in additional overlapping and often conflicting applications. If requested by the application, the data can first be transformed by the presentation layer by adding any or all of the requested features. These features might include encryption, compression, and error detection/correction algorithms. New categories of transformations and specific instances of each presentation type can also be easily added as new standards are introduced.

The session layer is thus being used to manage specific forms of connection-oriented service above and beyond basic VTP service. This can include TCP-like handshaking and secure authentication among others. The session layer is designed in a similar fashion to the presentation layer, specifying connection classes followed by the specific connection algorithm.

Two bits will be used in the transport header to indicate if either the presentation and/or the session layer has been requested for this flow. Error detection for the data is automatically handled by the presentation layer, and is not included in the VTP checksum. This will speed up general processing at the transport level, which can improve the performance of layer 4 switching[1] even if error detection is enabled.

Furthermore, in separating this functionality from the core transport two of the aspects of reliability - establishing and ending connections and error detection - have been made available.

The VTP presentation and session layer headers follow the same simple structure (shown in Figure 2.2); a category field followed by the header length and checksum and then one or more specific sub-header fields, one for each category requested. The category field is a collection of bits concatenated together to represent a small number of categories (32 bits). The sub-header field(s) contains an integer representation (16 bits) of the desired algorithm/protocol for the specified categories.

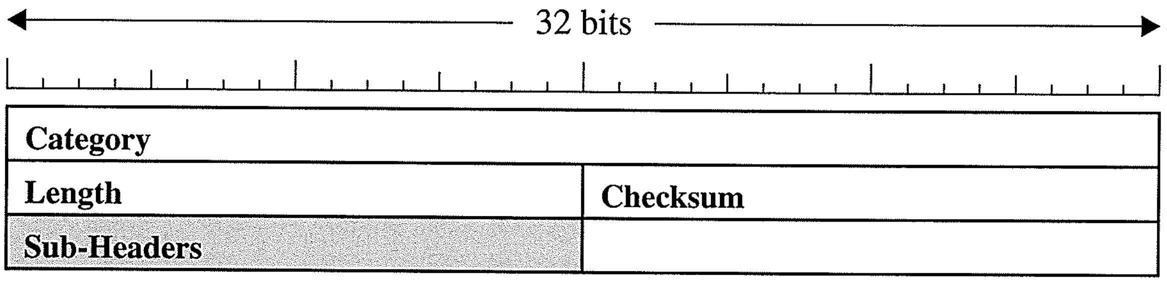


Figure 2.2 : Generic Presentation/Session Header

The next section provides a more in-depth look at rate-based control and the hierarchical control structure of VTP.

1. Used for a variety of purposes such as classifying/priorizing traffic, implementing firewalls and network address/port translation, as well as load balancing.

2.2 Rate-Based Control

The remaining aspects of reliability - flow control, re-transmission and re-ordering of packets - are provided through rate-based control. This control incorporates negative acknowledgments, packet (rather than byte) numbering, and buffer management. In addition, the level of control interaction can be varied to provide tight or loose control of each flow. By varying these parameters VTP can provide data transport ranging from a TCP-like service to a UDP-like service.

A separate control plane will provide connection setup and negotiate the initial service for each data flow as well as provide any active feedback throughout the span of the connection. The control information can be sent in separate control packets, or in a concatenated header with a data packet. The type of header/packet is specified using two flag bits: one for data and one for control (see Figure 2.3).

Rate-based control in VTP is accomplished in a hierarchical manner. A global controller oversees the allocation of available bandwidth, while local controllers initialize and monitor each flow, overseeing the desired level of reliability, signalling to change the bandwidth utilization characteristics as necessary.

Two parameters are used to characterize bandwidth: burst size and delay. The burst size specifies how many packets to send in a single burst (one right after the other); the delay then specifies how long to wait between bursts. Together with the packet size, this specifies the bandwidth of a connection, as well as the pattern of the traffic.

Once a connection is established, the incoming data flow is monitored and the rate parameters can be altered based on packet loss statistics, or updated bandwidth allocation from the global controller.

Recall that data loss can be characterized as either loss of data in the network (due to congestion, network outage, etc.) or loss of data at a host (due to buffer overflow). In performing rate-based control it is desirable to diagnose the cause of packet loss into categories that may cause the rate of the flow to be changed. For example network congestion is different from re-routing around a network problem. At the host, buffer overflow is different from corrupted data. Finally there is real packet loss vs. packet delay. Since the control is active, packet loss is calculated periodically. A packet may thus arrive after a re-transmission request has already been generated. This is typically detected when duplicate packets are received.

2.2.1 Global Control Mechanism

The global controller directly monitors the interface with the application, sets up local connections as required, and monitors and responds to requests from local connections for changes in parameters.

At the level of the local connection VTP is a greedy protocol[1], it is up to the global controller to allocate bandwidth and ensure that bandwidth exists for new connections or

1. With VTP a single connection will try to use all of the available bandwidth, choking off any additional connections - i.e. there is no inherent flow control with negative acknowledgements.

data flows (possibly including those made with non-VTP transport protocols). As such the global controller must be able to dynamically re-allocate bandwidth as new connections are started or finished.

When demand for bandwidth exceeds that which is available, the global controller can use parameters from the user, application or operating system to priorize bandwidth allocation among the connections.

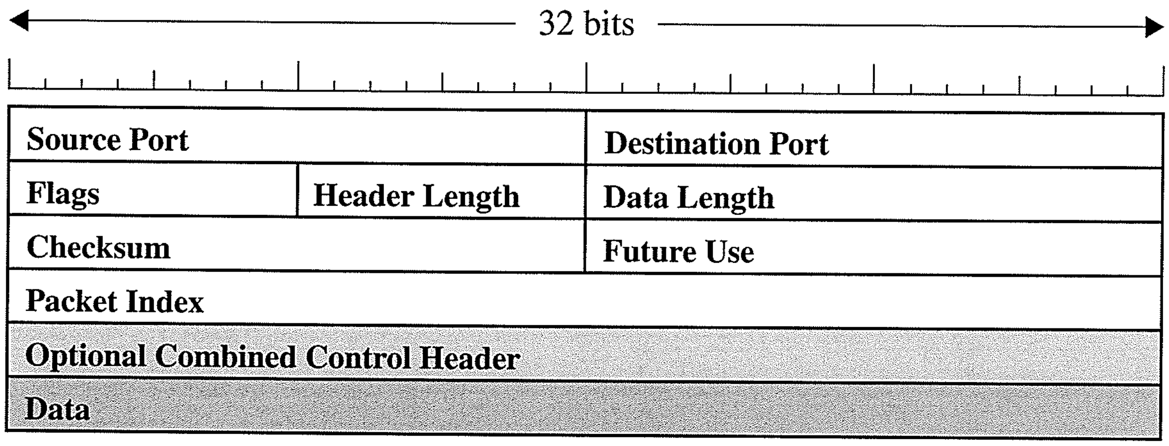


Figure 2.3 : VTP data packet header - The data header consists of two fields to identify the source and destination addresses, flags to indicate the type of header, and whether or not the data is being passed to the session or presentation layers. A header length field allows for optional concatenation of control fields. A data length field indicates how much of the fixed size packet is data. A header checksum for error detection, and finally a packet index to uniquely identify a packet within a flow (this may be made optional in the future).

2.2.2 Local Control Mechanism

In VTP each local control unit is responsible for only a single connection. Such a control unit directly controls the VTP I/O buffer for that application, monitors the state of the connection, requests re-transmission as required, and also initiates and tears down any connections. Finally the local controller is responsible for interacting with the global controller, which determines the overall bandwidth allocation among individual connections.

Like other transport protocols using negative acknowledgement, to provide fully reliable service, VTP actively manages the I/O buffer to ensure lossless flow control. When a connection request is made, the buffer size is negotiated (similar to the window size in TCP) with a maximum buffer size of 2^32 packets (matching the index field in the VTP header). Packet size is also negotiable (with a side goal of minimizing the fracturing of packets). Once set, the packet size remains constant for the life of the connection, with padding added if necessary. A length field is specified in the header to indicate the size of the data in the packet.

For bulk data transfer, where the data size can exceed the buffer size, the transfer is broken up into segments that are equal to or less than the buffer size. The local controller assigned to manage the entire data transfer maintains a single control connection with the remote host, and initializes each segment as a separate data connection. At the end of each segment, a segment complete control packet is sent (which may be concatenated into the request for the next data segment). This maintains flow control for bulk transfers. A timer at the sender will time out if no re-transmission or "segment complete" packet is received and will also free up the allocated buffer. Thus, the onus is entirely on the receiver (the data segment requester) to initiate all packet transmissions, a departure from TCP. Also unlike UDP, the transmissions will stop after any given segment if no further segments are requested.

In this manner a fully reliable connection-oriented service can be implemented, which would approximate the service of TCP with Big Windows [26], except that the equivalent maximum window size would be greater than 2^42 bytes (2^32 packets, with > 2^10 bytes/packet), rather than 2^30 bytes as with Big Windows.

The actual transmission rate will be guided by additional parameters that specify the number of packets to burst followed by a delay before the next burst. Together these two parameters control the rate and the shape of the data transmission.

Figure 2.4 illustrates a sample VTP session for just such a reliable bulk data transfer using explicit control and data packets. In this transaction, Host 1 initiates the connection to request that a data file be transferred from Host 2. In the initial control packet the data rate is specified by the burst size and delay, and the file segment size is set to match the available buffer (assigned by the global controller). Host 2 responds that the connection as specified is accepted, and Host 1 transmits the actual data request as its only data. When the data is passed to VTP, Host 2 begins to transmit the data in the agreed manner. Periodically, Host 1 evaluates the received packets and determines if the rate should be adjusted (and if so sends a control packet with the adjustment). At the end of each received segment Host 1 acknowledges the entire segment, and requests the next segment. This continues until the final segment is received, and the connection is closed.

If any packets are missing in a segment, Host 1 generates the necessary re-transmission request(s) before acknowledging the segment and waits for the missing data to be sent.

It is also possible that the rate must be adjusted at host 2, to accommodate other VTP sessions. In a reliable mode, any changes to the agreed-upon parameters will be transmitted from sender to receiver, even though a throttle back initiated by Host 2 should not affect Host 1 directly, and Host 1 can actively monitor this change via the incoming data rate.

2.3 Phat Application

This section presents a general look at the Phat application [62], focusing on the mechanisms useful for emulating VTP. Following this, the modifications to Phat made to support multiple connections, the fuzzy neural controller, and other elements required for emulating VTP are highlighted.

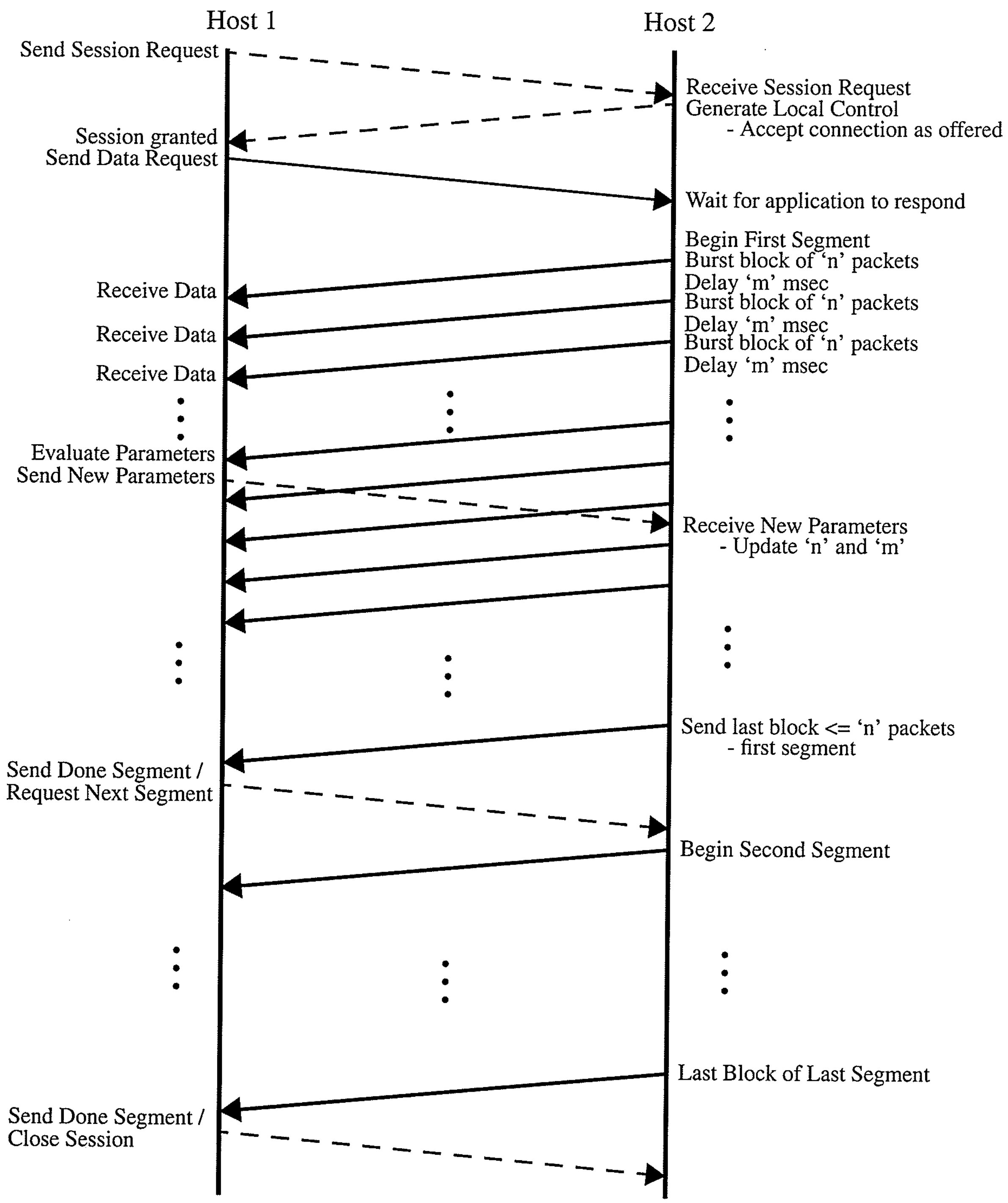


Figure 2.4 : A sample VTP session for reliable bulk transport. Dashed lines represent control packets, and solid lines represent data packets. Host 1 initiates a VTP session with Host 2, which accepts the parameters. Host 1 then transmits data to the application on Host 2 (requesting a file to be transmitted). As soon as the application on Host 2 responds, the first segment of the data is immediately transmitted in burst blocks of 'n' packets each with an 'm' msec delay between blocks. This is continued until the segment is complete, at which time Host 1 can request the next segment, or until all segments are received and the VTP session is closed.

The three main Phat client classes (modified for emulation) are presented in Appendix A. These classes are used to implement the global control (FSTPControl.java), the local control (PhatClient.java), and the data/packet handling (FSTPClient.java).

2.3.1 Overview

Phat[1] is the name of a client server application that uses phat packets (embedded in UDP packets) to provide a reliable bulk transport with negative acknowledgements [62]. Phat uses separate connections for data and control, similar to FTP [66], however the data flow is over UDP, while the control flow uses TCP.

The rate is specified using two parameters, a block size and a delay. As illustrated in Figure 2.5, the block size specifies how many packets to transmit over the network at one time (also known as the burst rate). This is followed by a delay, presumably for the receiving network to clear its input buffer. By manipulating these two parameters it is possible to set the bandwidth usage characteristics on a flow-by-flow basis. Originally Phat transferred bulk data as a single complete file. Figure 2.5 illustrates a modified version of Phat which breaks up the data into multiple independent segments. This will be explained in more detail in the next section.

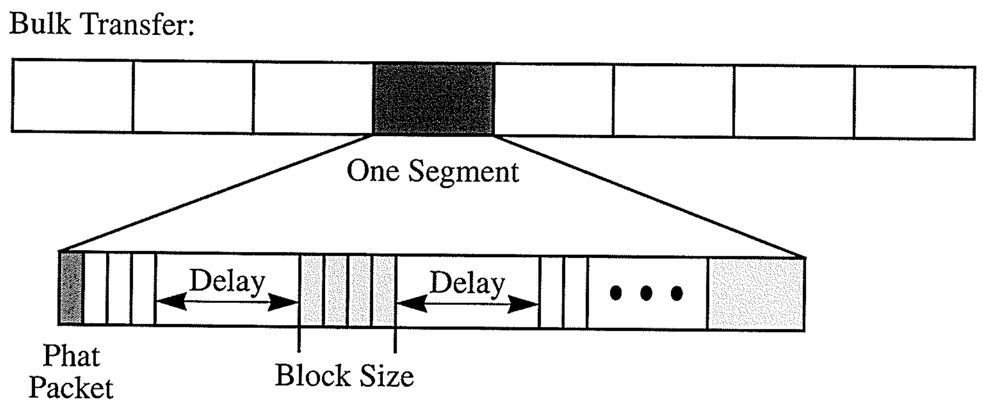


Figure 2.5 : Bulk data transfer with modified phat - Data is partitioned into segments. Each segment is composed of burst blocks of phat packets. The phat packet is the basic unit of transfer. The segment size specifies how often the connection parameters are evaluated.

The traffic characteristics of a flow can be modified at any time by the receiver, however Phat opts for a periodic examination of the received packets to determine if the rate should be adjusted. This is done by combining the perceived packet loss with a predefined acceptable level of packet loss, as illustrated in equation (1):

$$adjustment = (perceived\ packet\ loss)^{-1} \cdot (acceptable\ packet\ loss) \qquad (1)$$

Phat specifies the acceptable packet loss as a percentage of received packets called **`alpha`**. The perceived packet loss is then the ratio of packets expected to packets

1. a.k.a. Cool.

received. Note that this is a perceived (and not an actual) loss as it does not account for out-of-order or delayed packets.

The calculated adjustment is a multiplier term for the delay parameter: if less than 1 the delay is decreased (increasing the rate of transmission): if greater than 1 the delay is increased (decreasing the rate of transmission). The adjustment has a fixed upper bound (equal to two) to help reduce short term over-correction, and a lower bound of **`alpha`** when no packets are lost.

2.3.2 Modifications

The basic Phat client/server was first modified by [63] to split a bulk transfer over multiple connections to one or more servers. To do this, each file transfer is broken up into fixed-sized segments. A segment is defined to be a certain number of phat packets. Each segment is transferred as an independent data flow, handled in the same manner as described in section . Each connection has a single control flow which manages the download of many data segments. Each data segment is assigned to a connection in a round robin style as the previous segment is completed. Overseeing each connection is a single multi-connection handler.

This provides us with a hierarchical control environment for VTP: a local control for each connection (which may include multiple segments), and a global control over the entire data transfer.By treating each connection, which are assigned multiple data segments, as an independent VTP connection, it is possible to emulate the basic operation of one or more VTP sessions using Phat.

The remaining part of this section details the additional changes made to the Phat client and server to emulate and experiment with the VTP controllers.

Client

All of the active control comes on the client side. The client initializes buffers, signals the server, monitors the data transfer, requests retransmission and requests changes in transmission characteristics.

There are three basic parameters for controlling the flow of data: the block size, the delay between blocks, and the segment size. It is also possible to change the packet size, although for reliability (packet-oriented not byte-oriented) this must remain constant within each segment.

Since the control decisions are made at the client, the client requires a new control method that could incorporate the FNN architecture described in Chapter 3. In addition, the client needs to be modified to generate input and output for training the controller.

The nature of the emulation environment limits the information that can be used in making a control decision. The most reliable information available is the perceived packet loss. In addition, it is possible to determine some timing information which could be used to estimate the inter-packet arrival delay and bandwidth usage.

As the controller is experimented with, these parameters will need to be calculated, and a history maintained. Their precise usage is detailed in Chapter 4.

For testing the hierarchical control, the global controller of the client was modified to keep track of the number segments allocated to each connection, and after a pre-defined number of segments, to re-evaluate the bandwidth allocation, based on performance and a priority value.

Server

Modifications to the server were minimal, and were made with the goal of being able to emulate specific network conditions. This was done by adding a statistical packet dropper, which can be set to drop a pre-set percentage of packets according to a variety of statistical distributions.

Emulating different types of sessions, and network conditions, was handled by using separate servers with predefined characteristics.

III. Fuzzy Neural Networks (FNNs)

To introduce the fuzzy neural controller implemented for the VTP, an overview of the building blocks of fuzzy neural networks is presented including the fuzzy neurons themselves. This encompasses two main topics: the structure of the fuzzy neurons, and fuzzy neural learning.

The fuzzy neural structure is presented through a mathematical description of the aggregation process that makes up the two basic fuzzy neurons, the AND and the OR neurons. This is followed with the addition of referential parameters to further extend the basic neurons. At the end of this section it will be shown that all fuzzy neurons can be considered referential.

With the foundation of fuzzy neurons in place, generic neural learning is examined together with the relationship between learning and the triangular norms (the fuzzy sums and products) and related fuzzy operators. Specific learning rules are then derived using specific triangular norms.

The structure of the fuzzy neural networks that are of interest are multi-layer feed-forward networks [54]. Accordingly a back-propagation algorithm is developed for logic-based neurons which requires modifying the generalized delta rule [59] to form an extended generalized delta rule. This extended rule is then shown to hold for both classic (arithmetic) and fuzzy (logic-based) neurons.

During experimentation, further modifications were made and a specific Fuzzy Back-Propagation algorithm was developed. This does not change the need for, or the usage of, the extended generalized delta rule. The algorithm is presented in Chapter 5, together with the rest of the experimental results.

Finally, the Brain Construction Kit (BCK) [65], an open source Artificial Neural Network package, will be introduced. This software is used as the basis for the FNN. This section concludes with a description of the extensions to the BCK to support the two classes of fuzzy neurons (conjunctive/disjunctive), and an FNN with Fuzzy Back-propagation.

3.1 Fuzzy Neurons

Fuzzy neurons are logic-based processing units which use logical sum and product (OR and AND respectively) to perform an aggregation of weighted inputs. The OR and AND operations are realized through the standard fuzzy set connectives - triangular norms [54], with the s-norm for OR and t-norm for AND.

Triangular norms are defined as a mapping of:

$$[0, 1] \times [0, 1] \rightarrow [0, 1]$$

with the properties that these functions are:

- non-decreasing (in each argument)
- commutative
- associative, and
- bounded.

Where s-norms have the boundary conditions:

$$xs0 = x \qquad xs1 = 1$$

and t-norms have the boundary conditions:

$$xt0 = 0 \qquad xt1 = x$$

Unlike classic neurons, fuzzy neurons do not require an additional activation (or squashing) function as the triangular norms by definition produce a bounded output.

The next sections describe the basic aggregative logic neuron and then the compound referential logic neuron. This is followed by the extension of these two neuron types to include referential operations.

3.1.1 Aggregative OR and AND logic neurons

An n-input OR neuron realizes a mapping $[0,1]^n \rightarrow [0,1]$. This mapping is accomplished by first combining the inputs ANDwise with their corresponding connections (weights) while the aggregation is carried out ORwise - i.e. a sum of products. This is denoted by

$$\mathbf{y} = \mathrm{OR}\ (\mathbf{x}; \mathbf{w}) \tag{2}$$

with its coordinate-wise description given by:

$$y = (x_1 \text{ AND } w_1) \text{ OR } (x_2 \text{ AND } w_2) \text{ OR} \ldots \text{ OR } (x_n \text{ AND } w_n) \tag{3}$$

where $\mathbf{w}= [w_1, w_2, ..., w_n] \in [0,1]^n$ denotes the vector of connections (weights) of the neuron.

Substituting triangular norms for the logical sum and product, the function of the OR neuron can be reduced to the following aggregation:

$$y = \mathop{S}_{i=1}^{n} [x_i t w_i] \tag{4}$$

The AND neuron performs the complementary operation, a product of sums, with the OR and AND operators interchanged. First the inputs are linked ORwise with the corresponding connections then the final aggregation is carried out ANDwise:

$$y = \text{AND } (\mathbf{x}; \mathbf{w}) = (x_1 \text{ OR } w_1) \text{ AND } (x_2 \text{ OR } w_2) \text{ AND} \ldots \text{ AND } (x_n \text{ OR } w_n) \tag{5}$$

Employing the notation of the triangular norms, the above expression becomes:

$$y = \underset{i=1}{\overset{n}{T}} [x_i s w_i] \tag{6}$$

The role of the weights in a fuzzy neuron is to differentiate between particular levels of impact that the individual inputs might have on the result of aggregation. Owing to the boundary conditions of the triangular norms, one may note that higher values of the connections in the OR neuron cause a stronger influence on the output of the neuron than the corresponding inputs. The complementary weighting (ranking) effect takes place in the case of the AND neuron: the values of w_i close to 0 make the influence of x_i almost negligible. The specific numerical form of this relationship depends upon the triangular norms being used in the neuron's implementation.

The OR-type neurons perform a disjunctive form of aggregation, which is described as an "optimistic" character of the final aggregation. Similarly, the AND neuron performs a conjunctive form of aggregation and is characterized as being "pessimistic". What this means, is that given the same set of inputs, the output of the OR neuron will never be less than that of the AND neuron. Another way of thinking of this is the old saying "is the glass half empty or half full?".

3.1.2 Referential logic-based neurons

A second category of neurons can be built by extending the basic aggregative neurons. These neurons are useful in realizing reference computations and are thus called referential neurons.

The main idea behind this structure is that the input signals are not directly aggregated, as with the AND/OR neuron, but are first compared to a given reference point. The results

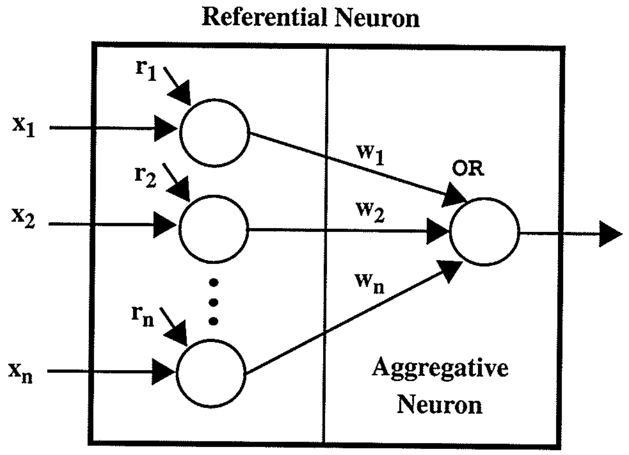


Figure 3.1 : OR aggregation - Topology of a disjunctive type fuzzy neuron with referential operations on the inputs. By separating the referential operation like this, it is possible to think of all fuzzy neurons as referential. Aggregative neurons are thus referential neurons with REF($\mathbf{x}$; $\mathbf{r}_p$) = $\mathbf{x}$.

of this comparison (involving such operations as matching, inclusion, difference, and dominance) are summarized by the aggregative part of the neuron. In general, one may describe the reference neuron as:

$$y = \mathrm{OR}\,(\mathrm{REF}(\mathbf{x};\, \mathbf{r_p}),\mathbf{w}) \tag{7}$$

(a *disjunctive* form of aggregation), or

$$y = \mathrm{AND}\,(\mathrm{REF}(\mathbf{x};\, \mathbf{r_p}),\mathbf{w}) \tag{8}$$

(a *conjunctive* form of aggregation) where the term REF($\mathbf{x}$; $\mathbf{r_p}$) represents a reference operation carried out with respect to the provided point of reference ($\mathbf{r_p}$). In fact, these neurons can be treated as an initial level of reference operations REF($\mathbf{x}$; $\mathbf{r_p}$) followed by either an AND or an OR neuron as illustrated in Figure 3.1.

The functional behavior of the neuron is described according to the reference operation (the reference formulas shown below pertain to the disjunctive form of aggregation),

(i) *The MATCHING neuron:* indicates the degree to which $\mathbf{x}$ is similar to the given reference point $\mathbf{r} = [r_1, r_2, \ldots, r_n]$.

$$y = \mathrm{MATCH}(\mathbf{x};\, \mathbf{r},\, \mathbf{w}) \tag{9}$$

The coordinatewise notation of the neuron reads as:

$$y = \underset{i\,=\,1}{\overset{n}{S}}\ [(x_i \equiv r_i)tw_i] \tag{10}$$

where $\mathbf{r} \in [0,1]^n$ is a reference point defined in the unit hypercube. To emphasize the referential character of the processing carried out by the neuron, one may rewrite (9) as:

$$y = \mathrm{OR}(\mathbf{x} \equiv \mathbf{r};\, \mathbf{w}) \tag{11}$$

The matching operator is defined as follows:

$$a \equiv b = \frac{1}{2}[(a\varphi b) \wedge (b\varphi a) + (\bar{a}\varphi\bar{b}) \wedge (\bar{b}\varphi\bar{a})] \tag{12}$$

where '$\wedge$' denotes minimum and $a\varphi b = (a \rightarrow b) = sup(c \in [0, 1] | atc \le b)$, usually referred to as a multi-valued implication or residuation operator. Examples of these fuzzy implication operators are shown in equations (27) and (28).

(ii) *The DIFFERENCE neuron*: summarizes the degree to which $\mathbf{x}$ is different from the given reference point $\mathbf{g}$. The neuron's output is interpreted as a global level of difference observed between the input $\mathbf{x}$ and the reference,

$$y = \mathrm{DIFFER}\,(\mathbf{x};\, \mathbf{w},\, \mathbf{g}) \tag{13}$$

or,

$$y = \underset{i\,=\,1}{\overset{n}{S}}\ [w_i t(x_i| \equiv g_i)] \tag{14}$$

As before, the referential character of processing is emphasized by rewriting (13), so that:

$$y = \text{OR}(\mathbf{x} \mid\equiv \mathbf{g}, \mathbf{w}) \tag{15}$$

where the difference operator is defined as the complement of the matching operator,

$$a \mid\equiv b = 1 - (a \equiv b) \tag{16}$$

(iii) *The INCLUSION neuron*: summarizes the degree to which **x** is included in a reference point **f**, (i.e. the degree to which **x** implies **f**),

$$y = \text{INCL}(\mathbf{x}; \mathbf{f}, \mathbf{w}) = \text{OR}(\mathbf{x} \rightarrow \mathbf{f}, \mathbf{w}) \tag{17}$$

or,

$$y = \mathop{S}_{i=1}^{n} [w_i t (x_i \rightarrow f_i)] \tag{18}$$

(iv) *The DOMINANCE neuron*: expresses the dual relationship to that carried out by the inclusion neuron, where **h** is a reference point. In other words, the dominance relationship generates the degree to which **x** dominates **h.**

$$y = \text{DOM}(\mathbf{x}; \mathbf{h}, \mathbf{w}) = \text{OR}(\mathbf{h} \rightarrow \mathbf{x}, \mathbf{w}) \tag{19}$$

or,

$$y = \mathop{S}_{i=1}^{n} [w_i t (h_i \rightarrow x_i)] \tag{20}$$

In general, any fuzzy neuron can be described as a referential neuron. In the case of the purely aggregative neuron it can be said that it actually implements the function $F(x_{kj}, r_j) = x_{kj}$. To this end the equation for the OR neuron (4) generalizes to:

$$y = \mathop{S}_{j=1}^{n} [F(x_{kj}, r_j) \; t \; w_j] \tag{21}$$

and similarly the equation for the AND neuron (6) generalizes to:

$$y = \mathop{T}_{j=1}^{n} [F(x_{kj}, r_j) \; s \; w_j] \tag{22}$$

3.2 Fuzzy Neural Learning

Learning with fuzzy neurons is typically described in the literature ([54][67][68][69][70][71]) by first defining the network, and then deriving a specific learning rule to match the network being implemented.

This section will derive the basic fuzzy learning rules for modifying the connection strengths (for referential parametric learning please see [57] for a complete derivation). This will be extended to a back-propagation algorithm for logic-based neurons which requires an extension to the generalized delta rule to accommodate these logic-based neurons. In this manner, a generic environment will be developed for multi-layered feed-forward FNNs.

As a starting point the basic learning rule using gradient descent is stated. Given a connection from neuron j to neuron k with strength w (the weight), we have:

$$w_{jk}^{new} = w_{jk}^{old} + \Delta w_{jk} \qquad , \Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}} \tag{23}$$

where α is the learning rate and E is the error function, which is given by the sum of squared errors[1] between a desired output (the target) and the actual output (o):

$$E = \frac{1}{2} \sum_{k=1}^{n} (target_k - o_k)^2 \tag{24}$$

This yields an error gradient of:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \left(\frac{1}{2} \sum_{k=1}^{n} (target_k - o_k)^2 \right) \tag{25}$$

By applying the chain rule with respect to the actual output we get:

$$\frac{\partial E}{\partial w_{jk}} = -(target_k - o_k) \cdot \frac{\partial o_k}{\partial w_{jk}} \tag{26}$$

To continue developing the learning rule, it is first necessary to specify both the type of aggregation (AND, OR) as well as the specific triangular norms being used.

3.2.1 Learning and Triangular Norms

Not all triangular norms may necessarily be good choices for performing learning. Of particular concern are the norms whose respective partial derivatives are zero over a certain range of the unit interval, and thereby impeding learning as detailed in [51]. An evident example is the minimum operation where regardless of the definition of the derivative at x=a, its value is zero for all x greater than a,

$$\frac{\partial}{\partial x} min(x, a) = \begin{cases} 1 & x < a \\ 0 & x > a \\ \text{undefined} & x = a \end{cases}$$

1. It should be noted that a second order error function was chosen to more closely follow the standard backpropagation in this derivation. However the approach used to determine and extend the generalized delta rule for FNNs is equally applicable to a first order error function which has been shown to be applicable to Fuzzy learning [53].

To overcome this limitation, the Boolean (two-valued) predicates are relaxed as proposed in [51] by fuzzifying the derivative. This solution uses the inclusion relation to provides a smooth (multi-valued) transition between the boundary points. This relation is performed using an implication operator such as the Lukasiewicz (27) or Godel (28) implication:

$$\text{Lukasiewicz:}\quad a \to b = \begin{cases} 1 - a + b & \text{if } a > b \\ 1 & otherwise \end{cases} \tag{27}$$

$$\text{Godel:}\quad a \to b = \begin{cases} b & \text{if } a > b \\ 1 & otherwise \end{cases} \tag{28}$$

where: $a, b \in [0, 1]$. This helps to clear the way for hardware implementations [56][57] which tend to favor triangular norms such as minimum, maximum, and the Lukasiewicz connectives:

-Lukasiewicz *and* operation:

$$a\, t\, b = \max\,[0, a+b-1] \tag{29}$$

-Lukasiewicz *or* operation:

$$a\, s\, b = \min\,[1, a+b] \tag{30}$$

where $a, b \in [0, 1]$.

Although a software implementation can make use of more flexible triangular norms, it is desirable for the target application to select hardware-friendly operations. However, it should be noted that this choice can be changed with minimal effort in the BCK software.

The remainder of this chapter will derive specific formulae based on the Lukasiewicz triangular norms and the implication operation.

3.2.2 OR Neuron

It is important to note that the presence (or absence) of the referential function does not impact the learning of connection strengths [57]. Therefore, for notational convenience, the derivation for the OR neuron will continue to use the definition of the disjunctive neuron from (4) rather than the more general definition in (21).

The derivation of the error gradient (26) now becomes:

$$\frac{\partial E}{\partial w_{jk}} = -(target_k - o_k) \cdot \frac{\partial}{\partial w_{jk}}\left(\mathop{S}_{i=1}^{n} [x_{ik}\ t\ w_{ik}] \right) \tag{31}$$

Now to carry out the detailed computation on the final multiplicand by first isolating the aggregation of the component that includes the weight that is being differentiated:

$$\underset{i=1}{\overset{n}{S}}[x_{ik}\ t\ w_{ik}] = \left(\underset{i \neq j}{S}[x_{ik}\ t\ w_{ik}] \right) s\ [x_{jk}\ t\ w_{jk}]$$

$$= min\left\{ 1, \left[\underset{i \neq j}{S}[x_{ik}\ t\ w_{ik}] \right] + [x_{jk}\ t\ w_{jk}] \right\} \tag{32}$$

Substituting for the specific triangular norms (29) and (30), and applying the chain rule again, this time defining a_{jk} as the aggregation of the j'th input with the weight connecting neuron j to neuron k:

$$\frac{\partial}{\partial w_{jk}}\left(\underset{i=1}{\overset{n}{S}} x_{ik}\ t\ w_{ik} \right) = \frac{\partial}{\partial a_{jk}}(min(1, A_{jk} + a_{jk}))\frac{\partial a_{jk}}{\partial w_{jk}} \quad \text{, where } A_{jk} = \underset{i \neq j}{S}[x_{ik}\ t\ w_{ik}]$$

$$\text{and } a_{jk} = x_{jk}\ t\ w_{jk} \tag{33}$$

$$\frac{\partial}{\partial a_{jk}}(min(1, A_{jk} + a_{jk})) = \begin{cases} 0 & , A_{jk} > \overline{a_{jk}} \\ 1 & , A_{jk} \leq \overline{a_{jk}} \end{cases} \tag{34}$$

$$\frac{\partial a_{jk}}{\partial w_{jk}} = \begin{cases} 0 & , w_{jk} < \overline{x_{jk}} \\ 1 & , w_{jk} \geq \overline{x_{jk}} \end{cases} \tag{35}$$

Substituting the inclusion reference, the derivative becomes:

$$\frac{\partial}{\partial w_{jk}}\left(\underset{i=1}{\overset{n}{S}} x_{ik}\ t\ w_{ik} \right) = INCL(\overline{a_{jk}}, A_{jk}) \cdot INCL(w_{jk}, \overline{x_{jk}}) \tag{36}$$

At this point it is worth noting that multiplication is in fact a t-type triangular norm, and is often replaced by a generic AND operator [51]. This has many advantages in generalization and implementation in hardware, however the multiplication will be left here for use in developing the extended generalized delta rule. Once formulated the product of those terms can, if desired, be replaced with a logical product.

3.2.3 AND Neuron

Beginning again from (26), the error gradient for the AND neuron using (6) is:

$$\frac{\partial E}{\partial w_{jk}} = -(target_k - o_k) \cdot \frac{\partial}{\partial w_{jk}}\left(\underset{i=1}{\overset{n}{T}}[x_{ik}\ s\ w_{ik}] \right) \tag{37}$$

A detailed computation on the final multiplicand is achieved by first isolating the aggregation of the component that includes the weight for that is differentiated:

$$\underset{i=1}{\overset{n}{T}}[x_{ik}\ s\ w_{ik}] = \left(\underset{i\neq j}{T}[x_{ik}\ s\ w_{ik}]\right)\ t\ [x_{jk}\ s\ w_{jk}]$$
$$= max\left\{0, \left[\underset{i\neq j}{T}[x_{ik}\ s\ w_{ik}]\right] + [x_{jk}\ s\ w_{jk}] - 1\right\} \qquad (38)$$

As with the OR neuron, we similarly define b_{jk} as the aggregation of the j'th input with the weight connecting neuron j to neuron k, and apply the chain rule as follows:

$$\frac{\partial}{\partial w_{jk}}\left(\underset{i=1}{\overset{n}{T}}x_{ik}\ s\ w_{ik}\right) = \frac{\partial}{\partial b_{jk}}(max(0, B_{jk}+b_{jk}-1))\frac{\partial b_{jk}}{\partial w_{jk}} \quad \text{, where } B_{jk} = \underset{i\neq j}{T}[x_{ik}\ sw_{ik}] \text{ and } b_{jk} = x_{jk}\ s\ w_{jk} \qquad (39)$$

$$\frac{\partial}{\partial b_{jk}}(max(0, B_{jk}+b_{jk}-1)) = \begin{cases} 0 & , B_{jk} < \overline{b_{jk}} \\ 1 & , B_{jk} \geq \overline{b_{jk}} \end{cases} \qquad (40)$$

$$\frac{\partial b_{jk}}{\partial w_{jk}} = \begin{cases} 0 & , w_{jk} > \overline{x_{jk}} \\ 1 & , w_{jk} \leq \overline{x_{jk}} \end{cases} \qquad (41)$$

Substituting the inclusion reference, the derivative becomes:

$$\frac{\partial}{\partial w_{jk}}\left(\underset{i=1}{\overset{n}{T}}x_{ik}\ s\ w_{ik}\right) = INCL(B_{jk}, b_{jk}) \cdot INCL(\overline{x_{jk}}, w_{jk}) \qquad (42)$$

For a complete derivation of learning rules for the FNN presented here (including parametric learning of references), hardware implementation, and network implementation please see Appendix C.

3.3 Fuzzy Back-propagation

Fuzzy neurons, more so than non-fuzzy neurons, form hierarchical multi-layered feed-forward networks, reflecting domain knowledge in the network structure. These rules can be implemented as simple sums of products or products of sums reflecting if-then rules, or far more complex structures involving many individual references at different levels of the rule.

A back-propagation algorithm has been proposed [51][56] which is suitable for feed-forward networks with hidden layers. However, the question remains: can the standard back-propagation algorithm [59] which is based on the generalized delta rule be applied to a logic-based aggregative network?

The next section introduces the foundation of the back-propagation algorithm - the generalized delta rule. As it happens, a broader generalization of this rule is required to allow for non-arithmetic summation and product operators. This new extended generalized delta rule will then be derived, and shown to apply to both cases of arithmetic and logic-based neurons. This will be followed by the derivation of the specific rules for the aggregative neurons using specific triangular norms as discussed in section 3.1. Finally, the set of referential learning rules will be presented.

3.3.1 The Generalized Delta Rule

The Generalized Delta Rule [59] uses the gradient descent of an error function to update the weight value and learn the training pattern. For a training pattern, p, the change to a weight in the connection from neuron i to neuron j is given as:

$$\Delta_p w_{ji} = \alpha \delta_{pj} o_{pi} \tag{43}$$

If the neuron is in the output layer, the error function can be determined directly by comparing the desired output (the target t) to the actual output (o). Using the sum of squared error function (24), δ_{pk} for an output neuron k and training pattern p is:

$$\delta_{pk} = (t_{pk} - o_{pk}) \cdot f'(net_{pk}) \tag{44}$$

where net is the net input to the activation function (f).

For a neuron in the hidden layer, the basic derivation is the same, except that the error function is not directly known. Instead it is an accumulation of the partial errors from the previous layer, relative to the connection strength between each pair of neurons. This will be presented in more detail for the extended derivation, but for now the result for a hidden neuron j and training pattern p is:

$$\delta_{pj} = \sum_k (\delta_{pk} \cdot w_{kj}) \cdot f'(net_{pj}) \tag{45}$$

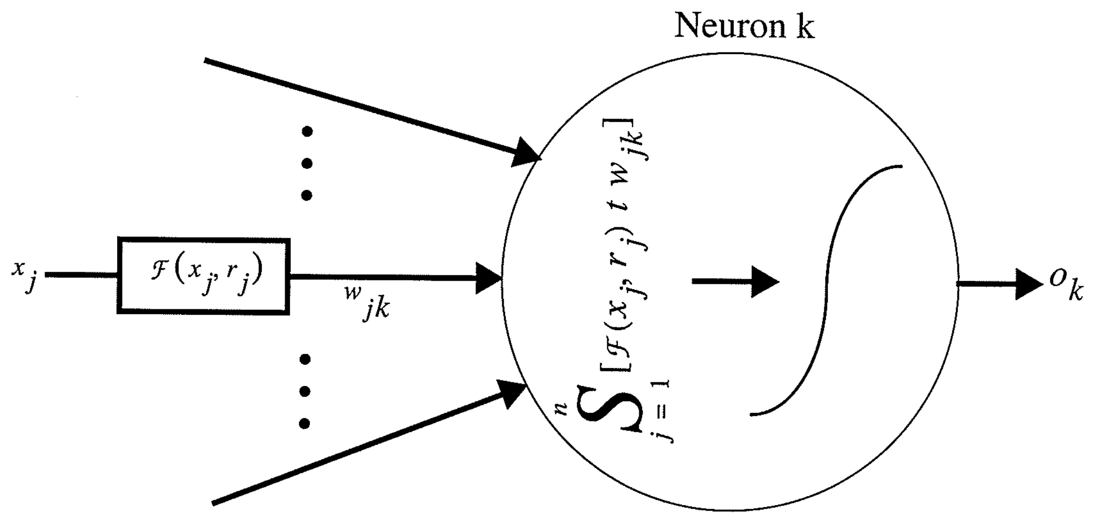
As it stands there are three problems with using the generalized delta rule with fuzzy neurons. First, a feature of the fuzzy neurons is that they do not require an activation function (f). Second, the last part of the rule (o_{pi}) is actually the derivative of the net input to the activation function (net_{pj}), which does not necessarily produce this result for a more general aggregation of inputs. Finally, there is the delta itself, which needs to be further extended to include a part of the derivative of net_{pj}.

To solve these problems, an extended generalized delta rule must be developed. The next section derives this rule following the same process as the derivation of the generalized delta rule.

3.3.2 The Extended Generalized Delta Rule (EGRD)

In deriving the delta rule for the output layer it is key to recall that non-fuzzy neurons specify an activation function which typically applies bounds and possibly a non-linear transfer to the aggregated input. This is not necessary in fuzzy neurons, as the aggregation step, be it conjunctive or disjunctive, automatically performs a normalizing function. In fact, this is a highlight of the fuzzy neuron architecture [51].

To facilitate the extension of the generalized delta rule to include logic-based neurons, the model used for the fuzzy neuron needs to be re-thought. Figure 3.2 illustrates the fuzzy neuron in correspondence with the more traditional neuron architecture. This consists of: weighted synapses (possibly with a referential transform of the input), followed by initial aggregation, followed by an activation function.



Neuron k
x_j
$\mathcal{F}(x_j, r_j)$
w_{jk}
$\underset{j=1}{\overset{n}{S}} [\mathcal{F}(x_j, r_j)\ t\ w_{jk}]$
o_k

Figure 3.2 : A traditional neuron view of a fuzzy OR neuron. The synapses (arrows pointing into the neuron) can cause a referential transform as well as a weighting of each input. These modified inputs are aggregated using a triangular norm and passed to an activation function to produce the neuron's output.

Since the aggregating triangular norm requires no further transformation of the input, a linear activation function is used in the above neuron model. Continuing with the derivation of the generalized delta rule it is recognized that the output is simply the net input to the activation function (denoted *net*) which is also the result of the aggregation.

To begin the chain rule is applied to the error gradient, as is done with the generalized delta rule:

$$\frac{\partial E_p}{\partial w_{jk}} = \frac{\partial E_p}{\partial o_{pk}} \cdot \frac{\partial o_{pk}}{\partial net_{pk}} \cdot \frac{\partial net_{pk}}{\partial w_{jk}} \tag{46}$$

Then add one additional chain, by recognizing that the net input to the activation function is a function of the weighted inputs. a_{jk} is defined to be the aggregation of the output from neuron j (possibly the output of a referential operation) to the weight connecting neuron j to neuron k. Unlike the specific definitions, this aggregation can be either a product or sum, logical or arithmetic - defined solely by the choice of neuron. This produces:

$$\frac{\partial E_p}{\partial w_{jk}} = \frac{\partial E_p}{\partial o_{pk}} \cdot \frac{\partial o_{pk}}{\partial net_{pk}} \cdot \frac{\partial net_{pk}}{\partial a_{pjk}} \cdot \frac{\partial a_{pjk}}{\partial w_{jk}} \tag{47}$$

$$\frac{\partial E_p}{\partial w_{jk}} = \frac{\partial E_p}{\partial o_{pk}} \cdot f'(net_{pk}) \cdot \frac{\partial net_{pk}}{\partial a_{pjk}} \cdot \frac{\partial a_{pjk}}{\partial w_{jk}} \qquad , o_{pk} = f(net_{pk}) \tag{48}$$

The first two terms of the RHS corresponds to the previous generalized delta. Extending this to include the third term, the derivative of the net input with respect to the specific weighted input from the j'th element becomes:

$$\frac{\partial E_p}{\partial w_{jk}} = \delta^e{}_{pk} \cdot \frac{\partial a_{pjk}}{\partial w_{jk}} \qquad , \delta^e{}_{pk} = \frac{\partial E_p}{\partial o_{pk}} \cdot f'(net_{pk}) \cdot \frac{\partial net_{pk}}{\partial a_{pjk}} \tag{49}$$

and the extended generalized delta rule becomes:

$$\Delta_p w_{ji} = \alpha \delta^e{}_{pk} \frac{\partial a_{pjk}}{\partial w_{jk}} \tag{50}$$

Given the above rule, it is now necessary to determine the first term, $\frac{\partial E_p}{\partial o_{pk}}$, for both output and hidden neurons.

For an output neuron, k, the error function is directly known (24), and the derivative is:

$$\frac{\partial E_p}{\partial o_{pk}} = -(t_k - o_k) \tag{51}$$

which yields for an output neuron:

$$\delta^e{}_{pk} = -(t_k - o_k) \cdot f'(net_{pk}) \cdot \frac{\partial net_{pk}}{\partial a_{pjk}} \tag{52}$$

For a hidden neuron, j, there is no target value, but instead a hidden neuron requires the sum of partial errors derived from neurons in the next layer (k):

$$\frac{\partial E_p}{\partial o_{pj}} = \sum_k \frac{\partial E_p}{\partial net_{pk}} \cdot \frac{\partial net_{pk}}{\partial a_{pjk}} \cdot \frac{\partial a_{pjk}}{\partial o_{pj}} \tag{53}$$

$$\frac{\partial E_p}{\partial o_{pj}} = \sum_k \delta^e{}_{pk} \cdot \frac{\partial a_{pkj}}{\partial o_{pj}} \tag{54}$$

which yields a hidden neuron error gradient of:

$$\delta^e{}_{pj} = \sum_k \left(\delta^e{}_{pk} \cdot \frac{\partial a_{pkj}}{\partial o_{pj}} \right) \cdot f'(net_{pk}) \cdot \frac{\partial net_{pj}}{\partial a_{pij}} \tag{55}$$

3.3.3 Proof of EGRD

To prove that (50) is an extended generalized delta rule, it must be shown that the EGRD does in fact reduce to either specific case of the classic (arithmetic) neuron or the fuzzy (logic) neuron.

First, to show that the extended GDR reduces to the GDR for classic neurons, equation (50) must be shown to reduce to (43) when implementing arithmetic product and summation. Starting with the last term in (50):

$$\Delta_p w_{ji} = \alpha \delta^e{}_{pk} \frac{\partial a_{pjk}}{\partial w_{jk}} \qquad , a_{pjk} = o_j \cdot w_{jk} \tag{56}$$

which now simplifies to:

$$\Delta_p w_{ji} = \alpha \delta^e{}_{pk} \cdot o_j \tag{57}$$

Now focussing on the error gradient:

$$\delta^e{}_{pk} = \frac{\partial E_p}{\partial o_{pk}} \cdot f'(net_{pk}) \cdot \frac{\partial net_{pk}}{\partial a_{pjk}} \tag{58}$$

and

$$\frac{\partial net_{pk}}{\partial a_{pjk}} = 1 \qquad , net_{pk} = \sum_j a_{jk} \tag{59}$$

so (58) becomes:

$$\delta^e{}_{pk} = \frac{\partial E_p}{\partial o_{pk}} \cdot f'(net_{pk}) \cdot 1 = \delta_{pk} \tag{60}$$

The extended generalized delta rule now reduces to the generalized delta rule, as (57) becomes:

$$\Delta_p w_{ji} = \alpha \delta_{pk} \cdot o_j \tag{61}$$

Next, to show that the extended GDR reduces to the learning rule for fuzzy neurons, equation (50) must be shown to reduce to the results from section 3.2 for general triangular (t and s) norms. This is relatively straight-forward as the main difference between the extended generalized delta rule and the rule, formalized earlier, for the fuzzy neural learning, is the derivative for the activation function. By choosing the linear activation function to model the fuzzy neuron as a classic neuron, the derivative of the activation function is 1.

$$\delta^e{}_{pk} = \frac{\partial E_p}{\partial o_{pk}} \cdot 1 \cdot \frac{\partial net_{pk}}{\partial a_{pjk}} \tag{62}$$

$$\Delta_p w_{ji} = \alpha \delta^e{}_{pk} \frac{\partial a_{pjk}}{\partial w_{jk}} = \alpha \cdot \frac{\partial E_p}{\partial o_{pk}} \cdot \frac{\partial net_{pk}}{\partial a_{pjk}} \cdot \frac{\partial a_{pjk}}{\partial w_{jk}} \tag{63}$$

Recognizing that $net_{pk} = o_{pk}$ for a fuzzy neuron, this is what was calculated in (26) and (33) for the OR neuron, and (39) for the AND neuron. This is why the product was not replaced by a generic AND operation in (36) and (42).

3.3.4 Specific Hidden Layer EGRD:

For a neuron in a hidden layer, there remains one term of the hidden layer extended delta that needs to be computed. This is the second term of the sum of back-propagated errors, which weights the relative inclusion of each error from neurons in the forward layer:

$$\frac{\partial a_{pkj}}{\partial o_{pj}} \tag{64}$$

In fact, this is the only term that depends on a referential input as for the general case for an OR neuron:

$$a_{pkj} = F(o_{pj}, r_j)\ t\ w_{kj} \tag{65}$$

Substituting a_{pkj} into (64) and applying the specific t-norm gives:

$$\frac{\partial a_{pkj}}{\partial o_{pj}} = \frac{\partial}{\partial o_{pj}}[max(0, F(o_{pj}, r_j) + w_{kj} - 1)] \tag{66}$$

At this point it is necessary to specify a specific reference function. However, it is evident already that this will yield a result similar to (35) consisting of: an inclusion reference between the three parameters of the weight, the reference value and the output of the hidden neuron.

For the case of no reference function, (66) becomes:

$$\begin{aligned}\frac{\partial a_{pkj}}{\partial o_{pj}} &= \frac{\partial}{\partial o_{pj}}[max(0, o_{pj} + w_{kj} - 1)] \\ &= \begin{cases} 0 & , w_{kj} < \overline{o_{pj}} \\ 1 & , w_{kj} \geq \overline{o_{pj}} \end{cases} \\ &= INCL(w_{kj}, \overline{o_{pj}})\end{aligned} \tag{67}$$

3.4 The Brain Construction Kit (BCK)

The Brain Construction Kit is a Java package containing a group of classes designed to facilitate an object-oriented and reusable resource for developing and implementing neural networks [65]. Currently the BCK supports five types of networks: Hopfield, Kohonen, Multi-Layer Perceptrons (MLP) using either Back-propagation or Quickprop, and Radial Basis Function.

This section provides a brief overview of the BCK, focussing on the MLP with back-propagation and how to extend this architecture to support FNNs.

3.4.1 Overview

The BCK is made up of two parts: BCK.ANN and BCK.GUI:

- BCK.ANN is the core package, providing the classes from which neural networks can be created and manipulated.
- BCK.GUI is a graphical user interface package that provides a flexible interface to most of the features of the BCK.ANN package.

For the purposes of incorporating this as an internal control unit to the VTP, only the BCK.ANN package was used.

One interesting feature of the BCK.ANN package is that it does not model the connections between neurons in the neural network object, instead it maintains a separate synapse object for this purpose. This is important for implementing the fuzzy neurons and is discussed further in section 3.4.2.

The main classes of the BCK.ANN package are the BCKNeuron class and the BCKNeuralNetwork class. As the names imply, these two classes represent a neuron and a neural network, respectively, providing the basic methods or templates for methods. To create a new neural structure one simply extends these classes to implement specific processing functions for the neurons, as well as network building (for specific architectures) and training functions for the neural network.

If additional functionality is needed by a specific type of neuron or type of network, it may be necessary to add additional templates to the neuron and neural network base classes.

3.4.2 Extensions

Extending the BCK.ANN package to implement an FNN was fairly straightforward, with the exception being the modifications required to the synapse class.

To begin with, the neuron class was extended to represent a general fuzzy neuron (BCKFuzzyNeuron). This class provides the basic triangular norms and reference functions, as well as a data structure to indicate the type of neuron and to support referential operations. This class is further extended by two classes, implementing the AND and OR type of fuzzy neurons. For compatibility with the MLP these classes provide a linear activation function. The last job of the neuron classes is to provide the necessary functionality for determining individual terms in the extended generalized delta rule.

The neural network class was extended to support construction of a fully-connected MLP style network (BCKFNN). Unwanted connections can then be removed by simply setting the weight parameters appropriately. Since the fuzzy neural structure realizes a mapping to [0,1] the neuron output is always excitatory. One method to add an inhibitory feature is to use complementary inputs. This can be described as two parallel structures, one excitatory and one inhibitory. These two structures are finally aggregated together to determine which feature dominates the neuron.

This led to a problem concerning how to implement the complementary aggregation - using a hierarchy of nodes, having neurons producing multiple outputs, or by modifying the synapse class to model a parametric vector affecting the input.

The last method was chosen yielding a synapse with one input and one or more outputs. In addition, referring back to Figure 3.2, it may be possible to move all referential operations and parameter(s) to the synapse. This may be beneficial to other neural models which desire more complicated synaptic modelling, and more consistent for learning applied to referential parameters. Extending the synapse in this way is left for future work.

Finally, the FNN class was also extended to implement fuzzy back-propagation. This class includes the main training method, as well as specialized methods for interacting with the fuzzy neuron classes to build the extended generalized delta value for each neuron, and to store this for use by earlier layers.

The following classes are included in Appendix B:

- BCKFuzzyNeuron.java
- BCKFuzzyORNeuron.java
- BCKFuzzyANDNeuron.java
- BCKFNN.java
- BCKFNNBprop.java
- BCKSynapse.java

The first five are the new classes added to support fuzzy neural networks. The last class is the synapse class which underwent major revision, and is used to implement the actual learning rule of the fuzzy back-propagation algorithm (which will be presented in Chapter 5).

To make the fuzzy neural extensions more compatible with the rest of the BCK code, it may be preferable to extend the synapse class into a fuzzy synapse, or bounded synapse class. However, corresponding place holder methods will still be required in all of the parent classes. This is also left to future work.

IV. The Experiments

To develop an experimental assessment of the VTP fuzzy neural controller, it was first necessary to precisely determine how to relate the emulation environment (the Phat client/server) to VTP. The first section in this chapter discusses the similarities and differences between the Phat application and VTP, and the underlying network used for evaluation. Section 4.2 provides a closer look at the fuzzy neural controller developed specifically for operating within the Phat environment. The basic fuzzy parameters (dealing with packet loss) are presented, and a fuzzy rule base is specified. Finally the actual fuzzy neural network is derived from the rule base. In Section 4.3 the actual experiments are described, focussing on the objective at each stage. This is followed by a discussion section which introduces additional topics that naturally arise from the use of VTP.

4.1 Setup

To emulate VTP, the Phat application has been set up to run on multiple servers with a single client. To make use of the Phat application, it is necessary to modify the overall control rules to operate specifically within the parameters of Phat. A discussion of these issues follows.

4.1.1 Phat Emulation

The general applicability of Phat as an emulation environment for VTP was discussed in Chapter 2. However, in developing the specific controller and experimentation, it is first necessary to look at how Phat can be used to emulate the operation of VTP.

Primarily Phat simply manages packet loss. It has no other goal to obtain or function to follow. VTP on the other hand strives to maximize bandwidth utilization within the confines imposed by the global controller, remote host, and network conditions.

Phat follows the simple rule that if packet loss is below an acceptable threshold then the data rate can be increased (slowly); if the packet loss moves above the threshold then the data rate must be throttled back. In this sense it can be thought of as performing part of the goal of VTP, maximizing the bandwidth with respect to the network conditions. However, with Phat there is no control over the underlying UDP buffers, and the current implementation has difficulty servicing high data rates. Consequently, packet loss can be quite high when using higher data rates, resulting in a severe choking of bandwidth. This cannot be helped in this emulation, but in trying to minimize the effect of buffer loss, the upper bandwidth of each connection is artificially limited (to 10msec - see Section 5.3).

To emulate a lower bandwidth available from the server then requested by the client, an initial bandwidth is chosen below the threshold bandwidth. Requests to than increase the bandwidth can be thought of as the server receiving additional bandwidth from the global controller, or as a recovery from an initial drop due to network conditions.

4.1.2 Network

The Phat emulation was run over a 10 Mbps Ethernet LAN. The LAN was semi-isolated during testing (on its own switch port) and, consequently, the background traffic was kept low.

The hosts are four workstations running Windows NT, three of which have the Phat server running on them and one the client. The experiments initially focussed on one client making a single connection to one server, for the purpose of testing the basic local control for reliable transport and then for fuzzy neural learning.

The remaining workstations were later used to create multiple single connections, to emulate multiple connection control via the global controller. This allows for simple experimentation with priority and re-allocation of bandwidth. In this situation, each server has its own settings, which overrules the standard local connection, forcing an explicit setting of parameters rather and ignoring the traditional Phat adjustment response. It is then upto the global controller (on the client) to determine if bandwidth should be re-allocated among the concurrent connections.

Network problems are emulated through manipulation of the servers. By delaying or dropping packets rather then sending them as expected, a variety of network conditions/ problems can be emulated.

Network conditions (such as a specific cause and characteristic of congestion) are not being defined beyond the scope of these experiments, nor specify an all encompassing rule set for the controllers. The goal here is to illustrate and evaluate the use of an adaptive (via supervised learning) rule-based system for controlling VTP, in the case of spontaneous network problems.

4.2 Fuzzy Neural Controller

The fuzzy neural controller is statically implemented within the local control thread. When an initial connection to a server is made, the neural network is initialized (using random weights), as are the fuzzy membership functions. The client thread can then call one of two methods, `fuzzyControl()` or `fuzzyTrain()`. The first method returns a crisp representation of the fuzzy controller output, which is mapped into the delay adjustment parameter. The second is used to tune the weight parameters that affect how strongly each rule affects its corresponding output.

The fuzzy controller is illustrated in Figure 4.1, with three distinct components [54]. The first component is used to fuzzify the inputs to the controller so that they are in a form that can be used by the fuzzy neural network. These fuzzy parameters are then fed into the FNN (the second component) which processes the input using a linguistic-based rules - which defines the network connectivity - and produces the control reaction as a fuzzy set. Finally, the last component resolves a crisp control value that can be used by the actual mechanism under control (the Phat server). This last process is called de-fuzzification.

The remainder of this section goes into the specific details of deriving the fuzzy controller, and concludes with a mapping of the standard fuzzy domain representation into a fuzzy neural network.

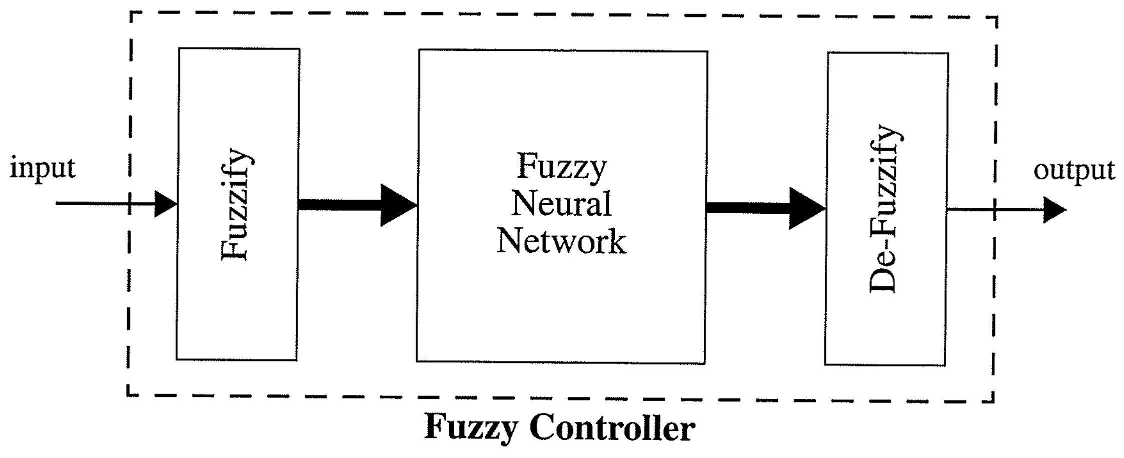


Figure 4.1 : The Fuzzy Controller. The design of the fuzzy controller is split into three parts: a pre-processor used to convert (fuzzify) crisp values to fuzzy numbers, which are then fed into the fuzzy neural network which produces a fuzzy output of the desired control. This control is then fed into a post-processor which converts (de-fuzzifies) the fuzzy number back to a crisp control output.

4.2.1 Parameters

It is possible to describe a fuzzy controller as a set of rules responding to the error and change in error of one or more parameters [54]. The most fundamental error parameter available in Phat is the apparent packet loss (referred to simply as packet loss throughout the remainder of this work). The default Phat control uses the packet loss by itself to determine a parameter called the "adjustment" to adjust the delay parameter on the server. Similarly, the fuzzy controller will use packet loss and change in packet loss as the input parameters.

The packet loss and change in packet loss will be fuzzified in a traditional manner using trapezoidal/triangular membership functions. Since packet loss is specified as a percentage, its universe of discourse is bounded by 0 to 100% packet loss. Consequently the change in packet loss has a range of [-1,1].

The fuzzy sets for packet loss are illustrated in Figure 4.2, with the numerical mapping relative to the acceptable packet loss (**`alpha`**). The packet loss is characterized by five labels (one per membership function), VS for very small, S for small, M for medium, L for large, and VL for very large. In this case, a small loss corresponds to an acceptable loss, and a large loss as unacceptable, with medium indicating room for improvement. The very small and very large membership functions provide boundary conditions.

Similarly, the change in packet loss is defined as a fuzzy set with five membership functions centered about zero. As with the packet loss, the numerical definition of each membership function is based on **`alpha`**. The change in packet loss is thus characterized by the labels zero change, and then positive large (PL) positive small (PS) negative large (NL) negative small (NS).

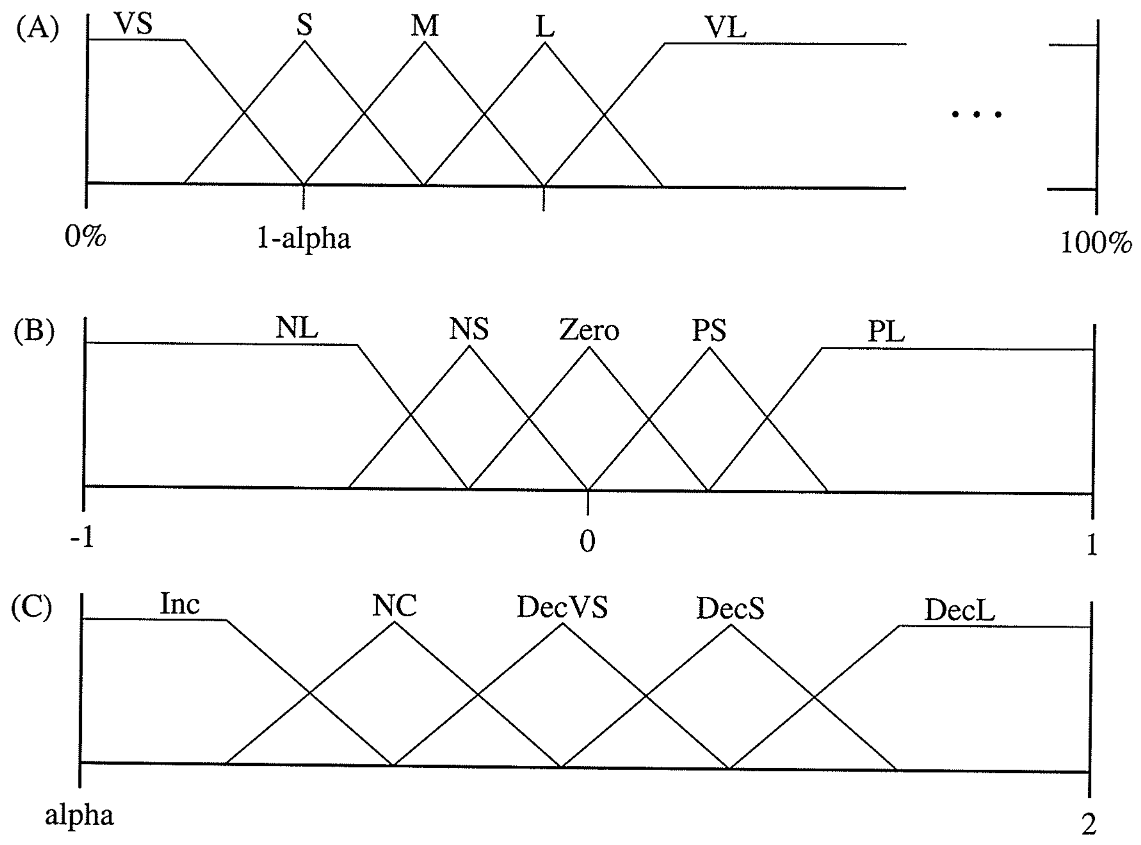


Figure 4.2 : The fuzzy Sets for (A) packet loss, (B) change in packet loss, and (C) the control (Phat adjustment).

The final parameter required is the output, or control quantity. To use the available control mechanism, the output is mapped into a fuzzy set representing the adjustment parameter. This fuzzy set is defined with five membership functions labelled Inc (increase), NC (no change), DecVS (decrease very small), DecS (decrease small) and DecL (decrease large). Once again the actual numerical mapping of the membership functions are related to **alpha**, which now acts as a lower bound for the adjustment. The upper bound of two (as shown in Figure 4.2) is also taken from the Phat control.

With the fuzzy sets defining the input and output to the FNN, it is now possible to derive the actual rule set for the controller.

4.2.2 Rule Base

In a fuzzy system the rule base defines the overall control policy in an inherently linguistic (rather than numeric) manner. Unlike traditional expert systems that try to develop exact and precise conditions, the fuzzy rule base forms a generalization of the desired control in a very simple and concise manner.

Let us recall that any fuzzy controller aims at mapping control rules of the form

$$\texttt{IF input}_1\texttt{ is A}_i\texttt{ AND input}_2\texttt{ is B}_j\texttt{ THEN control is C}_k$$

where A_i, B_j, and C_k come from a set of generic linguistic terms assumed by the corresponding input and control variables. In this sense, the controller represents qualitative domain knowledge by setting up conceptual links between the linguistic terms.

The rule base is then a concise representation of the `IF-THEN` control rules focusing on the linguistic terms. The rules are specified in the form of a matrix with each axis representing the labels, and the intersection of each label the rule.

The rule base for the VTP FNN is given in Table 1. It uses a two-dimensional matrix with each row representing one label describing packet loss (i.e. a membership function), and each column a label describing change in packet loss. The control policy (or rule) for each combination of fuzzy input membership functions, is given at the intersection of those membership functions.

		Change in Packet Loss				
		NL	NS	Zero	PS	PL
Packet Loss	VS	Inc	Inc	Inc	NC	NC
	S	NC	NC	NC	NC	Dec VS
	M	NC	NC	Dec VS	Dec VS	Dec S
	L	Dec VS	Dec VS	Dec S	Dec L	Dec L
	VL	Dec S	Dec L	Dec L	Dec L	Dec L

Table 1: Fuzzy Rule Set for Packet Loss

In this rule base each possible combination has a corresponding control action. However, certain combinations of the input may never occur, or depending on the exact quantification of the sets, combinations may not be possible. For example, the rule corresponding to a very small packet loss and a positive large change in packet loss may simply not be possible. The process of training the network to tune the weights should allow for the effective pruning of both obvious and possibly less obvious rules.

4.2.3 Network Mapping

One of the main advantages of fuzzy neural networks is that they do not need to learn their representation, instead it is provided through the domain knowledge of the rule base. Learning is then used to tune rather than learn the rules.

For the rule base in Table 1, the first rule is stated as:

```
IF Packet Loss is Very Small AND Change in Packet Loss is Negative Large
                        THEN control is Increment
```

This control rule can be directly mapped onto the structure of the fuzzy neural network, as illustrated in Figure 4.3. Observe that in the case of implementing the single rule, only a single neuron is required. If a second rule is added that produces the same output, such as:

```
IF Packet Loss is Very Small AND Change in Packet Loss is Negative Small
                        THEN control is Increment
```

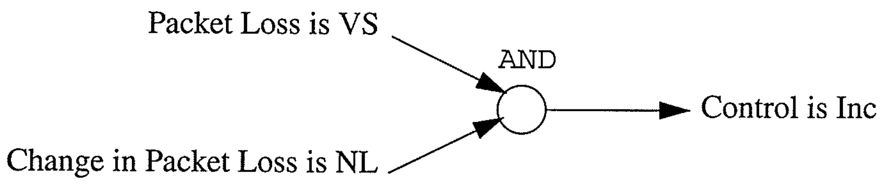


Figure 4.3 : A mapping of the first rule into a fuzzy AND neuron.

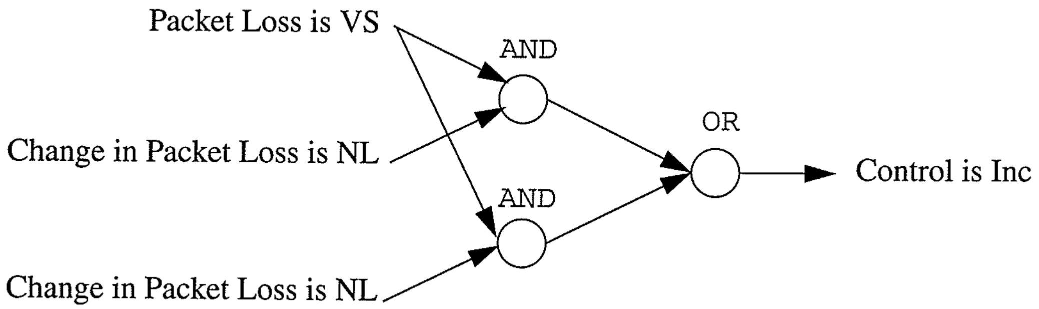


Figure 4.4 : A mapping of rules into a multi-layer network.

Then the control structure begins to form a layered network in a sum-of-products form (Figure 4.4). In this form each rule is combined in a single hidden layer consisting of AND neurons (the condition layer), and the output layer (control layer) is formed by OR neurons which summarize the control action.

In this manner the structure of the control FNN can be determined directly from the rule base (i.e. the linguistic description). The control structure has two input variables with five possible conditions for each (for a ten-neuron input layer). This results in a condition layer (without simplification) of twenty-five two-input AND neurons. This layer then feeds into the control layer using one OR neuron per control action for a total of five output neurons.

4.3 Simulations

Once the basic controller was in place, a series of simulations were designed to try to emulate specific features of VTP. Together these simulations were meant to provide a better understanding of the control process required by VTP and, specifically, to evaluate the applicability of the FNN as described in the previous section.

4.3.1 General Operation

Once all modifications to the emulation environment are completed, it is first necessary to verify the correct operation of the components. The first series of simulations were thus designed to validate the use of negative acknowledgements, illustrate how this can be used to maximize assigned bandwidth usage, and provide a baseline for further simulations.

To accomplish this, the modified Phat was first tested with its original control. This was followed by initial simulations to evaluate the FNN. Before proceeding any revisions to the FNN, or emulation environment, were implemented as required. The final introductory simulations include modest training of the network, with the goal of verifying the correctness of the supervised learning scheme, as well as the back-propagation algorithm and chosen parameters.

All of these simulations were performed using a single client and a single server. The network was monitored using an independent machine to ensure there was no actual congestion, thus any packet loss during the simulations could be traced to the host.

Together these simulations provided the basis for the VTP control, allowing the investigation of specific features to continue.

4.3.2 Learning

The next set of simulations were designed to investigate the potential of training to tune the weight parameters in the controller. These simulations again consisted of a single connection between the client and server, but now with the server modified to emulate some degree of congestion.

Tuning of the connection strengths in the controller was performed initially after each segment had been transferred. During the transfer of a segment, the actual input/output pairs were retained. After each control output had been applied to the network, a training parameter was developed by monitoring its affect over the next group of packets, as well as the cumulative effect over the entire segment.

4.3.3 Multiple VTP Connections

At this stage the testing was ready to simulate the ability of the global controller to oversee each local controller, and observe the reaction of individual connections to these changes.

The global controller monitors and allocates bandwidth to each of the local controllers. By using multiple connections, implemented as a single connection to each of multiple servers, it is possible to simulate the effect of the global controller when there is more than one VTP connection. By providing different settings at each of the servers we can simulate the bandwidth allocation to different types of applications. This could also be the result of applications assigned different priorities[1].

The use of multiple connections can also be used to simulate concurrent flows experiencing different network conditions. Again this is provided through different server settings on each host.

4.4 Discussion

This section presents a short discussion on the application of VTP on the server side

1. Priorities can be provided via a user interface in the application, parameters of the application or even from the operating system.

(which was not part of the emulation environment), and a brief look at some additional characteristics which could be applied to the general control.

4.4.1 Server Side Global Controller

The global controller, unlike the local controllers, can develop a sense of the overall network situation. For example, by monitoring connection requests, and usage statistics from the local connections, the global controller can try to manipulate individual flows to try to maintain some degree of service, providing graceful degradation rather than abrupt system failure.

One example of such a catastrophic failure is the result of a malicious attack on a server, such as a denial of service attack (DoS). A DoS attack attempts to make an exceedingly large number of fake connections, using up network resources until a server can no longer service valid connections [11]. At the very least this results in a loss of service to valid connection attempts; at worst it can take down a server (and possibly the network).

The VTP controller rules can be modified to try to provide a graceful degradation of service under extreme load by re-allocating bandwidth and reducing the level of service available to new connections.

Furthermore the global controller can try to classify the current condition and determine if it is in fact due to a DoS attack. As this classification becomes more positive, active measures can be taken to classify/drop hostile connections.

4.4.2 Alternative Parameters

The fuzzy controller developed in this thesis is based on one parameter of error, the packet loss. It may, however, be useful to investigate other parameters that can be calculated from packet arrival. In addition, there are many network metrics which if they could be obtained reliably, would be very useful.

Delay Characteristics

Inter-packet delay (within a block) can be used to give the controller an idea of the active jitter rate in the network. It may not be useful to try to match the data flow parameters to this rate, but there may be some useful information, possibly over longer durations such as during a bulk transfer.

As mentioned throughout the thesis, the packet loss parameter is actually perceived packet loss. Another characteristic related to delay is the arrival of out-of-order packets, specifically, but not limited to, those that have been characterized as packet loss and which resulted in a re-transmission request.

Burst Characteristics

Instead of focussing on the individual packet, the data block that is burst can also be evaluated. An error parameter could be based on what percentage of a block arrives, and if they arrive out-of-order. Also, an inter-block delay can be evaluated.

Network Metrics

Possibly the most useful information would be actual information regarding the current and changing state of the physical end-to-end network. Much work is being done to develop both passive and active techniques for determining these characteristics [73]. Probably the most useful to VTP would be the Bandwidth-Delay-Product (BDP).

The simplified BDP is calculated by multiplying the round trip time (RTT), which can be approximated[1] using the **`ping`** protocol, by the maximum bandwidth of the least capable hop in the route from source to destination. Unfortunately, there is currently no method for quickly and easily obtaining the maximum bandwidth of the least capable hop.

Nevertheless, as new measurement techniques are developed, it may be worthwhile to investigate their usage in VTP.

1. Ping uses ICMP echo request messages which are not necessarily treated the same as TCP/IP packets at intervening nodes (routers), or subject to data processing delays.

V. The Results

Preliminary results have been quite promising for using FNNs to provide for local connection control in the emulation environment.

In the initial simulations the emulation environment was found to be generally suitable, within certain guidelines. Specific problems encountered with the fuzzy controller in general and the fuzzy neural network specifically were identified and corrected.

The first two sections of this chapter deal specifically with these issues beginning with a discussion of the fuzzy rule base and a redesign of the fuzzy controller. This is followed by an investigation of a problem discovered with the learning algorithm, and the development of a fuzzy back-propagation algorithm. Finally, a brief section summarizes the general findings on using Phat to emulate VTP, with the modified FNN acting as a local (per flow) controller.

The last two sections then discuss the results obtained from tuning the controller under varying conditions, and the simulation of the effect of the global controller.

5.1 The Fuzzy Controller

Preliminary results after switching Phat to use the fuzzy controller, were quite good. Starting with smaller file transfers (< 1 MB), the controller provided a reliable transport as the bandwidth allocation to the connection was increased. However, over the lifetime of the connection there was a general trend towards a reduced data rate, despite the fact that the controller increased the data rate more often than decreasing it.

When larger files were used (> 20 MB) the delay quickly grew. The main difference between the control of the smaller and the larger file being that the number of times the delay is re-evaluated also grew. The problem was that the control value being generated was quite small for most increases, compared to the magnitude of the less frequent decreases.

By examining the specific numerical results, the problem became apparent. The fuzzy set control had been designed based on the original control structure. In particular, the universe of discourse and the membership functions had been made relative to **`alpha`**. While this made sense for small corrective responses, it did not allow for a response to a periodic large decrease.

The first change made was to extend the set boundary to accommodate a fast recovery from any aperiodic large decrease. As the control action is multiplicative, the lower bound was reduced to 0.5 for symmetrical control. Also, rather than defining a control action to mimic the Phat control, the membership functions were reformulated in a more traditional fuzzy sense. The new fuzzy set is illustrated in Figure 5.1, still using five membership functions. These functions have been re-labelled as PL (positive large), PS (positive small), ONE (no change), NS (Negative Small), and NL (negative large). This is a more traditional approach centered about the zero change membership function (in this case representing a control value of 1). The figure is shown with a non-linear universe of discourse to better illustrate the relation of the membership functions relative to their control.

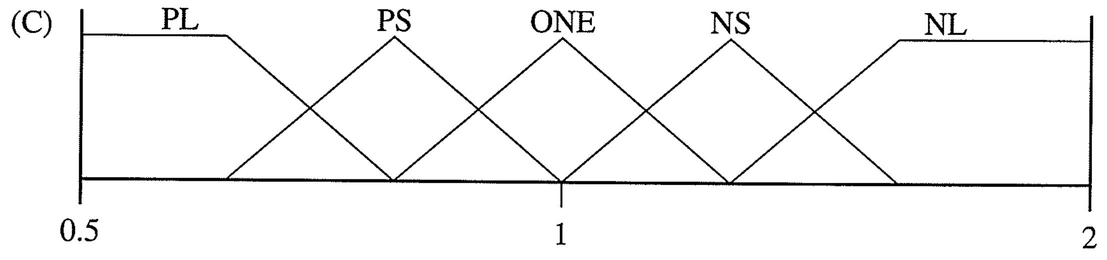


Figure 5.1 : Revised Fuzzy Set for Control Output.

After a few simulations with the revised control, it was determined that this would suffice. The new rule base for the VTP FNN is presented in Table 2.

Packet Loss \ Change in Packet Loss	NL	NS	Zero	PS	PL
VS	PL	PL	PL	PS	PS
S	PS	PS	PS	PS	ONE
M	PS	PS	ONE	NS	NS
L	ONE	ONE	NS	NL	NL
VL	NS	NL	NL	NL	NL

Table 2: Fuzzy Rule Set for Packet Loss

5.2 Fuzzy Learning Re-Visited

With the modified fuzzy controller in place, the initial indications was that this controller could better achieve the goals of VTP, maximizing bandwidth usage. The next step was to test the back-propagation algorithm for tuning the fuzzy rule base. At this point a problem with the general learning algorithm quickly became apparent.

The problem lies in the very essence of the learning rule (23) that was used:

$$w_{new} = w_{old} + \Delta w$$

The actual observed problem was quite small given that a representation is not being learned, merely tuning the relative parametric representation already derived. However in theory the problem is a fundamental error to the approach used, given that for logic-based neurons the weight is defined to be an element of [0,1]. Unfortunately no bounds were in place at the learning rule to insure that this condition was maintained.

In re-examination the basic back-propagation algorithm using the extended generalized delta rule the cause of this problem was found to be two-fold. The first problem deals with the arithmetic addition in the learning rule itself. Being an unbounded operation, the weights have the potential to exceed their required bounds. The most obvious solution was to replace this with some kind of triangular norms (an s-norm for incrementing and a t-norm for decrementing).

This lead to the second problem, that of using a bounded function to update the weight requires a re-examination of the actual change in weight to be used. Otherwise, an unbounded change in weight function has the potential to overwhelm any weight value, thus changing the nature of the parametric tuning to one of extreme oscillation.

Of course this type of problem is not new to neural networks and is the reason for the error derivative being proportional to the change in error. Thus a learning rate parameter is often used as a method to limit, and in some networks reduce over time, the effect of the weight changes on the network. Typically in fuzzy neural systems this problem is dealt with by using a very small learning rate, on the order of $<10^{-3}$ [55].

Within the extended generalized delta rule it is the summation of back-propagated errors which is unbounded. The upper bound of the summation is limited only by knowledge of the network structure, i.e. the number of neurons that are fed in the next layer. Since the maximum magnitude of each back-propagated error from the output layer is 1, then for any neuron in the hidden layer the sum term is at most M, where the neuron feeds into M output neurons. As a generalization it can then be said that for any neuron in a hidden layer, the maximum sum term is a product of the number of neurons in the later layers.

With this knowledge the maximum sum/error term for any neuron in the network can be specified as being the maximum sum term of layer 2, or the product of the dimension of layers 3 through N, where N is the output layer.

Because all of the other terms are elements of [0,1] or [-1,1], this product is also the maximum change in weight: Δw_{Max}. This term can be used to guarantee that the magnitude of the weight change maps into [0,1] for any connection strength in a feed-forward network. With this in place the new learning rule was developed.

Having mapped the magnitude of the change in weight to [0,1], triangular norms can be applied to aggregate the old weight with the weight change.

The learning rule now becomes:

$$w_{new} = \begin{cases} w_{old}\ s\ \Delta w_{norm} & , \Delta w \geq 0 \\ \overline{w_{old}\ t\ \Delta w_{norm}} & , \Delta w < 0 \end{cases} \tag{68}$$

where the normalized change in weight is:

$$\Delta w_{norm} = \left|\frac{\Delta w}{\Delta w_{Max}}\right| \tag{69}$$

Using the Lukasiewicz triangular norms, (29) and (30), the new learning rule is simply bounded addition and subtraction.

In essence what this does is to replace the standard notion of the learning rate as a network-independent parameter. This idea is replaced by one in which the final learning rate is partly network-independent and partly network-dependent, in essence:

$$\alpha_{modified} = \frac{\alpha_{independent}}{\Delta w_{Max}} \tag{70}$$

In this way the level of learning ($\alpha_{independent}$) can be maintained in a consistent and meaningful manner, which is separate from the use of the learning rate to compensate for network dimensionality.

Thus the fuzzy back-propagation algorithm is defined by applying the learning rule from (68) with the extended generalized delta rule.

5.3 Initial Simulations

After correcting the problems with both the fuzzy controller and the underlying learning rule for the FNN, the initial simulations were repeated.

These simulations began by verifying the use of negative acknowledgements and found them capable of a reliable method for bulk transport. Also, an important baseline condition was observed for use in future simulations.

This was the data rate at which UDP buffer loss[1] became prevalent. In essence this is caused by the overhead of the emulation environment (Phat) running over UDP. With no direct buffer control using the UDP socket interface, this is a fundamental limit in the remaining simulations. To properly analyze the remaining tests, the upper bound of the VTP controller (i.e. the allocated bandwidth) was always specified below this limit.

The limit was obtained by progressively increasing the data rate (decreasing the burst delay) and observing the packet loss. The same 20 MByte test file was transferred 5 times at each delay setting, varying the delay[2] from 50 to 1 msec. The results are summarized in Table 3. The results are relative to the evaluation size (the number of packets to receive before evaluating a control decision), which was set to 50. The parameters are maximum number of packets received (Max) minimum number received (Min), average number of packets received (Avg), the absolute number of packets lost (Abs. Loss), and the measured bandwidth (BW).

The measured bandwidth, even before packet loss occurred, is well below the theoretical bandwidth for each given delay setting. This is because the actual bandwidth is a function of three delay parameters. First there is the transmission delay of the bits on the wire (approximately 5 ms per block), second there is the processing delay, and third there is the explicit block delay. Overall, before packet loss started, the efficiency of the transfer

1. Recall that the test network is an isolated 10 Mbps LAN segment.
2. The delay in the emulation environment is specified as an integer in milliseconds, so the minimum delay available is 1 msec.

increased from 25% to 31% of the theoretical maximum data rate (ignoring processing time) as the delay decreased.

Delay (msec)	**50**	**40**	**30**	**20**	**15**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
Max. (0-50)	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
Min. (0-50)	50	50	50	50	50	50	50	49	48	47	46	47	46	33	15
Avg. (0-50)	50	50	50	50	50	50	50	49.97	49.90	49.78	49.75	49.60	49.32	48.68	47.26
Abs. Loss (packets)	0	0	0	0	0	0	0	6	20	48	56	89	149	290	602
BW (Kbps)	222	275	360	528	685	992	1050	1101	1112	1140	996	936	998	1152	1496

Table 3: Evaluation of UDP buffer loss for block delays of 50 down to 1 millisecond.

After losses began, the data rate fell and then picked up again as the difference between the increased data rate and the retransmission favoured the data rate. However the efficiency fell off noticeably to under 20%.

For comparison, a standard TCP-based FTP client[1] was tested in the same manner to transfer the same file in the test environment. The average transfer rate for this file transfer client was 802.26 Kbps. This was surpassed by VTP as the inter-block delay was decreased below 15 msec.

It is important to note that the actual data throughput improves even with packet loss and re-transmission. This helps to validate the basic precept of the control structure; that a level of acceptable packet loss can be used to improve the effective bandwidth usage.

With these baselines in place, the maximum bandwidth allocated to a connection for the latter simulations was set to correspond to a 10 msec delay.

5.3.1 Testing Controlled Transfer

After the initial simulations had resulted in the redesign of the fuzzy controller, a test of the controlled file transfer was repeated. This time the fuzzy controller performed well on its own, and in the case of achieving and maintaining maximum data rate it excelled.

Over the lifetime of the bulk transfer, the original Phat controller hardly increased its data rate, despite no, or very little, data loss. Unregulated, the VTP controller consistently balanced the data rate with a delay between 10 and 6 msec, even when started at 50. Consequently the VTP controller exhibited greater packet loss, and generated more traffic via re-transmission requests. This was successful in relation to the basic goals of VTP,

1. The FTP client used is WS_FTP LE v5.08, and the server was the WAR FTPD v1.7.

managing to find and stay close to a maximum data rate, within the acceptable parameters of packet loss. The original Phat controller needed to start much closer to its final value (within 10 msec) and was more conservative in maintaining it. This was similar to the result of the original fuzzy rule set described in section 5.1. Even within these limitations, and reducing the alpha parameter to 90% packet throughput, the original Phat control did not achieve the same overall throughput.

One possible improvement to the fuzzy controller would be to add additional membership functions for the controller fuzzy set to increase the resolution, in particular, about the no change point (ONE).

5.4 Tuning the Controller

With the initial simulations completed a basic functioning FNN-based controller for VTP was available for further evaluation. The next stage was to try and tune the FNN to respond to a specific network condition. This condition was a simple congestion-like packet loss emulated at the server side by dropping randomly selected packets.

To perform supervised training, a target value must be known for a specific set of inputs. To accomplish this with real time data, a training value had to be calculated after the result of applying the current control action is known, i.e. by observing the next set of packet statistics. That is, there is no pre-defined (off-line) training set for training and testing with this application. It is therefore necessary to actually develop the target value during training by observing the result of each control operation.

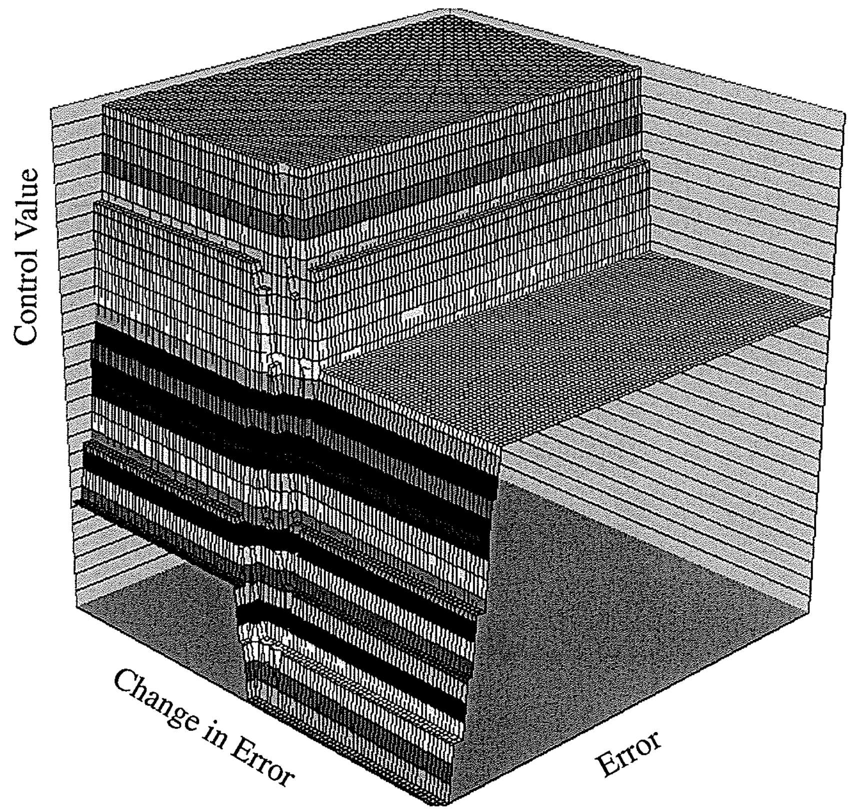
Each time the fuzzy controller is activated, the fuzzy inputs and the corresponding fuzzy control output are stored in a training pair. After the next control decision is made, the result of the control action is known, and the new control action is compared to the previous control action to determine a target value. This then replaces the original control action in the training pair. After each segment is completed, the accumulated training pairs are used to train the network.

5.4.1 Results of Tuning the Controller

A 20 MByte file was transferred using the modified Phat (with training enabled) and a 5% random packet drop set on the server, to generate the training data.This data was used to tune the FNN controller. Figure 5.2 shows the control surface generated by the FNN-based controller both before and after tuning.

Since the target value is not known, but calculated based on the result, a scheme to determine the target was first implemented that used the maximal change. The results from this first scheme were poor, resulting in oscillating of the tuning values, with little or no actual change between iterations. To correct this, a second scheme which compared the calculated control value to the result produced by that change was implemented. This produced a more stable tuning over time.

A)



B)

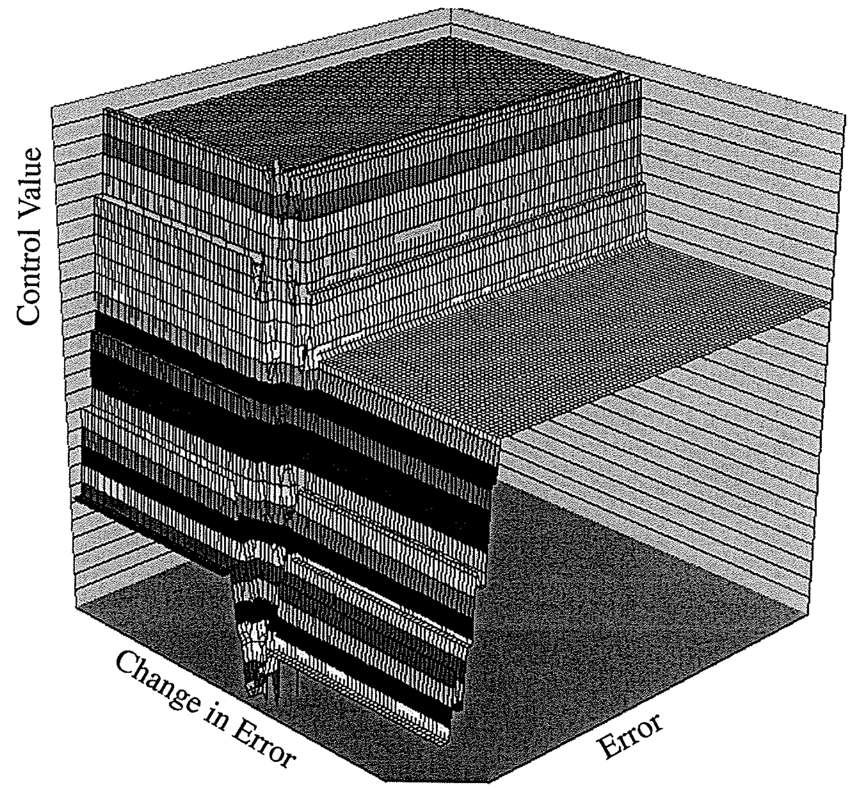


Figure 5.2 : FNN control surfaces; (a) before tuning and (b) after tuning.

To test the effectiveness of tuning in the controller data transfers were performed both with varying file size and varying percentage of packet loss. Figure 5.3 shows a comparison of transferring files of various size ranging from 1 to 100 Mbyte both before and after tuning the controller, and Figure 5.4 shows a comparison of transferring the same file both before and after tuning the controller, but with varying percentages of dropped packets. Both comparisons show a small improvement after tuning (between 2 and 3 percent).

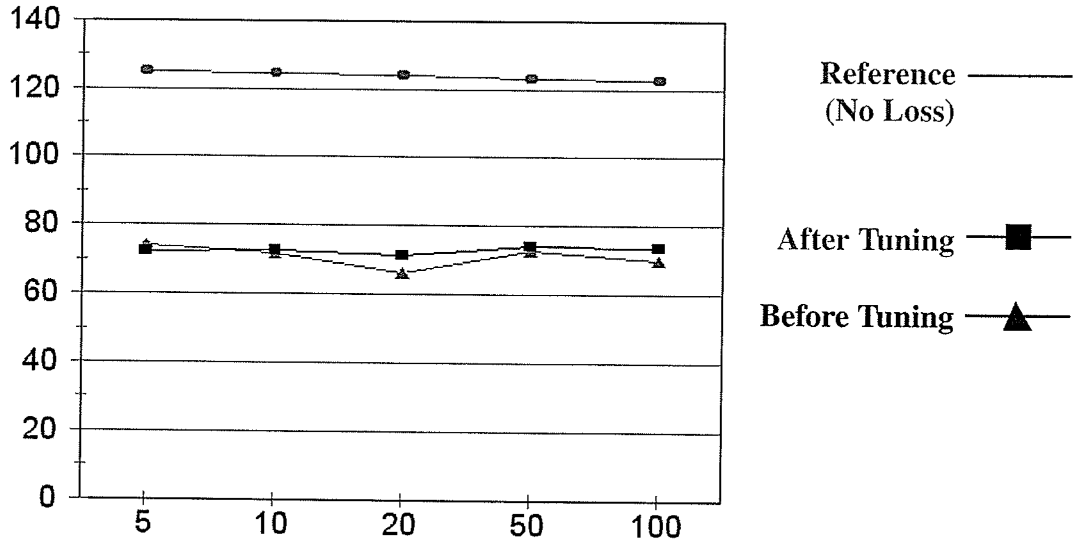


Figure 5.3 : Graph of throughput for varying file sizes with constant error rate (5%).

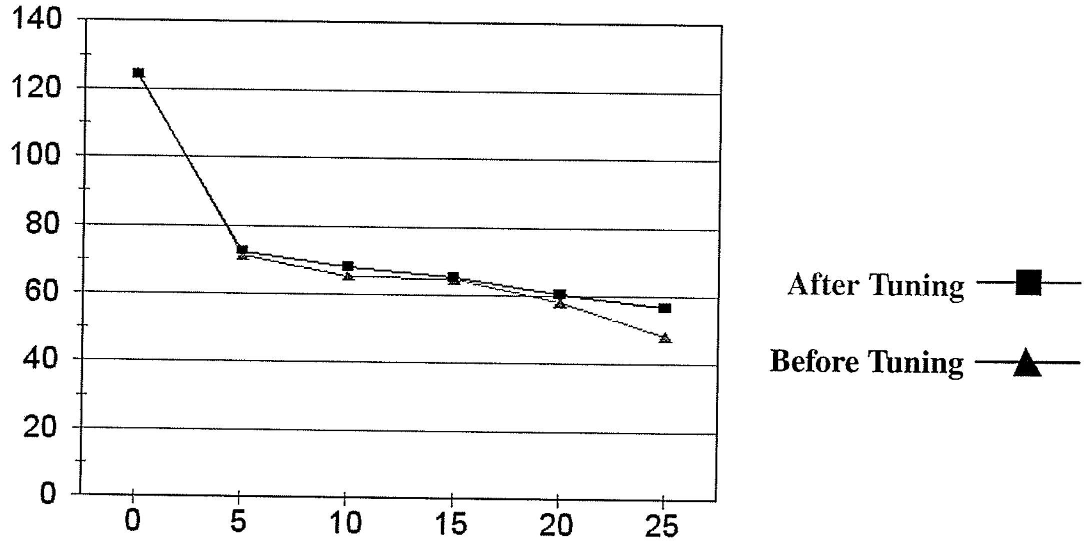


Figure 5.4 : Graph of throughput for transferring the same file (20 Mbyte) with varying error rates.

5.4.2 Discussion

In general this is a promising result, showing a small but steady improvement from a simple on-line tuning session. However the tuning itself presented a result that was not expected. The idea was that the tuning would be used to improve the reaction to the simulated network congestion. In short the tuning was to improve the detecting and re-acting to congestion of a certain characteristic, and therefore improve throughput. That characteristic was expressed as packet loss, and was independent of data length. As a result an improvement was sought over the first case (varying the file size) but was not expected to as great a degree when varying the percentage of packet dropped.

Re-evaluating this experiment it was found to be fundamentally flawed. That is, the implementation of network congestion - random packet dropping at the server - does not in fact describe congested behavior. Because each packet was evaluated independently, there was no relation between control reaction and future performance with regards to packet loss.

Instead what the controller tuning seemed to respond to, and where a performance improvement was found, was in the minimizing of re-transmission losses. When the Phat server goes into re-transmission mode it does not use the same controlled transmission as with the original transmission. As a result when large numbers of packets are requested to be re-transmitted, they are all burst, and buffer losses occur, resulting in repeated re-transmission requests. Thus the re-transmission mode has the characteristic of a damped oscillation, with each subsequent re-transmission reducing until reaching zero.

The tuned controller tries to minimize this occurrence through increased reaction when immediate losses occurs. This seems to improve throughput by reducing short periods of loss, then quickly recovering, thus having fewer re-transmissions at a time. This is most evident by the largest improvement corresponding to the largest packet loss (25% drop rate - Figure 5.4).

In this manner tuning was successful. Even with a low number of trials where the error is not statistically significant, each transfer showed an improvement and the results in all cases are repeatable.

5.5 A Simple Global Controller

The control in Phat employs a hierarchical approach with a global control (FSTPControl.java) that assigns segments of a file to each local connection thread. A single file transfer is setup in Phat using single connections to multiple (three) servers in order to emulate multiple simultaneous VTP connections.

This global controller was modified to implement a simple bandwidth broker to re-allocate bandwidth during the lifetime of the connections. The modifications included keeping track of the number of segments allocated to each connection (to represent bandwidth - more segments correspond to higher throughput), specifying initial maximum bandwidth to each connection and after the 40'th segment was allocated, to re-evaluate the maximum connection bandwidth and re-allocate if necessary. Once again the bandwidth is specified by a delay parameter, with a global maximum bandwidth of 10 msec delay.

Initially each connection is then allocated 10*(number of segments) msec delay. So for the three connections, each is initially set with a maximum of 30 msec. Re-allocation is hard encoded to retain a global 10 msec delay over all of the connections.

To provide different connection statistics, the servers are modified with a pre-set maximum delay (over-riding the initial setup). This is to emulate a heavily loaded server or lower bandwidth connection. In this way the servers use the lower of this maximum delay setting, or the connection delay parameter.

Two tests are run; first with one server set to a high delay value (connection 3), and the other two set evenly. Then, using the same setup, a priority value is introduced to the first connection made by the global controller (at the client).

5.5.1 Testing the Global Controller

In the first test the delay parameters were set so that the connection(s) unable to meet their assigned bandwidth (based on segments) would have the delay increase by 20 msec increments, with the freed bandwidth shared proportionally among the connections able to utilize their assigned bandwidth.

As summarized in Table 4, without using a priority, the bandwidth allocated to connection 3 was reduced from a 30 msec to a 50 msec delay. The freed bandwidth was then re-allocated evenly to the first two connections (which had requested 17 segments each) to reduce their delay from 30 to 25 msec.

	Initial Delay	**After 40 Segments (No Priority)**	**After 40 Segments (Priority)**
Connection 1	30	25	22
Connection 2	30	25	30
Connection 3	30	50	50

Table 4: Results of using a global bandwidth broker to re-allocate bandwidth from a low bandwidth connection (#3) to high bandwidth connections (#1 and #2).

The second test used a priority value assigned to differentiate between the two high bandwidth connections. The priority rule was set to first re-allocate bandwidth evenly among connection(s) with highest priority until reaching preset connection maximums (20 msec delays for individual connections in this test). Any remaining bandwidth is then distributed to connections with the next level of priority, and so on.

As shown in Table 4 (with priority), connection 3 again had its bandwidth reduced (to a 50 msec delay). This time all of the available bandwidth was re-allocated to the first connection (which had the highest priority), while the second connection was left unchanged.

VI. Conclusion

6.1 Overview

The work presented in this thesis brings together transport protocols and computational intelligence to try and improve upon the state of modern data network communication.

The Transmission Control Protocol (TCP) is the dominant transport layer protocol in use today, providing a reliable connection oriented service. However the windows based control applies an artificial limitation on bandwidth utilization. As the distance (measured in packet latency) increases so does the limitation of TCP on the bandwidth utilization. A readily available alternative to TCP is UDP, the User Datagram Protocol, which provides an unreliable connectionless service. UDP can be used by any application to avoid the overhead of TCP, however as its description implies, it provides no service to applications which may require some or all or the services of TCP. This is a very inefficient manner in which to solve the inherent limitations of TCP.

To overcome the limitations of these two protocols a new transport layer protocol was developed in this thesis called the versatile transport protocol (VTP). It aims to avoid the bandwidth limitation of TCP through the use of negative acknowledgements and active rate based control. Furthermore, unlike either TCP or UDP, VTP strives for versatility in providing a range of services as desired by the application. These services range from a fully reliable connection-oriented service (similar to TCP with big windows) all the way down to an unreliable connectionless service even more basic than UDP.

To do this VTP makes use of both session and presentation layers to implement areas of reliability that are not part of the core transport protocol. This allows for the re-introduction of security as an element of reliability, providing a standard interface for encryption or authentication algorithms. It also adds flexibility with a framework for easily adding standardized algorithms for data manipulation such as compression as well as having the added benefit of streamlining the processing requirements of the transport layer.

VTP uses two levels of control, a global controller which acts as a bandwidth broker to incoming and outgoing connections on the local host and a local (per session) controller, which independently controls each network connection.

To implement the control in VTP this thesis proposes a fuzzy neural controller with in-situ learning. This would allow each connection (via its local controller) to adapt its connection parameters to the perceived network conditions over the lifetime of the connection.

To implement the fuzzy neural controller, a fuzzy back-propagation algorithm is developed. In developing this algorithm the basic generalized delta rule (GDR) for back-propagation in arithmetic neurons was extended to include logic-based neurons. By extending the GDR, we are able to design and implement arbitrary feed-forward neural structures, without the need for custom implementations. In doing this not only are arithmetic and logic neurons unified, but also a foundation is laid for a hardware-based reconfigurable fuzzy processing unit.

Unlike arithmetic neurons the connection strengths for logic neurons must also be bounded. This led to the first modification of the fuzzy back-propagation algorithm, to guarantee the correctness of the connection strength parameters during training. This led to a second modification, requiring a different view of the learning rate, one that incorporates both a network independent as well as a network sensitive part.

In addition to contributions made to fuzzy neural network control theory, this work illustrates the viability of fuzzy control as applied to telecommunication applications such as bandwidth brokering within the confines of an IP network. In addition, the study illustrates the role that fuzzy control can play in improving end-to-end performance by adaptation to host and/or network bottlenecks.

6.2 Results

The initial data transfers showed that the emulation environment could achieve greater than 30% efficiency (relative to the theoretical maximum data rate) before UDP buffer losses became a factor. After overcoming the overhead due to buffer loss, the bandwidth usage and the actual data rate still increased to nearly double the measured bandwidth usage of a TCP-based FTP transfer under the same conditions. Despite the overall gain in bandwidth usage and effective data rate, the efficiency once packet loss began dropped significantly (to less than 20%).

This was used to form a baseline for further experiments within the emulation environment. A minimum delay setting for further experiments was set to restrict packet loss to the explicit dropped packets (representing network losses). This limit, set into Phat, was a minimum delay of 10 msec (corresponding to the maximum bandwidth allocation by the global controller in VTP). It also illustrates the underlying importance of active buffer control for reliable data transport (i.e. flow control).

The remaining initial transfers showed that the FNN-based fuzzy controller with the updated rule base outperformed the Phat controller in achieving and maintaining the data transfer at a maximum data rate (as limited by the acceptable packet loss). To reduce the oscillation about the maximum data rate (when packet loss occurs) the control fuzzy set may be extended to increase the control resolution about the ONE (no change) label.

With the basic transfer mechanism and learning algorithms in place on line tuning was performed. The results showed a 2-3% improvement over transferring various sized files, and with varying degree of packets loss. However the results also illustrated that the experiments with tuning were ill designed. Most importantly the method of packet loss did not properly recreate a form of congestion, instead the loss was unrelated to data rate. Secondly the Phat control variable (a delay multiplier) is not the actual desired target of the VTP control scheme. In addition the actual tuning (in-situ supervised training) was not well adapted to the problem with no known optimal target value.

The actual improvement seems to come from another problem in the actual Phat server, which can result in repeated loss of packets during re-transmission. The tuning appeared to actually minimize this loss, illustrated by the increased improvements with the higher packet drop rates.

The last tests were used to illustrate the functioning of the hierarchical VTP control. The global control was found to perform as desired, first re-allocating bandwidth from a connection with low bandwidth use to the connections with high bandwidth uses, within the confines of the maximum available bandwidth. Then using an additional priority parameter the bandwidth was similarly re-allocated but from low to high with higher priority being service first. It should be stressed that this was unused bandwidth being re-allocated. It is also desirous for the global controller to actually re-allocate used bandwidth (thus reducing the actual bandwidth usage of existing connections) in order not to choke off new connections when little or no bandwidth is left un-allocated or unused.

6.3 Future Work

There are many areas in which future work could continue what has been started in this thesis.

The main limitation in this thesis was not having a working transport protocol, but relying on the Phat application to emulate specific aspects of the protocol. The primary focus of future work should be to implement VTP as a transport protocol (as opposed to emulating it with an application over UDP). Once developed this may be used in either a network simulator or over actual test networks to properly compare VTP with TCP, UDP, or other transport protocols, as well as to properly evaluate different control architectures.

6.3.1 VTP

Specifically with respect to VTP, the following points highlight key aspects to continue the work started here:

- Implement VTP as a transport protocol, rather then emulating it over UDP.
- Investigate other error metrics.
- Protocol offload engines.

6.3.2 Computational Intelligence

The first questions that came to mind while performing the test transfers was how and what can we properly learn in this environment. The reason for using in-situ learning was to allow individual flows to react to individual network conditions through observed traffic characteristics. This leads to many potential alternatives:

- Use unsupervised learning to try and classify various network conditions offline, then try to detect them on-line.
- Use pre-determined responses to different classes, with the weighting reflecting the degree of belonging to a class.
- Use greater resolution around the neutral control point in the fuzzy control set.

VII. References

[1] Postel, J., "Internet Protocol - DARPA Internet Program Protocol Specification", STD 5, RFC 791, DARPA, Sept. 1, 1981.

[2] Postel, J., "Transmission Control Protocol - DARPA Internet Program Protocol Specificationl", STD 7, RFC 793, USC/Information Sciences Institute, Sept. 1, 1981.

[3] Postel, J., "User Datagram Protocol", STD 6, RFC 768, USC/Information Sciences Institute, Aug. 28, 1980.

[4] Tanenbaum, Andrew S., **Computer Networks 3'rd Edition**, Prentice Hall, 1996.

[5] Stallings, William, **Data and Computer Communications 6'th Edition**, Prentice Hall, 2000.

[6] Mockapetris, P. V., "Domain names - implementation and specification", STD 13, RFC 1035, Nov. 1, 1987.

[7] Hedrick, C. L., "Routing Information Protocol", RFC 1058, June 1, 1988.

[8] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, March 1997.

[9] Sollins, K., "The TFTP Protocol (Revision 2)", STD 33, RFC 1350, July 1992.

[10] Shepler, S., B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, D. Noveck, "NFS version 4 Protocol", RFC 3010, December 2000.

[11] **Maximum Security, A Hacker's Guide to Protecting Your Internet Site and Network First Edition**, Sams.net Publishing, 1997.

[12] Bellovin, S. M., "Security Problems in the TCP/IP Protocol Suite", Computer Communication Review, Vol. 19, No. 2, pp. 32-48, April 1989. URL: http://www.deter.com/unix/papers/tcpip_problems_bellovin.pdf.

[13] Farrow, R., "Sequence Number Attacks", Unix World. URL: http://www.networkcomputing.com/unixworld/security/001.txt.html

[14] Jacobson, V. and Braden, R. T., "TCP Extensions for Long-Delay Paths", RFC 1072, Oct. 1, 1988.

[15] Allman, Mark, Hans Kruse, Shawn Ostermann, "A History of the Improvement of Internet Protocols Over Satellites Using ACTS", Invited paper for ACTS Conference 2000, May 2000.

[16] Clark, David D., "Window and Acknowledgement Strategy in TCP", RFC 813, MIT Laboratory for Computer Science Computer Systems and Communications Group, July 1, 1982.

[17] Velten, D., Hinden, R.M., and Sax, J., "Reliable Data Protocol",RFC 908, BBN Communications Corporation, July 1, 1984.

[18] Partridge, C., Hinden, R. M., "Version 2 of the Reliable Data Protocol (RDP)",RFC 1151, BBN, April 1, 1990.

[19] Clark, D. D., Lambert, M. L., Zhang, L., "NETBLT: A Bulk Data Transfer Protocol", RFC 998, MIT Laboratory for Computer Science, March 1, 1987.

[20] Allman, Mark, Shawn Ostermann, "Multiple Data Connection FTP Extensions", Technical Report TR-19971, Ohio University Computer Science, February 1997.

[21] Mears, Jennifer, "Caching vendors take on last mile", Network World (http://www.nwfusion.com/), March 4, 2002.

[22] The IRCache project - NLANR's Global Caching Hierarchy, The National Laboratory for Applied Network Research (http://www.nlanr.net/)

[23] Cheriton, D. R., "VMTP: Versatile Message Transaction Protocol: Protocol Specification", RFC 1045, Stanford University, Feb. 1, 1988.

[24] Postel, J., "TCP maximum segment size and related topics", RFC 879, Nov. 1, 1983.

[25] Nagle, John, "Congestion control in IP/TCP Internetworks", RFC 896, Ford Aerospace and Communications Corporation, Jan. 6, 1984.

[26] Fox, R., "TCP Big Window and NAK Options", RFC 1106, Tandem, June 1, 1989.

[27] McKenzie, A. M., "Problem with the TCP Big Window Option", RFC 1110, BBN STC, Aug. 1, 1989.

[28] O'Malley, S., Peterson, L. L., "TCP Extensions Considered Harmful", RFC 1263, University of Arizona, Oct. 1, 1991.

[29] Jacobson, V., Braden, R., Borman, D., "TCP Extensions for High Performance", RFC 1323, May 1992.

[30] Connolly, T., Amer, P., Conrad, P., "An Extension to TCP : Partial Order Service", RFC 1693, University of Delaware, November 1994.

[31] Mathis, M., Mahdavi, J., Floyd, S., Romanow, A., "TCP Selective Acknowledgement Options", RFC 2018, October 1996.

[32] Allman, M., Floyd, S., Partridge, C., "Increasing TCP's Initial Window", RFC 2414, September 1998.

[33] Poduri, K., Nichols, K., "Simulation Studies of Increased Initial TCP Window Size", RFC 2415, Bay Networks, September 1998.

[34] Floyd, S., Henderson, T., "The New Reno Modification to TCP's Fast Recovery Algorithm", RFC 2582, April 1999.

[35] Shepard, T., Partridge, C., "When TCP Starts Up With Four Packets Into Only Three Buffers", RFC 2416, BBN Technologies, September 1998.

[36] Allman, M., Glover, D., Sanchez, L., "Enhancing TCP Over Satellite Channels using Standard Mechanisms", RFC 2488, January 1999.

[37] Paxson, V., Allman, M., Dawson, S., Fenner, W., Griner, J., Heavens, I., Lahey, K., Semke, J., Volz, B., "Known TCP Implementation Problems", RFC 2525, March 1999.

[38] Allman, M., Dawkins, S., Glover, D., Griner, J., Tran, D., Henderson, T., Heidemann, J., Touch, J., Kruse, H., Ostermann, S., Scott, K., Semke, J., "Ongoing TCP Research Related to Satellites", RFC 2760, February 2000.

[39] Allman, M., Paxson, V., Stevens, W., "TCP Congestion Control", RFC 2581, April 1999.

[40] Handley, M., Padhye, J., Floyd, S., "TCP Congestion Window Validation", RFC 2861, ACIRI, June 2000.

[41] Floyd, S., Mahdavi, J., Mathis, M., Podolsky, M., "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, July 2000.

[42] Lahey, K., "TCP Problems with Path MTU Discovery", RFC 2923, dotRocket Inc., September 2000.

[43] Paxson, V., Allman, M., "Computing TCP's Retransmission Timer", RFC 2988, November 2000.

[44] Allman, M., Balakrishnan, H., Floyd, S., "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, January 2001.

[45] Ramakrishnan, K., Floyd, S., Black, D., "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

[46] Kent, S., Atkinson, R., "Security Architecture for the Internet Protocol", RFC 2401, November, 1998.

[47] Deering, S., Hinden, R., "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[48] Pedrycz, W. and A. V. Vasilakos, "Computational Intelligence: A Development Environment for Telecommunication Networks", **Computational Intelligence in Telecommunication Networks**, CRC Press, 2000.

[49] Czezowski, Peter J. , "Applications of Computational Intelligence in ATM Networks", Candidacy Thesis, University of Manitoba, June 1998.

[50] Pedrycz, W.,"Fuzzy neural networks and neurocomputations," *Fuzzy Sets and Systems*, 56, pp. 1-28, 1993.

[51] Pedrycz, W. and A.F. Rocha, "Fuzzy-set based models of neurons and knowledge-based networks," *IEEE Trans. on Fuzzy Systems*, vol. 1, no. 5, pp. 254-266, 1993.

[52] Zadeh, L. A., "Fuzzy Sets", Information and Control, vol. 8, pp. 338-353, 1965.

[53] Pedrycz, Witold, **Fuzzy Control and Fuzzy Systems second, extended, edition**, John Wiley & Sons Inc., 1993.

[54] Pedrycz, Witold, **Fuzzy Sets Engineering**, CRC Press, Boca Raton, FL, 1995.

[55] Kosko, Bart, **Fuzzy Sets Engineering**, Prentice Hall, 1997.

[56] Pedrycz, W., C. H. Poskar and P. J. Czezowski, "A reconfigurable fuzzy neural network with in-situ learning", *IEEE Micro*, vol. 15, no. 4, pp. 19-30, August 1995.

[57] Poskar, C. H., P. J. Czezowski and W. Pedrycz, "Hardware Realization of Fuzzy Neural Networks", **Fuzzy Hardware: Architectures and Applications**, A. Kandel and G. Langholz (eds.), Kluwer Academic Press, pp. 43-76, 1998.

[58] Poskar, C. H., P. J. Czezowski and R. D. McLeod, "A Computational Intelligence Based Course-Grained Reconfigurable Element", ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 99), Monterey, California, U.S., 1999.

[59] Rumelhart, David E., McClelland, James L., et al, **Parallel Distributed Processing**, Vols 1 and 2, The MIT Press, 1986.

[60] Holland, J.H., "Adaptation in Natural and Artificial Systems", Ann Arbor: University of Michigan Press, 1975.

[61] Brown, Andrew D., Selecting Feature Detectors for Artificial Neural Networks Using Genetic Algorithms , Masters Thesis, University of Manitoba, 1997.

[62] URL: http://www.phatpackets.com

[63] Noghani, B., Kretschmann, S., and McLeod, R.D., Reducing Latency on the Internet using Component-Based Download and File-Segment Transfer Protocol: Experimental Results , Spects 2000, The proceedings of the 2000 Symposium on Performance Evaluation of Computer and Telecommunication Systems, July 2000.

[64] Floyd, S., Mahdavi, J., Mathis, M., Podolsky, M., "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, July 2000.

[65] Whelan, Eoin and Tom Doris, "The Brain Construction Kit Technical Manual", 1999. http://www.compapp.dcu.ie/~tdoris/BCK/

[66] Postel, J.,Reynolds, J., "File Transfer Protocol (FTP)", RFC 959, October 1985.

[67] Berenji, H.R. and P. Khedkar, “Learning and tuning fuzzy logic controllers through reinforcements,” *IEEE Trans. on Neural Networks*, vol. 3, no. 5, pp. 724-740, 1992.
[68] Eklund, P. and F. Klawonn, “Neural fuzzy logic programming,” *IEEE Trans. on Neural Networks*, vol. 3, no. 5, pp. 815-818, 1992.
[69] Horikawa, S., T. Furuhashi and Y. Uchikawa, “On fuzzy modeling using fuzzy neural networks with the back-propagation algorithm,” *IEEE Trans. on Neural Networks*, vol. 3, no. 5, pp. 801-806, 1992
[70] Jang, J.R., “Self-learning fuzzy controllers based on temporal back propagation,” *IEEE Trans. on Neural Networks*, vol. 3, no. 5, pp. 714-723, 1992.
[71] Zhang, Y. and A. Kandel, “Compensatory Neurofuzzy Systems with Fast Learning Algorithms,” *IEEE Trans. on Neural Networks*, vol. 9, no. 1, pp. 83-105, 1998.
[72] Gupta, M.M. and J. Qi, “Synaptic and Somatic Learning and Adaptation in Fuzzy Neural Systems,” IEEE
[73] **The Net100 Project**, URL: http://www.net100.org
[74] Butnariu, D. and E.P. Klement, *Triangular Norm-based Measures and Games with Fuzzy Coalitions*, Kluwer Academic Publishers, Dordercht, 1993.
[75] Card, H. “Digital VLSI backpropagation networks”, *Canadian J. Elect. & Comp. Eng.*, vol. 20, no. 1, pp. 15-23, 1995
[76] Clarckson, T.G., C.K. Ng and Y. Guan, “The pRAM: an adaptive VLSI chip,” *IEEE Trans. on Neural Networks*, vol. 4, no. 3, pp. 408-411, 1993.
[77] Dickson, J., R. McLeod and H.Card, “Stochastic arithmetic implementations of artificial neural networks with in-situ learning”, *Proc. IEEE Int. Conf. Neural Networks*, (San Francisco, CA), pp. 711-716, 1993.
[78] Hamilton, A. *et al*, “Integrated pulse stream neural networks: results, issues, and pointers,” *IEEE Trans. on Neural Networks*, vol. 3, no. 3, pp. 385-393, 1992.
[79] Lehmann, C., M. Viredaz and F. Blayo, “A generic systolic array building block for neural networks with on-chip learning,” *IEEE Trans. on Neural Networks*, vol. 4, no. 3, pp. 400-407, 1993.
[80] Manduit, M. *et al*, “Lneuro 1.0: a piece of hardware LEGO for building neural network systems,” *IEEE Trans. on Neural Networks*, vol. 3, no. 3, pp. 414-421, 1992.
[81] Tomlinson Jr., M.S., D.J. Walker, and A. Sivilotti, “A digital neural network architecture for VLSI”, *Proc. Int. Joint Conf. Neural Networks*, (San Diego, CA), vol. 2, pp. 545-550, 1990.
[82] Watanabe, T. *et al*, “A single 1.5-V digital chip for a 10^6 synapse neural network,” *IEEE Trans. on Neural Networks*, vol. 4, no. 3, pp. 387-393, 1993.
[83] Wawrzynek, J., K. Asanovic and N. Morgan, “The design of a neuro-microprocessor,” *IEEE Trans. on Neural Networks*, vol. 4, no. 3, pp. 394-399, 1993.
[84] Horst, R.W., “Task-flow architecture for WSI parallel processing,” *Computer*, vol. 25, no. 4, pp. 10-18, 1992.

Appendix A: The main Phat classes with modifications to emulate VTP with Fuzzy Neural Control:

The following sections of this appendix contain the source code for the Phat classes that were modified to emulate VTP and support Fuzzy Neural Networks. These comprise the three main classes used by the Phat client:

Appendix A.1 - FSTPController.java

- the main controller class, containing the class MultiConnectionHandler, which provides the global control over all concurrent Phat connections.

Appendix A.2 - PhatClient.java

- the control class for each client connection - used to implement the FNN for the local control of each connection.

Appendix A.3 - FSTPClient.java

- the basic client class for receiving and handling incoming packets for each segment - Phat uses this class to determine when packet loss should be evaluated.

The original Phat code is available from:

`http://www.phatpackets.com/`

A.1 FSTPController.java

```
package com.phatpackets.fstp.client;

import com.phatpackets.fstp.*;
import com.phatpackets.fstp.client.BitSetPacketTracker;
import com.phatpackets.fstp.util.ScreenOutputStream;
import com.phatpackets.util.Guard;

import java.net.*;   //fix
import java.io.*;
import java.util.NoSuchElementException;
import java.util.StringTokenizer;
import java.util.Vector;

/**
 * Title:
 * Description:
 * Copyright: Copyright (c) 2001
 * Company:
 * @author Greg Jaman, Chris Buzunis
 * @version 1.0
 * @author modified by C. H. Poskar, May 2002
 * @version support for Bandwidth Broker
 */

public class FSTPController implements CommandConstants {

  private PhatClient controlm;
  private long fileSize;
  private long lastSeqId;
  public ScreenOutputStream outs =new ScreenOutputStream(System.out);

  private static int conIDS=0;
  public static PacketTracker packetTracker;
  public static int numConnections = 1;
  private Vector mServers;
  public static int serverIndex = 1;

  private Vector connections;
  private String host;

  private long startTime;
  public Vector phatData;
  private int adjustSize;

  //for the sendm stuff
  private long offSet = 0;
  private long firstBlock = 0;

  DataStorage dataStore;
  private StatsListener statsListener;

  // not sure about these
  private long currAdjustSeqEnd;   // Inclusive
  private int currRcvSize;

  // Ignore some transients
  private boolean firstTime = true;

  public FSTPController(PhatClient control) {
    this.controlm = control;
    this.host = control.host;
    connections = new Vector();
    connections.add(new multiConnectionHandler(controlm));
  }
  public FSTPController(PhatClient control,long fileSize, int numCon, Vector
                                                                   mServers) {
    this.controlm = control;
    this.fileSize= fileSize;
    this.host = control.host;
    this.numConnections=numCon;
    this.mServers= mServers;
    connections = new Vector();
    connections.add(new multiConnectionHandler(controlm));
  }

  public synchronized void getFile(String filename) throws IOException{
    buildControls();
    final String fileName = filename;
    //how about set the ports in teh buildcontrols?
    if(!setPorts())
      return;
    if(!setDelay())
      return;
    //**do the stuff to set up the file
    try {
      File file = new File(fileName+"M");//added the M for now
      dataStore = new RandomAccessStorage(new File(fileName+"M")); //added the M
    } catch (FileNotFoundException ex) {
      System.out.println("phat: File \'" + filename + "\' cannot be written to.");
      return;
    }
    //end of file set up
    phatData = new Vector();//do this somewhere else??
    for( int i=0; i<connections.size(); i++){
      final int j = i;
      final PhatClient control = ((multiConnectionHandler)
                                          connections.elementAt(i)).control;
      final FSTPClient client = ((multiConnectionHandler)
                                          connections.elementAt(i)).client;
      final multiConnectionHandler conH = (multiConnectionHandler)
                                          connections.elementAt(i);

      try{
        sendSENDMCmd(fileName,conH);
        control.waitForReply();
        String rply = control.rplyQ.pop();
        if (!rply.startsWith(FILE_INFO_RPLY)) return;
        conH.tok = new StringTokenizer(rply);
        conH.tok.nextToken();
        conH.tok.nextToken(); // FILESIZE
        client.setFileSize(Long.parseLong(conH.tok.nextToken()));
        conH.tok.nextToken(); // STREAMID
        client.setStreamId(Integer.parseInt(conH.tok.nextToken()));
        conH.tok.nextToken(); // SERVERPORT
        client.connect(control.socket.getInetAddress(), Integer.parseInt(
                                                       conH.tok.nextToken()));
        if(j==0){
        //only need one packettracker!
```

```
          createPacketTracker(client);
        }
        if(j==connections.size()-1){
          outs.clearScreen();
          outs.goTo(2,1);
          outs.print(" running... ");
        }
        startTime = System.currentTimeMillis();
        new Thread(new Runnable() {
          public void run() {
           try{
            while(sendBoundCmd(fileName,conH)){
              conH.guard.resetGuard();
              client.receiveDataM(phatData,conH.miniPacketTracker,dataStore,conH.conId);
              conH.subBlockCount++;
              conH.guard.waitGuard();
              fireWriteToFile();
            }
           }catch(IOException e){}
           }
        }).start();
      } catch (NoSuchElementException ex) {
        System.out.println("phat: Invalid server reply.");
        return;
      }
      catch(Exception e){
        System.out.println("found an error! ");
      }
    }

    try{
      wait();
      //*this seems like a safe spot to close all the connections
      for( int i=0; i<connections.size(); i++){
        multiConnectionHandler conH = (multiConnectionHandler)connections.elementAt(i);
        conH.client.doneRcv();
        conH.client.close();
      }
    }catch(InterruptedException e){}
  }

  private boolean connectionsActive(){
    for(int i=0; i<connections.size(); i++){
     if(!(offSet>=lastSeqId)) return true;
      if( ((multiConnectionHandler)connections.elementAt(i)).miniPacketTracker!=null){
       if( !((multiConnectionHandler)
                                connections.elementAt(i)).miniPacketTracker.isComplete())
        return true;
      }
    }
    return false;
  }

    public synchronized void fireWriteToFile(){
    for(int i=0; i<connections.size(); i++){
    if(!(offSet>=lastSeqId)) return;
      if( ((multiConnectionHandler)connections.elementAt(i)).miniPacketTracker!=null){
       if(!((multiConnectionHandler)
                                connections.elementAt(i)).miniPacketTracker.isComplete())
        return;
        }
      }

      long time = System.currentTimeMillis() - startTime;
      try{
       Thread.sleep(1000); //wait a second before we write??
      }
      catch(Exception e)
      {}
      try{
      OutputStream outf;
      outf = new BufferedOutputStream(new FileOutputStream("results.txt"));
      String toWrite = new String();
      toWrite = "Number connections : " + numConnections+" \n ";
      outf.write(toWrite.getBytes());
      toWrite ="phat: DONE - " + fileSize + " bytes received in " + time/1000.f + "s ("
              + (float)((double)fileSize / (double)time / 1.024) + " kB/s).\n";
      outs.goTo(2,numConnections+7);
      outs.print(toWrite);
      outf.write(toWrite.getBytes());
      outs.println("\t\tReporter: ");
      multiConnectionHandler tempCH;
      for(int j=0; j<connections.size(); j++){
        tempCH = ((multiConnectionHandler)connections.elementAt(j));
        toWrite = " ["+tempCH.conId+" "+ tempCH.thisHost+"] - num subBlocks: "
                + tempCH.subBlockCount + "\n";
        outs.print(toWrite);
        outf.write(toWrite.getBytes());
      }
      outf.close();
      }catch(IOException e){}
      // this does nothing!!! the writing is done when the packet is processed!
      //but later we will want it here!
      //write to data was for when we had the idea to write to the file after increment
      //of the file instead of at one time.
      //this should be placed after we get a subblock?
      writeData(phatData);
      notify();
  }

  public void writeData(Vector phatData){
  }

   //later give this method a array of hosts for the creating of the sockets.
   //private void buildControls(Guard guard) throws IOException{
   private void buildControls() throws IOException{
     for(int i=1; i<numConnections; i++){
       connections.add(new multiConnectionHandler());
     }
   }

   private boolean setPorts() throws IOException{
     for(int i=0; i<connections.size(); i++){
       PhatClient control = ((multiConnectionHandler)
                                           connections.elementAt(i)).control;
       FSTPClient client = ((multiConnectionHandler)connections.elementAt(i)).client;
       control.cmdBuf.setLength(0);
       control.cmdBuf.append(PORT_CMD);
       control.cmdBuf.append(' ');
       control.cmdBuf.append(client.getLocalPort());
```

```
        control.sendMessage(control.cmdBuf.toString());
        control.waitForReply();
        if (!control.rplyQ.pop().startsWith(COMMAND_OKAY_RPLY)) return false;
      }
      return true;
    }
    private boolean setDelay() throws IOException{
        for(int i=0; i<connections.size(); i++){
        PhatClient control = ((multiConnectionHandler)connections.elementAt(i)).control;
        control.cmdBuf.setLength(0);
        control.cmdBuf.append(BLOCK_DELAY_CMD);
        control.cmdBuf.append(' ');
        control.cmdBuf.append(control.blockDelay);
        control.sendMessage(control.cmdBuf.toString());
        control.waitForReply();
        control.rplyQ.pop(); //put in if to check reply
      }
      return true;
    }

    privateboolean sendSENDMCommand(String filename) throws IOException{
      long offSet = 0;
      long firstBlock = 0;
      for(int i=0; i<connections.size(); i++){
        multiConnectionHandler mh = (multiConnectionHandler)connections.elementAt(i);
        PhatClient control = mh.control;
        control.cmdBuf.setLength(0);
        control.cmdBuf.append(SENDM_CMD);
        control.cmdBuf.append(' ');
        if(offSet==0)
          firstBlock = offSet;
        else firstBlock = offSet +1;
        control.cmdBuf.append(firstBlock);
        control.cmdBuf.append(' ');
        offSet = firstBlock + (lastSeqId/numConnections);
        if(numConnections ==i) offSet=lastSeqId +1; //??
        control.cmdBuf.append(offSet);
        control.cmdBuf.append(' ');
        control.cmdBuf.append(filename);
        control.sendMessage(control.cmdBuf.toString());
        //set the miniPacketTracker
        mh.miniPacketTracker = new BitSetPacketTracker();
        mh.miniPacketTracker.setExpectedPacketCount((offSet-firstBlock)+1);
        mh.miniPacketTracker.setOffset(firstBlock);
        control.waitForReply();
        if (!control.rplyQ.pop().startsWith(SENDING_FILE_RPLY)) return false;
      }
      return true;
    }

    private synchronized boolean sendSENDMCmd(String filename,
                                  multiConnectionHandler mh) throws IOException{
      PhatClient control = mh.control;
      control.cmdBuf.setLength(0);
      control.cmdBuf.append(SENDM_CMD);
      control.cmdBuf.append(' ');
      control.cmdBuf.append(filename);
      control.sendMessage(control.cmdBuf.toString());
    control.waitForReply();
    if (!control.rplyQ.pop().startsWith(SENDING_FILE_RPLY)) return false;
    return true;
    }
  private synchronized boolean sendBoundCmd(String filename,
                              multiConnectionHandler mh) throws IOException{
    PhatClient control = mh.control;
    if(offSet>=lastSeqId) return false;
    control.cmdBuf.setLength(0);
    control.cmdBuf.append(SENDMB_CMD);
    control.cmdBuf.append(' ');
    control.cmdBuf.append(mh.client.getStreamId());
    control.cmdBuf.append(' ');
    if(offSet==0)
    firstBlock = offSet;
    else firstBlock = offSet +1;
    control.cmdBuf.append(firstBlock);
    control.cmdBuf.append(' ');
    //offSet = firstBlock + (lastSeqId/numConnections);
    offSet = firstBlock + controlm.adjustSubBlockSize;
    //if(numConnections ==i) offSet=lastSeqId +1; //??
    if(offSet>lastSeqId) offSet=lastSeqId;
    control.cmdBuf.append(offSet); //? know more!
    control.sendMessage(control.cmdBuf.toString());
      //set the miniPacketTracker
    mh.miniPacketTracker = new BitSetPacketTracker();
    mh.miniPacketTracker.setExpectedPacketCount((offSet-firstBlock)+1);
    mh.miniPacketTracker.setOffset(firstBlock);

    return true;
  }
  private void createPacketTracker(FSTPClient theClient) {
    currAdjustSeqEnd = adjustSize - 1;
    firstTime = true;
    int streamIdSize = PhatPacket.calcStreamIdSize(theClient.getStreamId());
    int seqIdSize = PhatPacket.calcSeqIdSize(theClient.fileSize,
                                   theClient.getPacketSize(), streamIdSize);
    int packetDataSize = theClient.getPacketSize() - streamIdSize - seqIdSize;
    if (dataStore.isPacketDataSizeRequired()) {
      dataStore.setPacketDataSize(packetDataSize);
      dataStore.setTotalSize(theClient.fileSize);
    }
    long expPackets = (long)Math.ceil((double)theClient.fileSize /
                                                 (double)packetDataSize);
    packetTracker = new BitSetPacketTracker();
    packetTracker.setExpectedPacketCount(expPackets);
    if (statsListener != null) {
      statsListener.setExpectedPacketCount(expPackets);
    }
  }
  public interface StatsListener {
    public void setExpectedPacketCount(long count);
    public void packetReceived(long seqId);
    public void startResend();
    public void receiveTimeout();
  }

  private class multiConnectionHandler{
```

```
PhatClient control;
FSTPClient client;
boolean    done=false;
int        connectionID;
int        tagSize;
String     thisHost;
StringTokenizer tok;
final  Guard          guard = new Guard();
public PacketTracker miniPacketTracker;
public int           conId;
public int           subBlockCount=0;

public multiConnectionHandler() throws IOException{
  if(serverIndex==mServers.size()) serverIndex=0;
  thisHost = (String)(mServers.elementAt(serverIndex));
  Socket server = new Socket(thisHost, PhatClient.PHAT_PORT);
  serverIndex++;
  control = new PhatClient(server,true);
  buildClient();
  setConnectionValues();
  addClientListerners();
  conId=conIDS++;
  System.out.println("Connection: "+conId+ " is to " + thisHost);

  //TODO: set the values like alpha, block size - have to obtain these
  //from the main connection otherwiseit will use the defaults
}

public multiConnectionHandler(PhatClient con){
 control = con;
 thisHost=con.host;
 buildClient();
 setConnectionValues();
 addClientListerners();
 int streamIdSize = PhatPacket.calcStreamIdSize(client.getStreamId());
 int seqIdSize = PhatPacket.calcSeqIdSize(fileSize, client.getPacketSize(),
                                                        streamIdSize);
 tagSize = streamIdSize + seqIdSize;
 lastSeqId = (long)Math.ceil((double)fileSize/(double)(client.getPacketSize() -
                                                        tagSize)) - 1;
 conId=conIDS++;
}

private void buildClient(){
  try{
    client = new FSTPClient(control);
  }catch(SocketException e){
    System.out.println(" socket exception ");
  }
}

private void setConnectionValues(){
  //set this all up same as the originals
  control.blockDelay = controlm.blockDelay;
  client.setAlpha(controlm.alpha);
  client.setAdjustSize(controlm.adjustSize);
  client.setPacketSize(controlm.packetSize);
  client.setDynamicBlockSize(controlm.dynBlockSize);
  client.setStartBlockSize(controlm.startBlockSize);
  client.setDynamicBlockDelay(controlm.dynBlockDelay);
```

```
        }
      private void addClientListerners(){
        client.addClientListener(new FSTPClient.ClientListener() {
          public void receiveComplete(int streamId, long time) {
            long subFileSize = (offSet-firstBlock) * (client.getPacketSize()-tagSize);
            outs.goTo(2,conId+4);
            outs.clearLine();
            outs.print("[ "+conId+","+thisHost+"]; Block Count: "+ subBlockCount+";
                      false ; Speed of last block: "+(float)((double)subFileSize /
                      (double)time / 1.024)+"kB/s; ");
            guard.notifyGuard();
          }
          public void receiveTimeout(int streamId, float percentComplete, long time) {
            outs.goTo(2,conId+4);
            outs.clearLine();
            outs.print("[ "+conId+","+thisHost+"]; Block Count: "+ subBlockCount+";
                      TIMEOUT ; %: "+ client.getPercentComplete());
          }
          public void clientException(int streamId, Exception ex) {
            //System.out.println("phat: EX Error while receiving: " + ex);
            outs.goTo(2,conId+4);
            outs.clearLine();
            outs.print("phat: EX Error while receiving: " + ex);
            guard.notifyGuard();
          }
          public void receiveStopped(int streamId, long time) {
            outs.goTo(2,(int)conId+4);
            outs.clearLine();
            outs.print("[ "+conId+","+thisHost+"]; Block Count: "+ subBlockCount+";
                      receiveStopped ; %: "+ client.getPercentComplete());
            guard.notifyGuard();
          }
          public void blockSizeAdjusted(int streamId, int newBlockSize) {
          }
        });
        }
    }
}
```

A.2 PhatClient.java

```
package com.phatpackets.fstp.client;

import com.phatpackets.fstp.*;
import com.phatpackets.util.Guard;
import com.phatpackets.util.StringQueue;

import BCK.ANN.BCKFuzzyNeuron;
import BCK.ANN.BCKFNNBprop;
import BCK.ANN.BCKRecord;

import java.io.*;
import java.net.*;
import java.util.StringTokenizer;
import java.util.Vector;
import java.util.NoSuchElementException;

/**
 * Title:
 * Description:
 * Copyright:Copyright (c) 2001
 * Company:
 * @author Shawn Silverman, modified by Greg Jaman, Chris Buzunis
 * @version 1.0
 * @author modified by Greg Jaman, Chris Buzunis
 * @version added multi-connection / multi-server
 * @author modified by C. H. Poskar, May 2002
 * @version added Neural controller / Bandwidth Broker
 */

/**
 * Modified May, 2002
 * C. H. Poskar
 *
 * Added support for VTP Emulation and Fuzzy Neural Controller:
 *  - fnn, the fuzzy neural network,
 *  - numIONeurons, the number of Inupt and Ouptu neurons,
 *  - packetLoss, deltaPacketLoss, and control, membership functions for fuzzy sets
 *  - buildFNN(), construct and initialize the fuzzy neural controller,
 *  - buildPacketLossMF(), construct the packet loss membership function ,
 *  - buildDeltaPacketLossMF(), construct the delta packet loss membership function,
 *  - buildControlMF(), construct the control membership function,
 *  - fuzzify(), fuzzify a crisp input with the specified membership functions,
 *  - defuzzify(), defuzzify a fuzzy output using COG,
 *  - fuzzyControl(), determine the control by processing a set of inputs,
 *  - fuzzyTrain(), train the FNN with the given training pattern,
 *  - getBlockDelay(), which returns the initial block delay setting, and
 *  - sendBlockDelay(), sends a new block delay setting to the server.
 *
 * TODO:
 * Implement different de-fuzzification schemes.
 */

/*
 * AbstractHandler:
 * - appears to be a thread that processes commands
 * CommandConstants:
 * - has access to all the same commands and values as the server.
 */
```

```
public class PhatClient extends AbstractHandler implements CommandConstants {
  public static final int PHAT_PORT = 4269;//changed to public

// Some client parameters
  float alpha             = 0.95f;
  int packetSize          = 1472;
  int adjustSize          = 50;    //was 100
  int blockDelay          = 50;    //start with this value..
  int adjustSubBlockSize = 250;
  int startBlockSize      = 4;
  boolean dynBlockSize    = false;
  boolean dynBlockDelay   = false;
  boolean isSilent        = false;
  boolean logging;
  public String host;

// Internal auxiliary storage
  /** A storage space for Strings. */
  StringBuffer cmdBuf     = new StringBuffer();
  StringQueue rplyQ       = new StringQueue();
  public Vector mServers = new Vector();

// Fuzzy Neural Controller & I/O Records:
  public BCKFNNBprop fnn   = null;
  public int          numIONeurons;
  public double[][]  packetLoss;
  public double[][]  deltaPacketLoss;
  public double[][]  control;

  /**
   * Regular constructor for a PhatClient object.
   */
  public PhatClient(Socket s) throws IOException {
    super(s);
    this.host = s.getInetAddress().getHostName();
    mServers.addElement(this.host);

    //for debuggin ---
    //logging=true;

    // Start up the server listener and wait for the acceptance reply
    new Thread(this).start();
    waitForReply();
    rplyQ.pop();

    // Do commands
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    while (true) {
      System.out.print("phat> ");
      String line = in.readLine().trim();
      StringTokenizer tok = new StringTokenizer(line);
      if (!tok.hasMoreTokens()) continue;

      // Get the command
      String cmd = tok.nextToken();

      // This block of code should be replaced by a command pattern
      try {
        if (cmd.equals(SEND_CMD)) {
          sendSend(tok);
        } else if (cmd.equals(BLOCK_SIZE_CMD)) {
          sendBlockSize(tok);
        } else if (cmd.equals(BLOCK_DELAY_CMD)) {
```

```
    sendBlockDelay(tok);
  } else if (cmd.equals(PACKET_SIZE_CMD)) {
    sendPacketSize(tok);
  } else if (cmd.equals(LIST_CMD)) {
    sendList(tok);
  } else if (line.equals(QUIT_CMD)) {
    sendMessage(line);
    rplyQ.waitForItem(2000L);
    System.exit(0);
  } else if(cmd.equals(CWD_CMD)){
    sendCWD(tok);
  } else if(cmd.equals(CDUP_CMD)){
    sendCDUP(tok);
  } else if(cmd.equals(ADDSERVER_CMD)){
    String mHost = tok.nextToken();
    mServers.addElement(mHost);
    System.out.println("phat: Added host " + mHost + ".");
  } else if(cmd.equals(SHOWSERVER_CMD)){
    System.out.println("phat:  Hosts: ");
    for(int j=0; j<mServers.size(); j++)
      System.out.println("\t" + (String)(mServers.elementAt(j)));
  } else if(cmd.equals(MULTISEND_CMD) || cmd.equals("mgetm")){
    System.out.println(" in multisend ");

    buildFNN();//FNN

    sendMulTiSend(tok);
  } else if (line.equals("KILLSERVER")) {
    sendMessage(line);
    rplyQ.waitForItem(2000L);
    System.exit(0);
  } // Client-specific commands
    else if (cmd.equals("ALPHA")) {
    float f = Float.valueOf(tok.nextToken()).floatValue();
    if (f <= 0.0f || f > 1.0f) {
      System.out.println("phat: Alpha must be in (0, 1].");
    } else {
      alpha = f;
      System.out.println("phat: Alpha set to " + f + ".");
    }
  } else if (cmd.equals("ADJUSTSIZE")) {
    int i = Integer.parseInt(tok.nextToken());
    if (i <= 0) {
      System.out.println("phat: Adjust size must be > 0.");
    } else {
      adjustSize = i;
      System.out.println("phat: Adjust size set to " + i + ".");
    }
  } else if (cmd.equals("ADJUSTSUBSIZE")) {
    int i = Integer.parseInt(tok.nextToken());
    if (i <= 0) {
      System.out.println("phat: Adjust subBlockSize must be > 0.");
    } else {
      adjustSubBlockSize = i;
      System.out.println("phat: Adjust subBlocksize set to " + i + ".");
    }
  } else if (cmd.equals("LOG")) {
    if (tok.hasMoreElements()) {
      String state = tok.nextToken();
      if ("ON".equalsIgnoreCase(state)) {
              logging = true;
            } else if ("OFF".equalsIgnoreCase(state)) {
              logging = false;
            } else {
              System.out.println("phat: LOG can be either ON or OFF.");
            }
          }
          System.out.println("phat: LOG is " + (logging ? "ON." : "OFF."));
        } else if(cmd.equals("DYNAMICBLOCKSIZE")) {
          if (tok.hasMoreElements()) {
            String state = tok.nextToken();
            if ("ON".equalsIgnoreCase(state)) {
              dynBlockSize = true;
            } else if ("OFF".equalsIgnoreCase(state)) {
              dynBlockSize = false;
            } else {
              System.out.println("phat: Dynamic block size - ON or OFF.");
            }
          }
          System.out.println("phat: Dynamic block size is " + (dynBlockSize ?
                                                            "ON." : "OFF."));
        } else if(cmd.equals("DYNAMICBLOCKDELAY") || cmd.equals("DELAY")) {
          if (tok.hasMoreElements()) {
            String state = tok.nextToken();
            if ("ON".equalsIgnoreCase(state)) {
              dynBlockDelay = true;
            } else if ("OFF".equalsIgnoreCase(state)) {
              dynBlockDelay = false;
            } else {
              System.out.println("phat: Dynamic block delay - ON or OFF.");
            }
          }
          System.out.println("phat: Dynamic block delay is " + (dynBlockDelay
                                                          ? "ON." : "OFF."));
        }
      } catch (NumberFormatException ex) {
        System.out.println("phat: Invalid number.");
      } catch (NoSuchElementException ex) {
        System.out.println("phat: Command needs more arguments.");
      }
    }
  }

  public PhatClient(Socket s,boolean inSilent) throws IOException {
    super(s);
    buildFNN();   //FNN
    this.isSilent=inSilent;
    // Start up the server listener
    new Thread(this).start();
    // Wait for the acceptance reply
    waitForReply();
    rplyQ.pop();
    //i'm connnected weeeeeee!
  }

  /** Build Fuzzy Neural Network Control Structure.
   *  Construct and initialize FNN once for each connection.
   */
```

```
private void buildFNN() {
  int numLayers = 3;
  int layer1    = 10;
  int layer2    = 25;
  int layer3    = 5;

  int numNeurons   = layer1+layer2+layer3;

  int[] layers   = { layer1, layer2, layer3 };

  int[][] refTypes = new int[numLayers][];

  double[][][] refs = new double[numLayers][][];

  int[][][] netStruct = new int[numLayers][][];

  numIONeurons = layer1+layer3;

  for (int layer=0; layer<numLayers; layer++) {
    refs[layer] = new double[layers[layer]][];
    for (int post=0; post<layers[layer]; post++) {
      refTypes[layer][post] = BCKFuzzyNeuron.NO_REF;
      refs[layer][post]     = null;
    }
  }

  for (int layer=1; layer<numLayers; layer++) {
    netStruct[layer] = new int[layers[layer]][layers[layer-1]];

    //Initialize the network to be unconnected
    for (int post=0; post<layers[layer]; post++) {
      for (int pre=0; pre<layers[layer-1]; pre++) {
        netStruct[layer][post][pre] = 0;
      }
    }

    //Manually set the connections
    if (layer == 1) {
      int post = 0;
      for (int pre1=0; pre1<5; pre1++) {
        for (int pre2=5; pre2<10; pre2++) {
          netStruct[layer][post][pre1] = 1;
          netStruct[layer][post][pre2] = 1;
          post++;
        }
      }
    } else if (layer == 2) {
      netStruct[layer][0][0] = 1;
      netStruct[layer][0][1] = 1;
      netStruct[layer][0][2] = 1;
      netStruct[layer][1][3] = 1;
      netStruct[layer][1][4] = 1;
      netStruct[layer][1][5] = 1;
      netStruct[layer][1][6] = 1;
      netStruct[layer][1][7] = 1;
      netStruct[layer][1][8] = 1;
      netStruct[layer][1][10] = 1;
      netStruct[layer][1][11] = 1;
      netStruct[layer][2][9] = 1;
      netStruct[layer][2][12] = 1;
      netStruct[layer][2][13] = 1;
      netStruct[layer][2][15] = 1;
      netStruct[layer][2][16] = 1;
      netStruct[layer][3][14] = 1;
```

```
        netStruct[layer][3][17] = 1;
        netStruct[layer][3][20] = 1;
        netStruct[layer][4][18] = 1;
        netStruct[layer][4][19] = 1;
        netStruct[layer][4][21] = 1;
        netStruct[layer][4][22] = 1;
        netStruct[layer][4][23] = 1;
        netStruct[layer][4][24] = 1;
      }
    }
    try {
      BCKFNNBprop fnn = new BCKFNNBprop(layers, netStruct, refTypes, refs);
    } catch (Exception e){
      System.err.println("Error" + e);
    }
    buildPacketLossMF();
    buildDeltaPacketLossMF();
    buildControlMF();
  }

  /** Build Membership function for Packet Loss.
   *  - packetLoss has 5 membership functions.
   *  - Each Membership function has 4 points representing the x-axis
   *  co-ordinate of the corners of a trapazoid. The y-axis co-ordinates are
   * implied as 0, 1, 1, 0.
   */
  private void buildPacketLossMF() {
    double x = (1 - alpha) / 2;

    double[][] temp = { { 0.0,   0.0,   x,     (2*x) },
                        { x,     (2*x), (2*x), (3*x) },
                        { (2*x), (3*x), (3*x), (4*x) },
                        { (3*x), (4*x), (4*x), (5*x) },
                        { (4*x), (5*x), 1.0,   1.0   }
    };

    packetLoss = temp;
  }

  /** Build Membership function for Change in Packet Loss.
   *  - deltaPacketLoss has 5 membership functions.
   *  - Each Membership function has 4 points representing the x-axis
   *  co-ordinate of the corners of a trapazoid. The y-axis co-ordinates are
   * implied as 0, 1, 1, 0.
   */
  private void buildDeltaPacketLossMF() {
    double x = (1 - alpha) / 2;

    double[][] temp = { { -(1),    -(1),   -(2*x), -(x)   },
                        { -(2*x),  -(x),   -(x),    0.0   },
                        { -(x),    0.0,    0.0,     x     },
                        { 0.0,     x,      x,       (2*x) },
                        { x,       (2*x),  (2*x),   1     }
    };

    deltaPacketLoss = temp;
  }

  /** Build Membership function for Control Output.
   *  - deltaPacketLoss has 5 membership functions.
  *  - Each Membership function has 4 points representing the x-axis
  *  co-ordinate of the corners of a trapazoid. The y-axis co-ordinates are
  * implied as 0, 1, 1, 0.
  */
 private void buildControlMF() {
   double[][] temp = { { alpha, alpha, 0.97, 1.0 },
                       { 0.99,  1.0,   1.0,  1.2 },
                       { 1.0,   1.2,   1.2,  1.4 },
                       { 1.2,   1.4,   1.4,  1.6 },
                       { 1.6,   1.8,   2.0,  2.0 }
   };

   control = temp;
 }

 /** Fuzzify an Input
  * Fuzzify an input parameter with the given membership function.
  * - Returns an array indicating the degree to which the input (x)
  * belongs to each of the membership functions (specified in mf).
  */
 private double[] fuzzify(double x, double[][] mf) {
   double[] fuzz = new double[mf.length];

   //verify Universe of Discourse:
   if ( (x < mf[0][0]) || (x > mf[mf.length-1][3]) ) {
     System.err.println("The input: " + x + " exceeded the MF bounds");
     System.exit(-1);
   }

   for(int i=0; i<mf.length; i++) {
     if ( (x>mf[i][0]) && (x<mf[i][1]) ) {
       fuzz[i] = (x - mf[i][0]) / (mf[i][1] - mf[i][0]);
     } else if ( (x>=mf[i][1]) && (x<=mf[i][2]) ) {
       fuzz[i] = 1;
     } else if ( (x>mf[i][2]) && (x<mf[i][3]) ) {
       fuzz[i] = (x - mf[i][2]) / (mf[i][2] - mf[i][3]);
     } else {
       fuzz[i] = (double)0;
     }
   }

   return fuzz;
 }

 /** De-Fuzzify an Output
  * De-fuzzify the fuzzy output with the given membership function.
  * - Returns a crisp value representing the output of the fuzzy controller.
  * - Simplified Center of Gravity, (COG).
  */
 private double defuzzify(double[] y, double[][] mf) {
   double sum1  = 0;
   double sum2  = 0;

   for(int i=0; i<y.length; i++) {
     if (y[i] > 0) {
       sum1 += y[i] * (mf[i][2] + (mf[i][3] - mf[i][2])/2);
       sum2 += y[i];
     }
   }

   return (sum1/sum2);
 }
```

```
/** Fuzzy Control Decision
 * Use the Fuzzy Neural Network to make a control decision.
 * - first fuzzify the inputs,
 * - then setup the input vector,
 * - apply the input vector to the FNN and perform a forward pass.
 * - get the output and de-fuzzify.
 */
public double fuzzyControl(double pl, double delta) {
  double[] fuzzyPL    = fuzzify(pl, packetLoss);
  double[] fuzzyDelta = fuzzify(delta, deltaPacketLoss);

  double[] input = new double[fuzzyPL.length + fuzzyDelta.length];

  for (int i=0; i < fuzzyPL.length; i++) {
    input[i] = fuzzyPL[i];
  }

  for (int i=0; i < fuzzyDelta.length; i++) {
    input[fuzzyPL.length + i] = fuzzyDelta[i];
  }

  fnn.setInput(input);

  fnn.forwardPass();

  double[] output = fnn.getOutputs();

  return defuzzify(output, control);
}

/** Train Fuzzy Neural Network - specified number of epochs.*/
public double fuzzyTrain(double[] pl,double[] delta,double[] target,int numEpochs) {
  BCKRecord[] records = new BCKRecord[pl.length];

  for (int i=0; i < records.length; i++) {
    double[] fuzzyPL     = fuzzify(pl[i], packetLoss);
    double[] fuzzyDelta  = fuzzify(delta[i], deltaPacketLoss);
    double[] fuzzyTarget = fuzzify(target[i], control);

    double[] io = new double[fuzzyPL.length+fuzzyDelta.length+fuzzyTarget.length];

    if (numIONeurons != (io.length)) {
      System.err.println("The I/O size does not match the network.");
      System.exit(-1);
    }

    int offset = 0;
    System.arraycopy(fuzzyPL, 0, io, offset, fuzzyPL.length);
    offset += fuzzyPL.length;
    System.arraycopy(fuzzyDelta, 0, io, offset, fuzzyDelta.length);
    offset += fuzzyDelta.length;
    System.arraycopy(fuzzyTarget, 0, io, offset, fuzzyTarget.length);

    records[i] = new BCKRecord(io);
  }

  double correct = -1;
  try {
    fnn.setTrainingFile(records);
    fnn.setTestingFile(records);

    fnn.train(numEpochs);

    correct = fnn.test();
  } catch (Exception e){
    System.err.println("Error" + e);
    }
    return correct;
  }

  /** Train Fuzzy Neural Network - defaults to one training epoch.*/
  public double fuzzyTrain(double[] pl, double[] delta, double[] target) {
    return fuzzyTrain(pl, delta, target, 1);
  }

  public int getBlockDelay() {
    return blockDelay;
  }

  protected void processIOError(IOException ex) {
    System.out.println("phat: There was a connection problem.");
    System.out.println("phat: Error message: " + ex);
    System.exit(0);
  }

  /**
   * Start Execution of the phat client.
   * Requires one argument - the server hostname
   */
  public static void main(String[] args) throws IOException{
    if (args.length != 1) {
      System.out.println("Usage: PhatClient <host>");
      System.exit(0);
    }
    Socket server = new Socket(args[0], PHAT_PORT);
    PhatClient client = new PhatClient(server);
  }

  /**
   * Processes a response from the server. The response will be in the
   * same format as the FTP protocol.
   */
  protected void processMsg(String msg) {
    if (msg.startsWith(PING_CMD + " ")) {
      try {
        sendMessage(PING_REPLY_CMD+" "+msg.substring(PING_CMD.length()+1));
      } catch (IOException ex) {
      }
    } else {
      rplyQ.push(msg);
      // First make sure it is not an unsolicited command
      // Hmm.. are there any of these?
      // Yes: PING
      System.out.println(msg);
    }
  }

  public void waitForReply() {
   rplyQ.waitForItem(20000L);
   if (rplyQ.isEmpty()) {
     System.out.println("phat: No reply from server for 20s.");
     System.out.println("phat: Exiting...");
     // TODO Print some stats
     System.exit(0);
   }
  }

// Commands

  /**
   * Sends the SEND command.
   */
  private void sendSend(StringTokenizer tok) throws IOException {
    String cmd = tok.nextToken("");
    String filename;
    // Get the filename first
    int quotIdx = cmd.indexOf('\"');
    if (quotIdx >= 0) {
      // Quoted?
      int nextQuotIdx = cmd.indexOf('\"', quotIdx + 1);
      if (nextQuotIdx < 0) {
        System.out.println("phat: Missing final quote character.");
        return;
      }
      filename = cmd.substring(quotIdx + 1, nextQuotIdx);
    } else {
      // Not quoted
      StringTokenizer st = new StringTokenizer(cmd);
      filename = st.nextToken();
    }
    if (filename.length() == 0) {
      System.out.println("phat: Missing filename.");
      return;
    }

    // Make sure we can create the connection first

    final FSTPClient client;
    try {
      client = new FSTPClient(this);
    } catch (SocketException ex) {
      System.out.println("phat: Error opening connection.");
      return;
    }
    // Set some client properties here so that there is
    // one less thing to do when time is important later
    client.setAlpha(this.alpha);
    client.setAdjustSize(adjustSize);
    client.setPacketSize(packetSize);
    client.setDynamicBlockSize(dynBlockSize);
    client.setDynamicBlockDelay(dynBlockDelay);
    client.setStartBlockSize(startBlockSize);
    // Add a listener
    final Guard guard = new Guard();  // For waiting
    client.addClientListener(new FSTPClient.ClientListener() {
        public void receiveComplete(int streamId, long time) {
          System.out.print("phat: " + client.getFileSize() +
                           " bytes received in " + time/1000.f + "s (");
          System.out.print((float)((double)client.getFileSize() /
                                              (double)time / 1.024));
          System.out.println(" kB/s).");
          guard.notifyGuard();
        }
        public void receiveTimeout(int streamId, float percentComplete,
                                                      long time) {
```

```
        System.out.println("phat: " +percentComplete + "% complete in "
                                                  + time/1000.0f + "s");
      }

      public void clientException(int streamId, Exception ex) {
        System.out.println("phat: Error while receiving: " + ex);
        guard.notifyGuard();
      }

      public void receiveStopped(int streamId, long time) {
        System.out.println("phat: " + client.getPercentComplete() +
                                 "% complete in " + time/1000.0f + "s");
        guard.notifyGuard();
      }

      public void blockSizeAdjusted(int streamId, int newBlockSize) {
      }
    });
  // Set the receive port
  cmdBuf.setLength(0);
  cmdBuf.append(PORT_CMD);
  cmdBuf.append(' ');
  cmdBuf.append(client.getLocalPort());
  sendMessage(cmdBuf.toString());
  waitForReply();
  if (!rplyQ.pop().startsWith(COMMAND_OKAY_RPLY)) return;
  // Send the command
  cmdBuf.setLength(0);
  cmdBuf.append(SEND_CMD);
  cmdBuf.append(' ');
  cmdBuf.append(filename);
  sendMessage(cmdBuf.toString());
  waitForReply();
  if (!rplyQ.pop().startsWith(SENDING_FILE_RPLY)) return;
  // Organize the file storage
  DataStorage dataStore;
  try {
    File file = new File(filename);
    dataStore = new RandomAccessStorage(new File(filename));
  } catch (FileNotFoundException ex) {
    System.out.println("phat: File \'" + filename +
                                            "\' cannot be written to.");
    return;
  }

  // If we are logging, make this file, too

  FileStatsListener statsListener = null;
  if (logging) {
    try {
      statsListener=new FileStatsListener(new File(filename+".log"));
    } catch (FileNotFoundException ex) {
      System.out.println("phat: File \'" + filename +
                                            "\' cannot be written to.");
      return;
    }
    try {
      client.addStatsListener(statsListener);
    } catch (java.util.TooManyListenersException ex) {
      // Shouldn't happen
    }
    }

    waitForReply();
    String rply = rplyQ.pop();
    if (!rply.startsWith(FILE_INFO_RPLY)) return;

    // The file is being sent, so get the file information

    tok = new StringTokenizer(rply);
    tok.nextToken();
    try {
      tok.nextToken(); // FILESIZE
      client.setFileSize(Long.parseLong(tok.nextToken()));
      tok.nextToken(); // STREAMID
      client.setStreamId(Integer.parseInt(tok.nextToken()));
      tok.nextToken(); // SERVERPORT
      client.connect(socket.getInetAddress(), Integer.parseInt(
                                                    tok.nextToken()));
      client.receiveData(dataStore);
      // Now we can do some work!
      guard.waitGuard();
      if (statsListener != null) statsListener.close();
    } catch (NoSuchElementException ex) {
      System.out.println("phat: Invalid server reply.");
      return;
    }
  }

  /**
   * Sends the SENDM command - for multiple connections :)
   */
  private void sendSendM(StringTokenizer tok) throws IOException {
    String cmd = tok.nextToken("");

    String filename;

    // Get the filename first

    int quotIdx = cmd.indexOf('\"');
    if (quotIdx >= 0) {
      // Quoted?

      int nextQuotIdx = cmd.indexOf('\"', quotIdx + 1);
      if (nextQuotIdx < 0) {
        System.out.println("phat: Missing final quote character.");
        return;
      }
      filename = cmd.substring(quotIdx + 1, nextQuotIdx);
    } else {
      // Not quoted

      StringTokenizer st = new StringTokenizer(cmd);
      filename = st.nextToken();
    }

    if (filename.length() == 0) {
      System.out.println("phat: Missing filename.");
      return;
    }

    // Make sure we can create the connection first
    FSTPController FSTPcon = new FSTPController(this);
    FSTPcon.getFile(filename);
```

```
//removed
}

/**
 * Sends the MULTISEND command - for multiple connections :)
 */
private void sendMulTiSend(StringTokenizer tok) throws IOException {
  int numCon = Integer.parseInt(tok.nextToken());
   // System.out.println("num connections is : "+numCon);
  String cmd = tok.nextToken("");

  String filename;
  long fileSize;
  // Get the filename first

  int quotIdx = cmd.indexOf('\"');
  if (quotIdx >= 0) {
    // Quoted?

    int nextQuotIdx = cmd.indexOf('\"', quotIdx + 1);
    if (nextQuotIdx < 0) {
      System.out.println("phat: Missing final quote character.");
      return;
    }
    filename = cmd.substring(quotIdx + 1, nextQuotIdx);
  } else {
    // Not quoted

    StringTokenizer st = new StringTokenizer(cmd);
    filename = st.nextToken();
  }

  if (filename.length() == 0) {
    System.out.println("phat: Missing filename.");
    return;
  }

  cmdBuf.setLength(0);
  cmdBuf.append(MULTISEND_CMD);
  cmdBuf.append(' ');
  cmdBuf.append(filename);
  sendMessage(cmdBuf.toString());

  waitForReply();
  String rply = rplyQ.pop();
  if (!rply.startsWith(COMMAND_OKAY_RPLY)) {
  //this.startBlockSize = blockSize;
     /*rply = rplyQ.pop();
     if (!rply.startsWith(FILE_INFO_RPLY)){
    System.out.println(
    return;
     }
     */

      System.out.println("didn't get the okay command - " + rply);
      return;
  }
  //  System.out.println("we did get the command - " + rply);

  waitForReply();
  rply = rplyQ.pop();
  if (!rply.startsWith(MFILE_INFO_RPLY)){
    System.out.println("not a mfile reply!- "+rply.toString());
```

```
      return;
    }
    // The file is being sent, so get the file information
    //System.out.println("got the mfile info reply!");
    tok = new StringTokenizer(rply);
    tok.nextToken();
    try {
      tok.nextToken(); // FILESIZE
       // client.setFileSize(Long.parseLong(tok.nextToken()));
       fileSize=Long.parseLong(tok.nextToken());
       // System.out.println("file size is - "+fileSize);

    } catch (NoSuchElementException ex) {
      System.out.println("phat: Invalid server reply.");
      return;
    }

    FSTPController FSTPcon=new FSTPController(this,fileSize,numCon,mServers);
    FSTPcon.getFile(filename);
  }

  /**
   * Sends the desired block delay in milliseconds.
   */
  private void sendBlockDelay(StringTokenizer tok) throws IOException {
    cmdBuf.setLength(0);
    cmdBuf.append(BLOCK_DELAY_CMD);
    cmdBuf.append(' ');
    blockDelay = Integer.parseInt(tok.nextToken());
    cmdBuf.append(blockDelay);
    sendMessage(cmdBuf.toString());

    waitForReply();
    rplyQ.pop();
  }

  /**
   * Sends the desired block delay in milliseconds.
   */
  public void sendBlockDelay(int newDelay) throws IOException {
      cmdBuf.setLength(0);
      cmdBuf.append(BLOCK_DELAY_CMD);
      cmdBuf.append(' ');
      cmdBuf.append(newDelay);
      sendMessage(cmdBuf.toString());

      waitForReply();
      rplyQ.pop();
  }

  /**
   * Sends the desired block size in packets.
   */
  private void sendBlockSize(StringTokenizer tok) throws IOException {
    cmdBuf.setLength(0);
    cmdBuf.append(BLOCK_SIZE_CMD);
    cmdBuf.append(' ');
    int blockSize = Integer.parseInt(tok.nextToken());
    cmdBuf.append(blockSize);
    sendMessage(cmdBuf.toString());

    waitForReply();
    if (rplyQ.pop().startsWith(COMMAND_OKAY_RPLY)) {
      this.startBlockSize = blockSize;
    }
  }

  /**
   * Sets the Current Working Directory
   */
  private void sendCWD(StringTokenizer tok) throws IOException {
    //System.out.println("We are in the change dir part ");
    cmdBuf.setLength(0);
    cmdBuf.append(CWD_CMD);
    cmdBuf.append(' ');
    String str = tok.nextToken();
    cmdBuf.append(str);
    sendMessage(cmdBuf.toString());
    //System.out.println("/t "+cmdBuf.toString());

    waitForReply();
    //System.out.println("reply == " +rplyQ.pop());
    if(rplyQ.pop().startsWith(COMMAND_OKAY_RPLY)) {
      //null;  //just cheching reply??
    }
  }

  private void sendCDUP(StringTokenizer tok) throws IOException {
    //System.out.println("We are in the change dir part ");
    cmdBuf.setLength(0);
    cmdBuf.append(CDUP_CMD);
    sendMessage(cmdBuf.toString());
    //System.out.println("/t "+cmdBuf.toString());

    waitForReply();
    //System.out.println("reply == " +rplyQ.pop());
    if(rplyQ.pop().startsWith(COMMAND_OKAY_RPLY)) {
      //  null;  //just cheching reply??
    }
  }

  /**
   * Sends the list command
   */
  private void sendList(StringTokenizer tok) throws IOException {
    cmdBuf.setLength(0);
    cmdBuf.append(LIST_CMD);
    try {
      cmdBuf.append(" " + tok.nextToken(""));
    } catch (NoSuchElementException ex) {
    }
    sendMessage(cmdBuf.toString());

    waitForReply();
    if (!rplyQ.pop().startsWith(SENDING_DIR_RPLY)) return;

    waitForReply();
    rplyQ.pop();
  }

  /**
   * Sends the desired packet size in bytes.
   */
  private void sendPacketSize(StringTokenizer tok) throws IOException {
    cmdBuf.setLength(0);
    cmdBuf.append(PACKET_SIZE_CMD);
    cmdBuf.append(' ');
    int packetSize = Integer.parseInt(tok.nextToken());
    cmdBuf.append(packetSize);
    sendMessage(cmdBuf.toString());

    waitForReply();
    if (rplyQ.pop().startsWith(COMMAND_OKAY_RPLY)) {
      this.packetSize = packetSize;
    }
  }

} //PhatClient
```

A.3 FSTPClient.java

```
package com.phatpackets.fstp;

import com.phatpackets.fstp.client.BitSetPacketTracker;

import BCK.ANN.BCKRecord;

import java.io.InterruptedIOException;
import java.io.IOException;
import java.net.*;
import java.util.Vector;
import java.util.TooManyListenersException;
import java.io.*;

/**
 * Title:
 * Description:
 * Copyright:Copyright (c) 2001
 * Company:
 * @author Shawn Silverman, modified by Greg Jaman, Chris Buzunis
 * @version 1.0
 */

/**
 * Modified May, 2002
 * C. H. Poskar
 *
 * Added support for VTP Emulation and Fuzzy Neural Controller:
 *  In DataHandler:
 *   - lastPacketLoss, used to calculate the change in packet loss,
 *   - minDelay, used to model Max. allocated bandwidth in terms of an
 *   adjusted block delay (set to 90% of initial delay),
 *   - trackDelay, a local variable to track the changes in block delay.
 *  Separated control implementation out of processData(), and into:
 *   - rateController1(), the original control for modifying delay/block size,
 *   - rateController2(), for interacting with the FNN controller,
 *   - rateController3(), can over-ride existing block delay, or call FNN.
 *
 * TODO:
 * Recording additional statistice, block loss, pack delay, etc...
 */

public class FSTPClient extends FSTPPeer implements CommandConstants {
  // Listeners States that this client could be in when stopped
  // - for firing events after this is stopped, but a new listener is added

  private static final int NOT_STOPPED_STATE       = 0;
  private static final int COMPLETE_STOPPED_STATE = 1;
  public  static final int STOPPED_STOPPED_STATE   = 2; //for now not private
  private static final int EXCEPTION_STOPPED_STATE = 3;

  //public for now - used to notify after stopped
  public                int stoppedState = NOT_STOPPED_STATE;

  private Vector        listeners;
  private StatsListener statsListener;
  private Exception     stoppedStateEx;

  public  AbstractHandler control;
  private DataHandler     dataHandler;
```

```
public  float alpha;
public  int   adjustSize;
private int   desiredAdjustSize;

// For dynamic block size
private boolean dynBlockSize;
public  boolean dynBlockDelay = true;
private boolean blockSizeCanGoLower;

private int startBlockSize;
private int upperBlockSize;
public  int currBlockSize;
private int lowerBlockSize;

public  long fileSize;    //G changed from private!
private long startTime;

private StringBuffer adjDelayBuf;
private StringBuffer adjBlockSizeBuf;
private StringBuffer rsndBuf;
private static final int ADJ_DELAY_LEN      = ADJUST_DELAY_CMD.length() + 1;
private static final int ADJ_BLOCK_SIZE_LEN =ADJUST_BLOCK_SIZE_CMD.length()+1;
private static final int RSND_LEN           = RESEND_PACKETS_CMD.length() + 1;

private static final int ADJUST_BLOCK_SIZE_STATE = 0;
private static final int ADJUST_DELAY_STATE      = 1;
// Two states: adjusting the block size, and adjusting the delay
private int state = ADJUST_BLOCK_SIZE_STATE;

private boolean singleServer = true;  //g

/**
 * A listener interface for listening to events from this server.
 *
 * @author Shawn Silverman
 */
public interface ClientListener {
  public void receiveComplete(int streamId, long time);

  public void receiveTimeout(int streamId, float percentComplete, long time);

  public void clientException(int streamId, Exception ex);

  /**
   * Indicates that the receive was stopped externally.
   */
  public void receiveStopped(int streamId, long time);

  /**
   * Indicates that the block size was adjusted.  This event only occurs
   * during the dynamic block sizing mode.
   */
  public void blockSizeAdjusted(int streamId, int newBlockSize);
}

/**
 * An interface for listening to packets received, for statistics
 * purposes.
 *
 * @author Shawn Silverman
 */
public interface StatsListener {
  public void setExpectedPacketCount(long count);
```

```
    public void packetReceived(long seqId);

    public void startResend();

    public void receiveTimeout();
  }

  /**
   * Creates a new client that listens for UDP packets on an assigned system
   * default port.
   * <ul>
   * <li>packetSize = 1472 bytes</li>
   * <li>streamId = 0</li>
   * <li>alpha = 0.95</li>
   * <li>adjustSize = 50 packets</li>
   * <li>dynamicBlockSize = true</li>
   * <li>startBlockSize = 4 packets</li>
   * </ul>
   */
  public FSTPClient(AbstractHandler control) throws SocketException {
    this(-1, control);
  }

  /**
   * Creates a new client that listens for UDP packets on the specified
   * port.  If the port is less than zero, then a system default port will
   * be assigned.
   * <p>
   * The default values for the properties are:
   * <ul>
   * <li>packetSize = 1472 bytes</li>
   * <li>streamId = 0</li>
   * <li>alpha = 0.95</li>
   * <li>adjustSize = 50 packets</li>
   * <li>dynamicBlockSize = true</li>
   * <li>startBlockSize = 4 packets</li>
   * </ul>
   * </p>
   */
  public FSTPClient(int port, AbstractHandler control) throws SocketException {
    super(port);

    //Default properties
    packetSize = 1472;
    streamId   = 0;
    alpha      = 0.95f;
    desiredAdjustSize = adjustSize = 50;
    dynBlockSize    = true;
    startBlockSize = 4;

    this.control = control;

    // Some storage space for send commands
    adjDelayBuf = new StringBuffer();
    adjDelayBuf.append(ADJUST_DELAY_CMD);
    adjDelayBuf.append(' ');

    adjBlockSizeBuf = new StringBuffer();
    adjBlockSizeBuf.append(ADJUST_BLOCK_SIZE_CMD);
    adjBlockSizeBuf.append(' ');

    rsndBuf = new StringBuffer();
    rsndBuf.append(RESEND_PACKETS_CMD);
    rsndBuf.append(' ');
  }

  /*
   * Listeners.
   */

  /*
   * FIX 11-Sep-2001 (Understood on 07-Sep-2001)
   * Found by Sheng Huang
   * Fix by Shawn Silverman
   *
   * This addresses the issue of a file completing before the listener has a
   * chance to get the appropriate stopped event.
   *
   * Synchronized all the methods in this section of code because it was
   * possible to add/remove listeners just, say, as a file was finishing up,
   * thus producing 'fire' events (for example).
   *
   * In addition, if a ClientListener is added after some important events,
   * then the events will be fired to this listener.  The reason for this is
   * that in the time it takes to start the file send and to initialize
   * client code, the server is already sending packets, and we want to miss
   * as few as possible.
   */
  public synchronized void addClientListener(ClientListener l) {
    if (l != null) {
      // Fire an event if this client is stopped
      // I'm beginning to think here that event objects would be a
      // cleaner approach

      if (NOT_STOPPED_STATE != stoppedState) {
        int id = getStreamId();
        long time = System.currentTimeMillis() - startTime;
        switch (stoppedState) {
          case COMPLETE_STOPPED_STATE:
            l.receiveComplete(id, time);
            break;
          case STOPPED_STOPPED_STATE:
            l.receiveStopped(id, time);
            break;
          case EXCEPTION_STOPPED_STATE:
            l.clientException(id, stoppedStateEx);
            break;
        }
      }

      if (listeners == null) listeners = new Vector();
      listeners.addElement(l);
    }
  }

  public synchronized void removeClientListener(ClientListener l) {
    if (l != null && listeners != null) {
      listeners.removeElement(l);
    }
  }

  /**
   * Adds a StatsListener.  This can only have one of these.  This should be
   * added before we start to receive.
   *
   * @throws TooManyListenersException if there is already a stats listener
   *   that has been registered.
   */
  public synchronized void addStatsListener(StatsListener l)
                                              throws TooManyListenersException {
    if (dataHandler != null) {
      throw new RuntimeException("Already receiving data.");
    }
    if (l != null) {
      if (statsListener != null) {
        throw new TooManyListenersException();
      }
      statsListener = l;
    }
  }
  public synchronized void removeStatsListener() {
    statsListener = null;
  }
  /**
   * Fires the ReceiveTimeout event.
   */
  private synchronized void fireReceiveTimeout(float percentComplete, long time) {
    if (listeners == null) return;
    for (int i = listeners.size(); --i >= 0; ) {
      ((ClientListener)listeners.elementAt(i)).receiveTimeout(getStreamId(),
                                                    percentComplete, time);
    }
  }
  /**
   * Fires the ReceiveComplete event.  The receive is stopped at this point.
   */
  private synchronized void fireReceiveComplete(long time) {
    stoppedState = COMPLETE_STOPPED_STATE;
    if (listeners == null) return;
    for (int i = listeners.size(); --i >= 0; ) {
      ((ClientListener)listeners.elementAt(i)).receiveComplete(getStreamId(), time);
    }
  }
  /**
   * Fires the ClientException event.  The receive is stopped at this point.
   */
  private synchronized void fireClientException(Exception ex) {
    stoppedState = EXCEPTION_STOPPED_STATE;
    stoppedStateEx = ex;
    if (listeners == null) return;
    for (int i = listeners.size(); --i >= 0; ) {
      ((ClientListener)listeners.elementAt(i)).clientException(getStreamId(), ex);
    }
  }

  /**
   * Fires the ReceiveStopped event.  The receive is stopped at this point.
   */
  private synchronized void fireReceiveStopped(long time) {
    stoppedState = STOPPED_STOPPED_STATE;
    if (listeners == null) return;
    for (int i = listeners.size(); --i >= 0; ) {
      ((ClientListener)listeners.elementAt(i)).receiveStopped(getStreamId(), time);
    }
  }

  /**
   * Fires the BlockSizeAdjusted event.
   */
  private synchronized void fireBlockSizeAdjusted(int newBlockSize) {
    if (listeners == null) return;
    for (int i = listeners.size(); --i >= 0; ) {
      ((ClientListener)listeners.elementAt(i)).blockSizeAdjusted(getStreamId(),
                                                               newBlockSize);
    }
  }

  private synchronized void fireStatsPacketReceived(long seqId) {
    if (statsListener != null) {
      statsListener.packetReceived(seqId);
    }
  }

  private synchronized void fireStatsStartResend() {
    if (statsListener != null) {
      statsListener.startResend();
    }
  }

  private synchronized void fireStatsReceiveTimeout() {
    if (statsListener != null) {
      statsListener.receiveTimeout();
    }
  }

  /*
   * Accessors.
   */

  /**
   * Sets the file size.
   *
   * @throws IllegalArgumentException if fileSize < 0.
   */
  public void setFileSize(long fileSize) {
    if (fileSize < 0) {
      throw new IllegalArgumentException("fileSize < 0");
    }
    this.fileSize = fileSize;
  }

  /**
   * Gets the file size.
   */
  public long getFileSize() {
    return fileSize;
  }

  /**
```

```
   * Gets the time elapsed.
   */
  public long getTimeElapsed() {
    return System.currentTimeMillis() - startTime;
  }

  /**
   * Sets the alpha value.
   *
   * @throws IllegalArgumentException if alpha is not in (0, 1].
   */
  public void setAlpha(float alpha) {
    if (alpha <= 0.0f || alpha > 1.0f) {
      throw new IllegalArgumentException("alpha not in (0, 1]");
    }
    this.alpha = alpha;
  }

  /**
   * Gets the alpha value.
   */
  public float getAlpha() {
    return alpha;
  }

  /**
   * Gets the number of packets for which to analyse and send the delay
   * adjustment.  Note that if dynamic block sizing is on, then this might
   * return a different value than when set.
   */
  public int getAdjustSize() {
    return this.adjustSize;
  }

  /**
   * Sets the number of packets for which to analyse and send the delay
   * adjustment.  This should be set to a multiple of the block size.
   * <p>
   * If the <code>dynamicBlockSize</code> property is set to
   * <code>true</code>, then the resulting adjust size will be the closest
   * multiple of the block size, or the block size itself, depending if the
   * block size is bigger than the desired adjust size.
   * </p>
   *
   * @throws IllegalArgumentException if adjustSize < 1.
   */
  public void setAdjustSize(int adjustSize) {
    if (adjustSize < 1) {
      throw new IllegalArgumentException("adjustSize < 1");
    }
    this.adjustSize = adjustSize;
    desiredAdjustSize = adjustSize;
  }

  /**
   * Special code here for calculating the adjust size for the dynamic
   * block size mode.
   *
   * At a block size of one, we want the adjust size.
   * At a block size of the adjust size or greater, we want the adjust size
   * to be equal to the block size.
   */
  private void calcDynAdjustSize(int blockSize) {
    if (blockSize >= desiredAdjustSize) {
      adjustSize = blockSize;
    } else if (blockSize == 1) {
      adjustSize = desiredAdjustSize;
    } else {
      adjustSize = (int)Math.ceil((float)adjustSize / (float)blockSize) * blockSize;
    }
  }

  /**
   * Sets that we shold adjust the block size when we start receiving.
   */
  public void setDynamicBlockSize(boolean flag) {
    dynBlockSize = flag;
  }

  public void setDynamicBlockDelay(boolean flag) {
    dynBlockDelay = flag;
  }

  /**
   * Checks if this should adjust the block size when we start receiving.
   */
  public boolean isDynamicBlockSize() {
    return dynBlockSize;
  }

  /**
   * Sets the starting block size for the dynamic block size mode.  This
   * value will be set to the next highest power of two if it is not already
   * a power of two.
   *
   * @throws IllegalArgumentException if the starting block size < 1.
   */
  public void setStartBlockSize(int startBlockSize) {
    if (startBlockSize < 1) {
      throw new IllegalArgumentException("startBlockSize < 1");
    }
    this.startBlockSize = (int)Math.pow(2.0, Math.ceil(Math.log(startBlockSize)/
                                                        Math.log(2.0)));
  }

  /**
   * Gets the percent complete.  This returns -1.0f if this is closed.
   * TODO - fix this to check for multiServer
   */
  public float getPercentComplete() {
    if (isClosed()) {  // TODO Fix this mess
      if (dataHandler != null && dataHandler.packetTracker != null) {
        return dataHandler.packetTracker.getPercentComplete();
      } else {
        return -1.0f;
      }
    }

    if (dataHandler == null || dataHandler.packetTracker == null) {
      return 0.0f;
    } else {
      return dataHandler.packetTracker.getPercentComplete();
    }
  }

  /*
   * Methods.
   */

  /**
   * Receives data into the specified data store.  This does nothing if we
   * are already receiving data, or if the client is closed.
   */
  public void receiveData(DataStorage dataStore) {
    if (isClosed()) return;

    if (dataHandler == null) {
      // Set to one of two states, or receiving modes

      if (dynBlockSize) {
        state = ADJUST_BLOCK_SIZE_STATE;
        // Note that startBlockSize will be a power of two
        upperBlockSize = lowerBlockSize = currBlockSize = startBlockSize;
        if (startBlockSize > 1) {
          blockSizeCanGoLower = true;
        }

        calcDynAdjustSize(currBlockSize);
      } else {
        state = ADJUST_DELAY_STATE;
      }

      dataHandler = new DataHandler(dataStore);
      startTime = System.currentTimeMillis();
      new Thread(dataHandler).start();
    }
  }

  //Added by g - Note that startBlockSize will be a power of two
  public void receiveDataM(Vector phatData, PacketTracker pt) {
    if (isClosed()) return;

    if (dataHandler == null) {
      // Set to one of two states, or receiving modes

      if (dynBlockSize) {
        state = ADJUST_BLOCK_SIZE_STATE;

        upperBlockSize = startBlockSize;
        lowerBlockSize = startBlockSize;
        currBlockSize  = startBlockSize;

        if (startBlockSize > 1) {
          blockSizeCanGoLower = true;
        }

        calcDynAdjustSize(currBlockSize);
      } else {
        state = ADJUST_DELAY_STATE;
      }

      dataHandler = new DataHandler(phatData,pt);     //changed! G
      startTime = System.currentTimeMillis();
      new Thread(dataHandler).start();            //change me!!!
    }
  }

  //Added by B - Note that startBlockSize will be a power of two
  public void receiveDataM(Vector phatData, PacketTracker pt, DataStorage
                                                    dataStore,int conID) {
    if (isClosed()) return;

    if (dataHandler == null) {
      // Set to one of two states, or receiving modes
      if (dynBlockSize) {
        state = ADJUST_BLOCK_SIZE_STATE;

        upperBlockSize = startBlockSize;
        lowerBlockSize = startBlockSize;
        currBlockSize  = startBlockSize;

        if (startBlockSize > 1) {
          blockSizeCanGoLower = true;
        }

        calcDynAdjustSize(currBlockSize);
      } else {
        state = ADJUST_DELAY_STATE;
      }

      dataHandler = new DataHandler(phatData,pt,dataStore, conID); //changed! G
      startTime = System.currentTimeMillis();
      new Thread(dataHandler).start();            //change me!!!
    } else {
      //System.out.println("Never closed connection from before!!!!!");
      dataHandler.packetTracker=pt;
      dataHandler.dataStore=dataStore;

      startTime = System.currentTimeMillis();
      new Thread(dataHandler).start(); //change me!!!
    }
  }

  public boolean isReceiving() {
    //return dataHandler != null;
    return dataHandler != null ? !dataHandler.stopped : false;
  }

  public void close() {
    //System.out.println("!!!!!the client is being closed!!!!");
    if (dataHandler != null) {
      dataHandler.stop();
      dataHandler = null;
    }
    if(singleServer) super.close();
  }

  public void doneRcv(){
    try{
      control.sendMessage(DONE_RECEIVING_CMD + " " + getStreamId());
    }
    catch(IOException e){}
  }
  /**
   * Sends the amount by which to adjust the delay.
   */
  private void sendAdjustDelay(int streamId, float factor) throws IOException {
    adjDelayBuf.setLength(ADJ_DELAY_LEN);
    adjDelayBuf.append(streamId);
    adjDelayBuf.append(' ');
    adjDelayBuf.append(factor);
    control.sendMessage(adjDelayBuf.toString());  //TODO Make more efficient
  }

  /**
   * Sends the amount to set the block size.
   */
  private void sendAdjustBlockSize(int streamId, int size) throws IOException {
    adjBlockSizeBuf.setLength(ADJ_BLOCK_SIZE_LEN);
    adjBlockSizeBuf.append(streamId);
    adjBlockSizeBuf.append(' ');
    adjBlockSizeBuf.append(size);
    control.sendMessage(adjBlockSizeBuf.toString()); //TODO Make more efficient
  }

  /**
   * Sends the RESENDPACKETS command.  The last element should contain the
   * earliest seq ID.
   *
   * @param seqIds a long array containing the packet sequence ID's to resend
   * @param off the offset into the array
   * @param len the length, starting from <code>off</code>
   */
  private void sendResendPackets(long[] seqIds,int off,int len) throws IOException {
    rsndBuf.setLength(RSND_LEN);
    rsndBuf.append(getStreamId());
    for (int i = off + len; --i >= off; ) {
      rsndBuf.append(' ');
      rsndBuf.append(seqIds[i]);
    }
    control.sendMessage(rsndBuf.toString());
  }

  /**
   * Handles incoming UDP data.
   *
   * @author Shawn Silverman
   */
  private class DataHandler implements Runnable {
    volatile boolean stopped;
    //public boolean stopped = false;
    private RetransmitRequester req;
    private Thread reqThread;
    PacketTracker packetTracker;
    private DataStorage dataStore;

    private long    currAdjustSeqEnd;  // Inclusive
    private int     currRcvSize;
    private boolean firstTime = true;  // Ignore some transients
    private boolean retransmitting;
    private int     conID;
  Vector phatData;  //this is the pointer to the data?? sure.. why no.
  //FNN
  private double lastPacketLoss = 0;
  private double trackDelay     = (double)control.getBlockDelay();
  private double minDelay       = (0.9 * trackDelay);

  /**
   * Creates a new data handler that listens to UDP packets.
   */
  DataHandler(DataStorage dataStore) {
    this.dataStore = dataStore;
  }


  //G added
  //hmm.. pass a pointer to the controler thread??
  DataHandler(Vector data) {
    singleServer=false;
    this.phatData=data;
    stopped=false;
  }

    //we need to remove this later - G
    //hmm.. pass a pointer to the controler thread??
    DataHandler(Vector data,PacketTracker packetTracker) {
    singleServer=false;
    this.packetTracker=packetTracker;
    this.phatData=data;
    stopped=false;
  }

    //hmm.. pass a pointer to the controler thread??
    DataHandler(Vector data,PacketTracker packetTracker,DataStorage dataStore,
                                                          int conID ) {
    singleServer=false;
    this.packetTracker=packetTracker;
    this.phatData=data;
    this.dataStore = dataStore;
    this.conID=conID;
    stopped=false;
  }

  /**
   * Stops this receiver.
   */
  void stop() {
    //System.out.println("calling stopped from stop");
    stopped = true;
    if (req != null) {
      req.stop();
    }
  }

  /**
   * Creates the packet tracker.
   */
  private void createPacketTracker() {
    currAdjustSeqEnd = adjustSize - 1;
    firstTime = true;
    // Calculate the # of packets
    int streamIdSize   = PhatPacket.calcStreamIdSize(getStreamId());
    int seqIdSize      = PhatPacket.calcSeqIdSize(fileSize, getPacketSize(),
                                                          streamIdSize);
    int packetDataSize = getPacketSize() - streamIdSize - seqIdSize;
    if (dataStore.isPacketDataSizeRequired()) {
      dataStore.setPacketDataSize(packetDataSize);
      dataStore.setTotalSize(fileSize);
    }
    long expPackets = (long)Math.ceil((double)fileSize/(double)packetDataSize);
    packetTracker = new BitSetPacketTracker();
    packetTracker.setExpectedPacketCount(expPackets);

    if (statsListener != null) {
      statsListener.setExpectedPacketCount(expPackets);
    }
  }

  public synchronized void continueRun(){
    notify();
  }

   public void run() {
    if (stopped) {
      System.out.println("the server is stopped??");
      return;
    }
    retransmitting = false;
    try {
      DatagramSocket dataPort = udpSocket;
      dataPort.setSoTimeout(2000);
      DatagramPacket udpPacket=new DatagramPacket(new byte[packetSize],packetSize);
      PhatPacket phatPacket = new PhatPacket(udpPacket);
      if(singleServer) {
        createPacketTracker();
      } else {
        currAdjustSeqEnd = adjustSize - 1; //done in the create packetTracker!
        currAdjustSeqEnd = packetTracker.getOffset() + adjustSize -1;
        firstTime = true;
      }
      while (!stopped && !packetTracker.isComplete()) {
        try {
          // FIX cuz the length sticks at the last length received
          udpPacket.setLength(packetSize);
          dataPort.receive(udpPacket);
          phatPacket.update();
          processData(phatPacket);
        } catch (InterruptedIOException ex) {
          if (!stopped) {
            fireReceiveTimeout(packetTracker.getPercentComplete(),
                  System.currentTimeMillis() - startTime);
            fireStatsReceiveTimeout();
            retransmitting = true;
            fireStatsStartResend();
            // Start more retransmit requests if not yet done
          if (!packetTracker.isComplete()) {
            if (reqThread == null || !reqThread.isAlive()) {
              req = new RetransmitRequester(packetTracker);
              reqThread = new Thread(req);
              reqThread.start();
            }
          } else {
            System.out.println("no need for resend!!");
          }
        }
      } //catch
    } //while

    // Send the DONE_RECEIVING regardless - don't close!!! please!!
    if (singleServer) {
      control.sendMessage(DONE_RECEIVING_CMD + " " + getStreamId());
    } else {
      control.sendMessage(DONE_RECEIVINGBLOCK_CMD + " "+ getStreamId());
    }

    long time = System.currentTimeMillis() - startTime;
    if (packetTracker.isComplete()) {
      fireReceiveComplete(time);
    } else if (stopped) {
      fireReceiveStopped(time);
    }
  } catch (IOException ex) {
    if (!stopped) {                // To avoid some SocketExceptions
      fireClientException(ex);
    } else {
      try{
        if(singleServer)
        control.sendMessage(DONE_RECEIVING_CMD + " " + getStreamId());
        else control.sendMessage(DONE_RECEIVINGBLOCK_CMD + " " + getStreamId());
      }
      catch (IOException ex2) {}
    }
  } catch (FSTPException ex) {
    fireClientException(ex);
  } finally {
    if(singleServer){
      System.out.println("calling stopped!");
      stopped = true;
    }
    try {
      if(singleServer) dataStore.close();
    }
    catch (IOException ex) {
    }
    if(singleServer) close();
  }
}

/**
 * Processes a packet of newly received data.
 */
private void processData(PhatPacket packet) throws IOException {
        long seqId = packet.getSeqId();
        fireStatsPacketReceived(seqId);

        // Track the packet
        if (packetTracker.isPacketReceived(seqId)) return;

        packetTracker.packetReceived(seqId);

        // Send it to the storage
        try {
          if (singleServer) {
            dataStore.pushData(packet);
          } else {
            dataStore.pushData(packet);
          }

        } catch (IOException ex) {
        } catch (FSTPException ex) {
          ex.printStackTrace();
          if (ex instanceof PacketSizeException) { } // TODO Do something
        }

        // We need to adjust differently, depending if we are retransmitting.
        if (!retransmitting) {
          // If we have started receiving the next block then adjust the
          // alpha or the block size
          if (seqId > currAdjustSeqEnd) {
            if (!firstTime) {
              rateController1();
            } else {
              firstTime = false;
            }

            // Readjust the seq monitoring, regardless of the state.
            currRcvSize = 1;
            currAdjustSeqEnd += adjustSize;
          } else {
            this.currRcvSize++;
          }

        } else {
          // TODO Adjust while retransmitting, too
        }
      }

      private void rateController1() throws IOException {
        // Do adjustments here, depending on the state
        if (state == ADJUST_BLOCK_SIZE_STATE) {
          if ((float)currRcvSize / (float)adjustSize < alpha) {
            // Block size is too big
            upperBlockSize = currBlockSize - 1;
            if (lowerBlockSize <= upperBlockSize) {
              currBlockSize = (lowerBlockSize + upperBlockSize)/2;
              if (currBlockSize == lowerBlockSize) {
                if (blockSizeCanGoLower && lowerBlockSize != 1) {
                  lowerBlockSize /= 2;
                  currBlockSize = lowerBlockSize;
                  if (lowerBlockSize == 1) {
                    blockSizeCanGoLower = false;
                  }
                } else {
                  state = ADJUST_DELAY_STATE;
                }
            }
          } else {
            if (upperBlockSize > 0) {
              currBlockSize = upperBlockSize;
            }
            state = ADJUST_DELAY_STATE;
          }
        } else {
          if (blockSizeCanGoLower) {
            blockSizeCanGoLower = false;
          }

          // All is well, so increase the block size
          if (currBlockSize != upperBlockSize) {
            lowerBlockSize = currBlockSize + 1;
            if (lowerBlockSize <= upperBlockSize) {
              currBlockSize = (lowerBlockSize + upperBlockSize)/2;
              if (currBlockSize == upperBlockSize) {
                state = ADJUST_DELAY_STATE;
              }
            } else {
              state = ADJUST_DELAY_STATE;
            }
          } else {
            lowerBlockSize = upperBlockSize;
            currBlockSize = (upperBlockSize *= 2);
          }
        }

        // currBlockSize now contains the latest test block size

        sendAdjustBlockSize(streamId, currBlockSize);
        fireBlockSizeAdjusted(currBlockSize);
        calcDynAdjustSize(currBlockSize);
      } else {
        // The ADJUST_DELAY state

        float adj;
        if (currRcvSize != 0) {
          adj = (float)adjustSize/(float)currRcvSize * alpha;
        } else {
          // Increase the delay if no packets are received
          adj = 1.0f / alpha;
        }

        // Don't send 1.0 since it's redundant
        // Don't send >= 2.0 since that's probably pre-emption or something
        // In this case, send a factor that we can undo by receiving 100% in
        // the next round

        if (adj >= 2.0f) adj = 1.0f / alpha;

        if (adj != 1.0f) {
          if(dynBlockDelay) {
            sendAdjustDelay(streamId, adj);
          }
        }
      }
    } //rateController1()

    private void rateController2() throws IOException {
      double adj;
      if (currRcvSize != 0) {
        double pl = 1 - ((float)currRcvSize/(float)adjustSize);
        double deltaPL = pl - lastPacketLoss;

        adj = control.fuzzyControl(pl, deltaPL);

        lastPacketLoss = pl;
      } else {
        // Increase the delay if no packets are received
        adj = (double)(1.0 / alpha);
      }

      //Don't send if adj=1, no change
      //Don't send if achieving Max. allocated bandwidth, specified
      //in terms of a minimum delay.
      if ( (adj != 1.0f) && trackDelay > minDelay) {
        sendAdjustDelay(streamId, (float)adj);
        trackDelay *= adj;
      }

    } //rateControl2()

  private void rateController3(int newDelay) throws IOException {

    if ( (newDelay > 0) && (trackDelay != newDelay) ) {
      control.sendBlockDelay(newDelay);
      minDelay = trackDelay = (double)newDelay;
    } else {
      rateController2();
    }

  } //rateControl3()

  }//DataHandler


  /**
   * Handles retransmission requests, after the whole file has been sent.
   * This sends out a request for all blocks of packets and then terminates.
   * @author Shawn Silverman
   */
  private class RetransmitRequester implements Runnable {
    private StringBuffer cmdBuf = new StringBuffer();
    //private volatile boolean stopped;
    private boolean stopped;
    private PacketTracker packetTracker;

    RetransmitRequester(PacketTracker packetTracker) {
      super();
      this.packetTracker = packetTracker;
    }

    void stop() {
      stopped = true;
    }

    public void run() {
      // Go through the packet tracker and request retransmissions in
      // blocks of the adjust size.

      // The array has the earliest sequence ID as the last element
      //System.out.println("\n\n!!!IN RETRANSMITT!!! for" + getStreamId());
      long[] seqIds = new long[adjustSize];
```

```
        int index = adjustSize;
        synchronized (packetTracker) {
          try {
            while (!stopped && !packetTracker.isComplete()) {
              packetTracker.startIterateMissing();
              long seqId;
              while (-1L != (seqId = packetTracker.nextMissing())) {
                seqIds[--index] = seqId;

                if (index == 0) {
                  sendResendPackets(seqIds, 0, adjustSize);
                  index = adjustSize;
                }
              }

              // Send any extra
              if (index < adjustSize) {
                sendResendPackets(seqIds, index, adjustSize - index);
                index = adjustSize;
              }

              try {
                Thread.sleep(500);
              } catch (InterruptedException ex) {
              }

            }
          } catch (IOException ex) {
            stopped = true;
            close();
            fireClientException(ex);
            // TODO This might possibly result in a double exception
          }
        }
      }
    } //RetransmitRequester

  }//FSTPClient
```

Appendix B: Extension Classes to the BCK for Implementing Fuzzy Neural Networks:

The following sections of this appendix contain the source code for the classes added to the BCK to support Fuzzy Neural Networks. In addition the last class is provided as it required major revisions, for implementing the fuzzy back-propagation learning rule. Changes made to the remaining classes were merely to provide support (via placeholder methods) and do not contain added functionality. The code is presented in the following order

Appendix B.1 - BCKFuzzyNeuron.java

- an extension to the BCKNeuron class to provide support for the two general classes of fuzzy neurons: OR and AND.

Appendix B.2 - BCKFuzzyORNeuron.java

- an extension of BCKFuzzyNeuron to implement a general OR type fuzzy neuron, including methods required to support learning.

Appendix B.3 - BCKFuzzyANDNeuron.java

- an extension of BCKFuzzyNeuron to implement a general AND type fuzzy neuron, including methods required to support learning.

Appendix B.4 - BCKFNN.java

- an extension of the BCKNeuralNetwork class to create and manage a fuzzy neural network.

Appendix B.5 - BCKFNNBprop.java

- an extension of the BCKFNN class to implement network training and calculation of the numerical weight changes.

Appendix B.6 - BCKSynapse.java

- A class to model the connections between neurons, and also manages the learning rule (weight update) methods.

The original BCK code is available from:

`http://www.compapp.dcu.ie/~tdoris/BCK/`

B.1 BCKFuzzyNeuron.java

```
package BCK.ANN;

import BCK.ANN.BCKFNN;
import java.util.*;
import java.io.*;

/** BCKFuzzyNeuron
 * The Fuzzy Neuron class extends the neuron class, adding the functionality
 * needed to form the basis for a general Referential Fuzzy Neuron. The basic
 * functions are provided through triangular norm logic functions (t and s
 * norms) and a fuzzy implication function.
 * This is the base class for the two classes of Fuzzy Neurons, Disjunctive
 * (OR type) and Conjunctive (AND type).
 *
 * @author C. H. Poskar, May 2002
 * @version 1.0
 *
 * TODO
 * Change Synapse to model parameters rather than weights, and add references
 * to synapse - maybe, extra parameter per synapse.
 */
public class BCKFuzzyNeuron extends BCKNeuron implements Serializable{

/* Constants */
  public final static int CONJUNCTIVE = 0;
  public final static int DISJUNCTIVE = 1;

  public final static int NO_REF      = 0;
  public final static int MATCHING    = 1;
  public final static int DIFFERENCE  = 2;
  public final static int INCLUSION   = 3;
  public final static int DOMINANCE   = 4;

  public final static int NUM_WEIGHTS = BCKFNN.NUM_WEIGHTS;

  public BCKFuzzyNeuron(){
    super();
    type ="Fuzzy";
    refType = NO_REF;
  }

/* Lukasiewicz OR operation:
 *       x s y = min[1, x+y]
 */
  public double s_norm (double x, double y) {
    double temp = x + y;

    if (temp > 1) {
      return (double)1;
    }

    return temp;
  }

/* Lukasiewicz AND operation:
 *       x t y = max[0, x+y-1]
 */
```

```
  public double t_norm (double x, double y) {
    double temp = x + y - 1;

    if (temp < 0) {
      return (double)0;
    }

    return temp;
  }

/* Godel Implication:
 *       x->y = y, if  x > y
 *              1, otherwise
 */
  public double impl(double x, double y) {
    if (x > y) {
      return y;
    }

    return (double)1;
  }

/* Generic Reference Function:
 * - accepts two reference arguments and a reference type.
 * - executes the correct reference operation and returns the result [0..1].
 * - if noreference operation is selected, return the first argument.
 * - if reference type is invalid, returns -1.
 */
  protected double ref(double x, double y, int type) {

    if (type == NO_REF) {
      return x;
    } else if (type == MATCHING) {
      return match(x,y);
    } else if (type == DIFFERENCE) {
      return diff(x,y);
    } else if (type == INCLUSION) {
      return inc(x,y);
    } else if (type == DOMINANCE) {
      return dom(x,y);
    }

    return (double)-1;
  }

/* Matching Function:
 *       Match(x, y) = 1/2[(x->y) ^ (y->x) + (~x->~y) ^ (~y->~x)]
 */
  protected double match(double x, double y) {
    double temp1 = t_norm(impl(x,y),impl(y,x));
    double temp2 = t_norm(impl((1-x),(1-y)),impl((1-y),(1-x)));

    return ( (0.5) * s_norm(temp1,temp2) );
  }

/* Difference Function:
 *       Diff(x, y) = 1 - Match(x, y)
 */
  protected double diff(double x, double y) {
    return ( 1 - match(x,y) );
  }
```

```
/* Inclusion Function:
 *       Inc(x, y) = x -> y
 */
  protected double inc(double x, double y) {
    return ( impl(x,y) );
  }

/* Dominance Function:
 *       Dom(x, y) = y -> x
 */
  protected double dom(double x, double y) {
    return ( impl(y,x) );
  }

  /* randomParameters()
   * - return an array (of size arraSize) of random parameters.
   */
  private double[] randomParameters(int arraySize) {
    double[] params = new double[arraySize];

    for(int i=0; i<arraySize; i++){
      params[i] = 3*(0.5-Math.random());
    }

    return params;
  }

  /* setReferences()
   * - set the reference array from a given reference array.
   */
  public void setReferences(double[] refs){
    references = new double[refs.length];
    System.arraycopy(refs, 0, references, 0, refs.length);
  }

  /* setReferences()
   * - set the reference array to a new random parameters.
   */
  public void setReferences(int size){
    references = randomParameters(size);
  }

  /* setRefType()
   * - set the reference operation.
   */
  public void setRefType(int ref){
    refType = ref;
  }

  /* getReferences()
   * - return the reference array.
   */
  public double[] getReferences(){
    return references;
  }

  /* getRefType()
   * - return the Reference Type.
   */
  public int getRefType(){
      return refType;
    }

    protected double[] references = null;  //Array of reference values
    protected int      refType;            //Type of reference operation
    protected int      aggType;            //Type of aggregative neuron

} // BCKFuzzyNeuron
```

B.2 BCKFuzzyORNeuron.java

```
package BCK.ANN;

import java.util.*;
import java.io.*;

/** BCKFuzzyORNeuron - A Generalized Disjunctive Referential Fuzzy Neuron
 * - performs an OR type aggregation.
 * - has two optional parameters, a reference type, and an array of
 * reference values.
 * - if no reference type is specified, or a reference type without a
 * corresponding reference array, it is assumed the reference operation
 * is No Reference - i.e. an Aggregative Neuron.
 *
 * @author C. H. Poskar, May 2002
 * @version 1.0
 *
 * Valid Reference types are:
 *   0 - No reference
 *   1 - Matching
 *   2 - Difference
 *   3 - Inclusion
 *   4 - Dominance
 */
public class BCKFuzzyORNeuron extends BCKFuzzyNeuron implements Serializable{

  public BCKFuzzyORNeuron(){
    super();
    aggType = DISJUNCTIVE;
  }

  public BCKFuzzyORNeuron(int rType){
    super();
    aggType = DISJUNCTIVE;
  }

  public BCKFuzzyORNeuron(int rType, double[] refs){
    super();
    aggType = DISJUNCTIVE;
    if (refType != NO_REF) {
      setRefType(rType);
      setReferences(refs);
    }
  }

  public BCKFuzzyORNeuron(int rType, int size){
    super();
    aggType = DISJUNCTIVE;
    if (refType != NO_REF) {
      setRefType(rType);
      setReferences(size);
    }
  }

  // s_norm(0,X) = X
  protected double calcActivation() throws Exception{
    this.activation=0;
    BCKSynapse syn;
    double temp;

    for(int i=0; i<inputs.size(); i++) {
      syn = (BCKSynapse)inputs.elementAt(i);

      double   input   = syn.output.getState(syn.delay);
      double   inputC  = 1 - input;
      double[] weights = syn.getWeights();
//check output of functions for -1 - invalid refType??
      if (refType != NO_REF) {
        input  = ref(input,  references[i], refType);
        inputC = ref(inputC, references[i], refType);
      }
      temp = s_norm(t_norm(input, weights[0]), t_norm(inputC, weights[1]));
      this.activation = s_norm(this.activation, temp);
    }
    return this.activation;
  }

  //using the current activation, calculate the neuron's new output
  protected double transfer(){
    timeAdvance();
    StateHistory[tick] = this.activation;

    return StateHistory[tick];
  }

  //return the derivative of the transfer function at the current activation:
  public double[] netprimes(int i){
    double A  = 0;    // s_norm(0,X) = X
    double a  = 0;
    double   inputI   = 0;
    double[] weightsI = null;
    for(int j=0; j<inputs.size(); j++) {
      BCKSynapse syn = (BCKSynapse)inputs.elementAt(j);
      double   input   = syn.output.getState(syn.delay);
      double[] weights = syn.getWeights();
      if (i == j) {
        inputI = input;
        weightsI = new double[weights.length];
        System.arraycopy(weights, 0, weightsI, 0, weights.length);
      } else {
        double temp = s_norm(t_norm(input, weights[0]),
                    t_norm(1-input, weights[1]));
        A = s_norm(A, temp);
      }
    }
    double[] netprimes  = new double[weightsI.length];
         da  = new double[weightsI.length];
    for (int j=0; j<netprimes.length; j++) {
      if (j == 0) {
        A  = s_norm(A, t_norm(1-inputI, weightsI[1]));
        a  = t_norm(inputI,  weightsI[0]);
```

```
          da[j] = impl(weightsI[0], 1-inputI);
        } else {
          A  = s_norm(A, t_norm(inputI, weightsI[0]));
          a  = t_norm(1-inputI,  weightsI[1]);
          da[j] = impl(weightsI[1], inputI);
        }

        netprimes[j]  = impl(1-a,A);
      }

      return netprimes;
    }

    public double[] getLastDerivative(){
      return da;
    }

    double[] da = null;   // Fuzzified derivative of 'a'

  } //BCKFuzzyORNeuron
```

B.3 BCKFuzzyANDNeuron.java

```java
package BCK.ANN;

import java.util.*;
import java.io.*;

/** BCKFuzzyANDNeuron - A Generalized Conjunctive Referential Fuzzy Neuron
 * - performs an AND type aggregation.
 * - has two optional parameters, a reference type, and an array of
 * reference values.
 * - if no reference type is specified, or a reference type without a
 * corresponding reference array, it is assumed the reference operation
 * is No Reference - i.e. an Aggregative Neuron.
 *
 * @author C. H. Poskar, May 2002
 * @version 1.0
 *
 * Valid Reference types are:
 *   0 - No reference
 *   1 - Matching
 *   2 - Difference
 *   3 - Inclusion
 *   4 - Dominance
 */
public class BCKFuzzyANDNeuron extends BCKFuzzyNeuron implements Serializable{

  public BCKFuzzyANDNeuron(){
    super();
    aggType = CONJUNCTIVE;
  }

  public BCKFuzzyANDNeuron(int rType){
    super();
    aggType = CONJUNCTIVE;
  }

  public BCKFuzzyANDNeuron(int rType, double[] refs){
    super();
    aggType = CONJUNCTIVE;
    if (refType != NO_REF) {
      setRefType(rType);
      setReferences(refs);
    }
  }

  public BCKFuzzyANDNeuron(int rType, int size){
    super();
    aggType = CONJUNCTIVE;
    if (refType != NO_REF) {
      setRefType(rType);
      setReferences(size);
    }
  }

  // t_norm(1,X) = X
  protected double calcActivation() throws Exception{
    this.activation=1;
    BCKSynapse syn;
    double temp;

    for(int i=0; i<inputs.size(); i++) {
      syn = (BCKSynapse)inputs.elementAt(i);

      double   input   = syn.output.getState(syn.delay);
      double   inputC  = 1 - input;
      double[] weights = syn.getWeights();
//check output of functions for -1 - invalid refType??
      if (refType != NO_REF) {
        input  = ref(input,  references[i], refType);
        inputC = ref(inputC, references[i], refType);
      }

      temp = t_norm(s_norm(input, weights[0]), s_norm(inputC, weights[1]));

      this.activation = t_norm(this.activation, temp);
    }

    return this.activation;
  }

  //using the current activation, calculate the neuron's new output
  protected double transfer(){
    timeAdvance();

    StateHistory[tick] = this.activation;

    return StateHistory[tick];
  }

  //return the derivative of the transfer function at the current activation:
  public double[] netprimes(int i){
    double B  = 1;   // t_norm(1,X) = X
    double b  = 0;

    double   inputI   = 0;
    double[] weightsI = null;
    for(int j=0; j<inputs.size(); j++) {
      BCKSynapse syn = (BCKSynapse)inputs.elementAt(j);

      double   input   = syn.output.getState(syn.delay);
      double[] weights = syn.getWeights();

      if (i == j) {
        inputI = input;
        weightsI = new double[weights.length];
        System.arraycopy(weights, 0, weightsI, 0, weights.length);
      } else {
        double temp = t_norm(s_norm(input, weights[0]),
                    s_norm(1-input, weights[1]));
        B = t_norm(B, temp);
      }
    }

    double[] netprimes = new double[weightsI.length];
    db = new double[weightsI.length];
    for (int j=0; j<netprimes.length; j++) {
      if (j == 0) {
        B  = t_norm(B, s_norm(1-inputI, weightsI[1]));
        b  = s_norm(inputI,  weightsI[0]);
        db[j] = impl(1-inputI, weightsI[0]);
        } else {
          B  = t_norm(B, s_norm(inputI, weightsI[0]));
          b  = s_norm(1-inputI,  weightsI[1]);
          db[j] = impl(inputI, weightsI[1]);
        }
        netprimes[j]  = impl(B,1-b);
      }
      return netprimes;
    }
    public double[] getLastDerivative(){
      return db;
    }
    double[] db = null;   // Fuzzified derivative of 'b'
} //BCKFuzzyANDNeuron
```

B.4 BCKFNN.java

```
package BCK.ANN;

import java.util.*;
import java.io.*;

/** BCKFNN
 *
 * This class extends BCKNeuralNetwork to implement a Fuzzy Neural Network
 *
 * It uses multi-weight synapses to provide seperate weights for the input
 * and the complement of the input.
 *
 * @author C. H. Poskar, May 2002
 * @version 1.0
 *
 * TODO:
 * lastWeightChange?? is it necessary?
 */
public class BCKFNN extends BCKNeuralNetwork implements Serializable{
  public final static int NUM_WEIGHTS = 2;            //# of weights/synapse

  protected           int          numNeurons;        // Neurons in network
  protected transient double       globalError;       // The squared error
  protected transient double[][][] lastWeightChange;  // Remembers last change
  protected           int[]        numNeuronsInLayer; // Neurons in each layer
  protected           BCKNeuron    biasAND;           // A bias (dummy) neuron
  protected           BCKNeuron    biasOR;            // ..

  /* Default Constructor
   * - privatised to prevent use.
   */
  protected BCKFNN(){
  }

  /* Parameterised Constructor - fully connected
   * - numNeuronsPerLayer: contains an entry describing the number of
   * neurons in each layer, the network is constructed fully connected:
   */
  public BCKFNN(int[] numNeuronsPerLayer, int[] agg, int[] ref, double[][]
                                                               references){
    super(0, "fuzzy");
    numNeurons=0;
    BCKNeuron[] inputArray;

    numOutputs = numNeuronsPerLayer[numNeuronsPerLayer.length-1];
    numInputs  = numNeuronsPerLayer[0];

    numNeuronsInLayer = new int[numNeuronsPerLayer.length];
    for(int i=0;i < numNeuronsPerLayer.length; i++){
      numNeuronsInLayer[i]=numNeuronsPerLayer[i];
      numNeurons+=numNeuronsPerLayer[i];
    }

    lastWeightChange = new double[numNeurons][numNeurons][NUM_WEIGHTS];
    for(int i=0; i<lastWeightChange.length; i++){
      for(int j=0; j<lastWeightChange[i].length; j++){
        for(int k=0; k<NUM_WEIGHTS; k++){
```

```
        lastWeightChange[i][j][k]=0;
      }
    }
  }

  double[] weights   = new double[NUM_WEIGHTS];
  int  preStart  = 0;
  int  postStart = numInputs;

  //create the required neurons
  for(int i = 0; i < numNeurons; i++){
    if (agg[i] == BCKFuzzyNeuron.DISJUNCTIVE) {
      addNeuron(new BCKFuzzyORNeuron(ref[i], references[i]));
    } else if (agg[i] == BCKFuzzyNeuron.CONJUNCTIVE) {
      addNeuron(new BCKFuzzyANDNeuron(ref[i], references[i]));
    } else {
      System.out.println("Error in the BCKFNN constructor "
                + "- invalid neuron type");
    }
  }

  //create a dummy neuron whose output is always 1 in order to model
  //bias - note that this requires 2 bias neurons, one for AND (0)
  //and one for OR (1):
  biasAND = new BCKNeuron();
  biasAND.id=-1;
  biasAND.setOutput(0.0);
  biasOR = new BCKNeuron();
  biasOR.id=-2;
  biasOR.setOutput(0.0);

  double[] biasWeights = new double[NUM_WEIGHTS];
  for (int i=0; i<NUM_WEIGHTS; i++) {
    biasWeights[i] = 0.5;
  }

  //connect each neuron in a layer to all neurons in previous layer:
  //for each layer
  for(int layer=1; layer < numNeuronsInLayer.length; layer++) {

    //for each postsynaptic neuron...
    for(int post=0; post < numNeuronsInLayer[layer]; post++) {
      int postIndex = postStart+post;//Create index for post

      //connect to the bias neuron
      try{
        if (layer == 1) {
          connectExternal(biasAND, postIndex, biasWeights, 0);
        } else {
          connectExternal(biasOR, postIndex, biasWeights, 0);
        }
      }
      catch(Exception e){
        System.out.println("Error connecting to bias neuron"
                  + e.toString());
      }

      //for each presynaptic neuron...
      for(int pre=0; pre < numNeuronsInLayer[layer-1]; pre++){
        int preIndex = preStart+pre;
        //with random weight and no (zero) signal delay:

        for (int i=0; i<NUM_WEIGHTS; i++) {
          weights[i] = (Math.random() + 0.5) / 2;
        }
        try {
          connectInternal(preIndex, postIndex, weights, 0);
        }
        catch (Exception e){
          System.out.println("Error in BCKFNN constructor while "
                   + "connecting neurons: " + e.toString());
        }
      }
    }
    preStart  +=numNeuronsInLayer[layer-1];
    postStart +=numNeuronsInLayer[layer];
  }
}

/* Parameterised Constructor - AND/OR network specified
 * - numNeuronsPerLayer: contains an entry describing the number of
 * neurons in each layer.
 * - netStructure: provides the structure for the connection to layers
 * 2 (AND plane) and layer 3 (OR plane).
 */
public BCKFNN(int[] numNeuronsPerLayer, int[][][] netStructure, int[][] ref,
                                                   double[][][] references){
  super(0, "fuzzy");
  numNeurons=0;
  BCKNeuron[] inputArray;

  numOutputs = numNeuronsPerLayer[numNeuronsPerLayer.length-1];
  numInputs  = numNeuronsPerLayer[0];

  numNeuronsInLayer = new int[numNeuronsPerLayer.length];
  System.arraycopy(numNeuronsPerLayer, 0, numNeuronsInLayer, 0,
                                                   numNeuronsInLayer.length);

  double[] weights   = new double[NUM_WEIGHTS];
  int  preStart  = 0;
  int  postStart = numInputs;

  //create the required neurons
  for(int layer=0; layer<numNeuronsInLayer.length; layer++) {
    numNeurons+=numNeuronsInLayer[layer];
    for(int post=0; post<numNeuronsInLayer[layer]; post++) {
      if (layer == 0) { //input layer
        addNeuron(new BCKNeuron());
      } else if (layer == 1) {  //hidden layer (AND plane)
        addNeuron(new BCKFuzzyANDNeuron(ref[layer][post],
                                                   references[layer][post]));
      } else if (layer == 2) {  //output layer (OR plane)
        addNeuron(new BCKFuzzyORNeuron(ref[layer][post],
                                                   references[layer][post]));
      } else {
        System.out.println("Error in the BCKFNN constructor "
                  + "- invalid neuron type");
      }
    }
  }

  lastWeightChange = new double[numNeurons][numNeurons][NUM_WEIGHTS];
  for(int i=0; i<lastWeightChange.length; i++){
    for(int j=0; j<lastWeightChange[i].length; j++){
      for(int k=0; k<NUM_WEIGHTS; k++){
        lastWeightChange[i][j][k]=0;
      }
    }
  }

  //create a dummy neuron whose output is always 1 in order to model
  //bias - note that this requires 2 bias neurons, one for AND (0)
  //and one for OR (1):
  biasAND = new BCKNeuron();
  biasAND.id=-1;
  biasAND.setOutput(0.0);
  biasOR = new BCKNeuron();
  biasOR.id=-2;
  biasOR.setOutput(0.0);

  double[] biasWeights = new double[NUM_WEIGHTS];
  for (int i=0; i<NUM_WEIGHTS; i++) {
    biasWeights[i] = 0.5;
  }

  //connect each neuron in a layer to all neurons in previous layer:
  //for each layer
  for(int layer=1; layer < numNeuronsInLayer.length; layer++) {

    //for each postsynaptic neuron...
    for(int post=0; post < numNeuronsInLayer[layer]; post++) {
      int postIndex = postStart+post;//Create index for post

      //connect to the bias neuron
      try{
        if (layer == 1) {
          connectExternal(biasAND, postIndex, biasWeights, 0);
        } else {
          connectExternal(biasOR, postIndex, biasWeights, 0);
        }
      }
      catch(Exception e){
        System.out.println("Error connecting to bias neuron"
                  + e.toString());
      }

      //for each presynaptic neuron...
      for(int pre=0; pre < numNeuronsInLayer[layer-1]; pre++){
        //if no connection from pre to post, skip to next pre
        if (netStructure[layer][post][pre] == 0) {
          continue;
        }

        int preIndex = preStart+pre;
        //with random weight and no (zero) signal delay:

        for (int i=0; i<NUM_WEIGHTS; i++) {
          weights[i] = (Math.random() + 0.5) / 2;
        }
        try {
          connectInternal(preIndex, postIndex, weights, 0);
        }
        catch (Exception e){
          System.out.println("Error in BCKFNN constructor while "
                   + "connecting neurons: " + e.toString());
        }
      }
```

```
        }
        preStart  +=numNeuronsInLayer[layer-1];
        postStart +=numNeuronsInLayer[layer];
      }
  }

  //*************** Processor Methods ***********************

  /* forwardPass()
   * - execute a forward pass through the network.
   */
  public void forwardPass() {
    try{
      for(int i=numInputs; i < getNumberOfNeurons(); i++){
        calcState(i);
      }
    }
    catch (Exception e) {
      System.out.println(e.toString());
    }
  }

  /* setInput()
   * - TODO - check inputVector length.
   */
  public void setInput(double[] inputVector) {
    for(int i=0;i< inputVector.length; i++) {
      setOutput(i, inputVector[i]);
    }
  }

  //************* Accessor Methods ************************

  /* getBias()
   * - returns null, since there is no single bias neuron.
   */
  public BCKNeuron getBias(){
    return null;
  }

  /* getOutputs()
   * - returns the current output of output neurons.
   */
  public double[] getOutputs() {
    double[] outs = new double[numOutputs];
    int j=0;
    for(int i=numNeurons - numOutputs; j < numOutputs; j++) {
      outs[j] = getState(i+j);
    }
    return outs;
  }

  /* getGlobalError()
   * - returns the current global error.
   */
  public double getGlobalError(){
    return globalError;
  }

  /* randomiseWeights()
```

```
     * - reset all weights in the network to random values.
     */
    public void randomiseWeights() {
      BCKSynapse[] synarray;
      for(int i=0; i<neurons.size(); i++){
        BCKNeuron node = (BCKNeuron)neurons.elementAt(i);
        synarray = node.getSynapseArray();
        for(int s=0; s<synarray.length; s++){
           double[] weights = new double[NUM_WEIGHTS];
           for (int j=0; j<NUM_WEIGHTS; j++) {
            weights[i] = (Math.random() + 0.5) / 2;
          }
          synarray[s].setWeights(weights);
        }
      }

      //must reset lastWeightChange to 0
      for(int i=0; i<lastWeightChange.length; i++){
        for(int j=0; j<lastWeightChange[i].length; j++){
          for(int k=0; k<NUM_WEIGHTS; k++){
            lastWeightChange[i][j][k]=0;
          }
        }
      }
    }

    /* getNumberOfLayers()
     * - returns the number of layers in the network.
     */
    public int getNumberOfLayers(){
      return numNeuronsInLayer.length;
    }

    /* getNumberOfNeuronsInLayer()
     * - returns the number of neurons in the specified layer.
     */
    public int getNumberOfNeuronsInLayer(int i){
      return numNeuronsInLayer[i];
    }

}
```

B.5 BCKFNNBprop.java

```
package BCK.ANN;

import java.util.*;
import java.io.*;

/** BCKFNNBprop
 *
 * This class extends BCKFNN to implement Back Propagation Learning for
 * a Fuzzy Neural Network.
 *
 * @author C. H. Poskar, May 2002
 * @version 1.0
 */
public class BCKFNNBprop extends BCKFNN  implements Serializable{
  private double LearningRate;
  private double Momentum;

  //must not allow construction of uninitialised network
  protected BCKFNNBprop(){
    LearningRate = .1;
    Momentum = .7;
  }

  public BCKFNNBprop(int[] numNeuronsPerLayer, int[] agg, int[] ref, double[][]
                                                                 references){
    super(numNeuronsPerLayer, agg, ref, references);
    LearningRate = .1;
    Momentum = .7;
  }

  public BCKFNNBprop(int[] numNeuronsPerLayer, int[][][] netStruct, int[][] ref,
                                                  double[][][] references){
    super(numNeuronsPerLayer, netStruct, ref, references);
    LearningRate = .1;
    Momentum = .7;
  }

  public void setLearningRate(double lr){
    this.LearningRate=lr;
  }

  public void setMomentum(double mom){
    this.Momentum=mom;
  }

  //the standard backprop requires that we add a momentum term, which
  //in turn requires that we 'remember' the weights as they were in the
  //last training iteration
  public void train(int numEpochs) throws Exception{
    int       maxDW         = 1;
    double    o             = 0.0;
    double[] rec            = new double[numOutputs+numInputs]; //a record
    double[] inputVector    = new double[numInputs];  //current input vector
    double[] error          = new double[numNeurons]; //error at each neuron
    double[] desiredOutput  = new double[numOutputs]; //desired output vector
    double[][][] gradient   = new double[numNeurons][numNeurons][NUM_WEIGHTS];
    double[][][] sumTerm2  = new double[numNeurons][numNeurons][NUM_WEIGHTS];

    BCKNeuron[] inputArray;

    if(trainData==null){
      throw new Exception("You must specify a training file before "
                + "attempting to train the network.");
    }

    // Calculate the normalization parameter.
    for (int layer=2; layer<numNeuronsInLayer.length; layer++) {
      maxDW *= numNeuronsInLayer[layer];
    }

    int NumberOfRecords = trainData.getNumberOfRecords();
    for(int epochnumber=0; epochnumber<numEpochs; epochnumber++){

      globalError=0.0;
      for(int recordNumber=0; recordNumber<NumberOfRecords; recordNumber++){
        //get the next record
        rec = trainData.getNextRecord();

        //extract inputs
        for(int i=0;i<numInputs;i++) {
          inputVector[i] = rec[i];
        }

        //extract desired outputs
        for(int i=0; i<numOutputs; i++) {
          desiredOutput[i] = rec[i+numInputs];
        }

        //feed input vector into network
        setInput(inputVector);

        //evaluate network's reaction to input
        forwardPass();

        /**
         * Calculate the error measure and gradients on each neuron and
         * change the weights from each layer as appropriate:
         */
        for(int post=numNeurons-1; post>numInputs-1; post--){
          o=getState(post);//current neuron's output

          /* If the neuron is in the output layer, calculate the
           * error for the specified output and update the global
           * error.
           * If the neuron is in a hidden layer, backpropagate and
           * sum the related error gradients from the next layer.
           */
          double  err  = 0;
          boolean isOutput = ( (numNeurons - post) <= numOutputs );
          if(isOutput) {
            int index=(numNeurons-post-1);  //going backwards
            err = desiredOutput[index] - o;
            globalError += ( err*err );
            error[post]  = err;
          } else {
            for(int postpost=numNeurons-1; postpost>post; postpost--){
              for (int i=0; i<2; i++) {
//Fuzzify the sum to ensure [-1,1]
                err += (gradient[postpost][post][i] *
                                                      sumTerm2[postpost][post][i]);
          }
        }
        if (Math.abs(err) > 1) {
          System.out.println("Bad: |summation| > 1");
        }
      }
      inputArray = getInputArray(post);
      /* Update the weight of each neuron that feeds into post.
       *
       * Using the extended generalized delta rule is:
       * deltaw = learning rate * gradient * last derivative
       *
       * Where the last derivative is fuzzified using an
       * implication.
       */
      for(int pre=0; pre<inputArray.length; pre++){
        int preNum = getNeuronNumber(inputArray[pre]);

        if ( inputArray[pre].id < 0 ) {
          continue;
        }

        /* The error gradient using the extended generalized
         * delta rule is:
         *        err * fprime * netprime
         * For linear activation function, fprime = 1.
         */
        double[] netprimes = getPrimes(post, pre);
        double[] lastImpl  = getLastImpl(post);

        double[] dw = new double[netprimes.length];
        for (int i=0; i<dw.length; i++) {
          double grad = err*netprimes[i];

          //Calculate change in weight

          dw[i] = LearningRate * grad * lastImpl[i];

          //save the weight change and gradient[post][pre]:
          sumTerm2[post][preNum][i]          = lastImpl[i];
          lastWeightChange[post][preNum][i] = dw[i];
          gradient[post][preNum][i]          = grad;
        }

        //change the weight:
        try{
          System.out.println("Neuron " + post + " from " + preNum + "\t");
          deltaWeight(inputArray[pre], post, 0, dw, maxDW);
        }
        catch (Exception e){
          System.out.println("Error while updating weights "
                + e.toString());
        }
      }

    }
  } //end of record iteration
} //end of epoch iteration
System.out.println("finished training run update, Global Error is now : "
                                                            + globalError);
  }

  // return the derivative of the transfer function of neuron n
  public double[] getPrimes(int n, int i){
    return ((BCKNeuron)neurons.elementAt(n)).netprimes(i);
  }

  // return the last part of the derivative
  public double[] getLastImpl(int n){
    return ((BCKNeuron)neurons.elementAt(n)).getLastDerivative();
  }

} // BCKFNNBprop
```

B.6 BCKSynapse.java

```
package BCK.ANN;

import java.util.*;
import java.io.*;

/**
 * Model a connection from one the input of one
 * neuron to the output of another neuron
 */

/**
 * Modified May, 2002
 * C. H. Poskar
 *
 * Added support for multi-weight synapses:
 *   - weights[], PRIVATE storage for an array of weights.
 *   - constructors, added paramerised constructor & modified others accordingly.
 *   - deltaWeights(), a function to update weights[].
 *   - deltaWeights(), update weights[] using normalized delta and norms.
 *   - getWeights(), a function that returns weights[].
 *   - setWeights(), a function to set weights[]
 *
 * TODO:
 * Need to keep weight, because it was made public, and may be referenced
 * somewhere. Otherwise, can change it to an array of 1.
 */

public class BCKSynapse implements Serializable{

  public  BCKNeuron output;  //a reference to the pre-neuron
  public  int       delay;   //the time delay
  public  double    weight;  //the weight of the connection
  private double[]  weights; //the weight of the connection

  //default constructor
  public BCKSynapse () {
    this.output  = null;
    this.weight  = 0.0;
    this.weights = null;
    this.delay   = 0;
  }

  //parameterised constructor - single weight:
  public BCKSynapse(BCKNeuron output, double weight, int delay){
    this.output  = output;
    this.weight  = weight;
    this.weights = null;
    this.delay   = delay;
  }

  //parameterised constructor - weight[]:
  public BCKSynapse(BCKNeuron output, double[] weights, int delay){
    this.output  = output;
    this.weight  = 0;
    this.weights = new double[weights.length];
    System.arraycopy(weights, 0, this.weights, 0, weights.length);
    this.delay   = delay;
  }

  //copy constructer:
  public BCKSynapse(BCKSynapse s){
    this.output = s.output;
    this.weight = s.weight;

    if (s.weights == null) {
      this.weights = null;
    } else {
      this.weights = new double[weights.length];
      System.arraycopy(s.weights, 0, this.weights, 0, s.weights.length);
    }

    this.delay = s.delay;
  }

  //accessor functions
  public void deltaWeight(double dw){
    weight += dw;
  }

  public void deltaWeights(double[] dw){
    for (int i=0; i<dw.length; i++) {
      weights[i] += dw[i];
    }
  }

  /* Method to calculate multi-weight update using normalized delta weight
   * and triangular norms rather then arithmetic add/sub.
   */
  public void deltaWeights(double[] dw, int maxDW){
    BCKFuzzyNeuron fn = new BCKFuzzyNeuron();

    for (int i=0; i<dw.length; i++) {
      double wNorm = Math.abs(dw[i]/maxDW);

      if (dw[i] > 0) {
        weights[i] = fn.s_norm(weights[i],wNorm);
      } else if (dw[i] < 0) {
        weights[i] = fn.t_norm(weights[i],(1-wNorm));
      }

      if ((weights[i] < 0) || (weights[i] > 1)) {
        System.exit(-1);
      }
    }
  }

  public void setWeights(double[] newWeights){
    this.weights = new double[newWeights.length];
    System.arraycopy(newWeights, 0, this.weights, 0, newWeights.length);
  }

  public double getWeight(){
    return weight;
  }

  public double[] getWeights(){
    return weights;
  }

  //getActivation returns this synapse's contribution
  //to the activation of postsynaptic neurons
  // - Only applicable to single weight conditions
  public double getActivation() throws Exception{
    return output.getState(delay)*weight;
  }

  //return reference to presynaptic neuron:
  public BCKNeuron getNeuron(){
    return output;
  }

  //return the delay implemented:
  public int getDelay(){
    return delay;
  }

}//BCKSynapse
```

Appendix C: Hardware Implementation of a Fuzzy Neuron

This appendix explains the original hardware design for a reconfigurable fuzzy neural processing element. This was originally published in [57].

C.1 RFN - General Design

The design of the reconfigurable fuzzy neuron (RFN) encompasses three distinct components, as illustrated in Figure C.1. The main computational unit is the reconfigurable fuzzy processor (RFP). It processes the inputs and produce the outputs during both regular operation and during learning. The memory unit (MU) stores the parametric values (weights and references), as well as any intermediate values such as the input values over each iteration that are necessary for learning. The Learning Unit (LU) determines parametric updates during the training of the network.

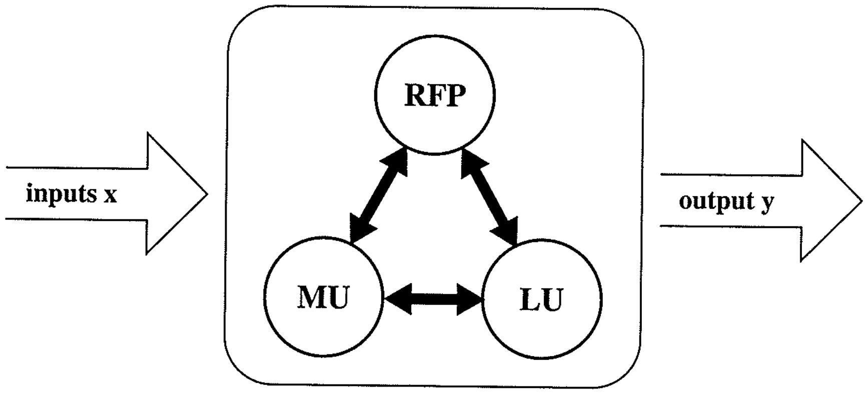


Figure C.1 : Schematic representation of the main parts of a fuzzy neuron; the reconfigurable fuzzy processor (RFP), learning unit (LU) and memory unit (MU).

C.1.1 The Reconfigurable Fuzzy Processor

The specifications of a problem will directly induce the structure of the fuzzy neural network. This occurs at two distinct levels embracing structural and parametric modifications by the processors. First, the qualitative specifications call for the *reconfigurability* of the processor. The detailed quantitative specifications require modifications to the connections of the neurons. In other words, reconfigurability is associated with structural learning while the changes of the connections deal with parametric learning. The structure proposed in Figure C.2 is aimed at assuring flexibility at these two levels. Additionally, the parametric learning effectively performs structural pruning by adjusting the interconnect strengths to the point where an input may have no affect on the neuron output.

In examining Figure C.2, the structural reconfigurability is emphasized, while the parametric generality is inferred. Four control (configuration) signals provide the structural flexibility, by allowing one RFP to be configured (by means of two switching blocks) to any of the previously discussed aggregative or referential neurons. A single signal ($\mathtt{Ref/\overline{Agg}}$) is used to configure the neuron as Referential, or Aggregative. Both modes allow for parametric learning of interconnect strength (or weights). In addition the referential mode incorporates parametric learning of the reference values. The second signal ($\mathtt{OR/\overline{AND}}$) configures the neuron to perform either a disjunctive or conjunctive aggregation. The remaining two signals serve to select the referential computation to be performed, and will be discussed directly.

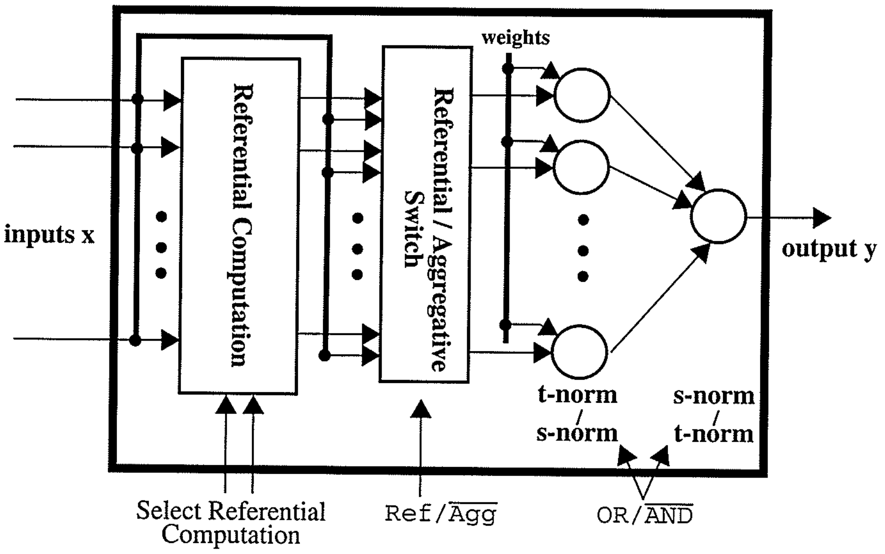


Figure C.2 : The reconfigurable fuzzy processor - general architecture.

The Referential Computation Unit

The referential computation unit is used to implement each of the four referential types, and then to select the desired referential operation.

Consider a single input (x_i) to a referential neuron. Figure C.3 shows the referential computation component for this input. The MATCH of the input and the reference point (r_i) is then computed as in (12), using a fuzzy implication operator over the four combinations of $\{x_i,\bar{x}_i\}$ and $\{r_i,\bar{r}_i\}$, two logical ANDs, and a sum of inputs weighted by 1/2. As expressed in the previous section this referential calculation contains the DOM and INCL operations. Furthermore, the DIFF operation is simply the complement of the MATCH.

The desired referential output (ref(x_i;r_i)) as well as the original input (x_i) are both passed to a switch. This switch makes a selection based on whether the neuron is configured as a Referential or Aggregative neuron. In fact, we can (and as was the case with deriving the learning rules for the weight parameter do) consider the aggregative neuron to be a referential neuron whose function is simply to pass the original value. That is, ref(x_i;r_i) = x_i. The result, u_i, is then passed to the next stage, for aggregation. In this manner a single aggregative unit can be designed independently as it only sees an arbitrary input, u_i.

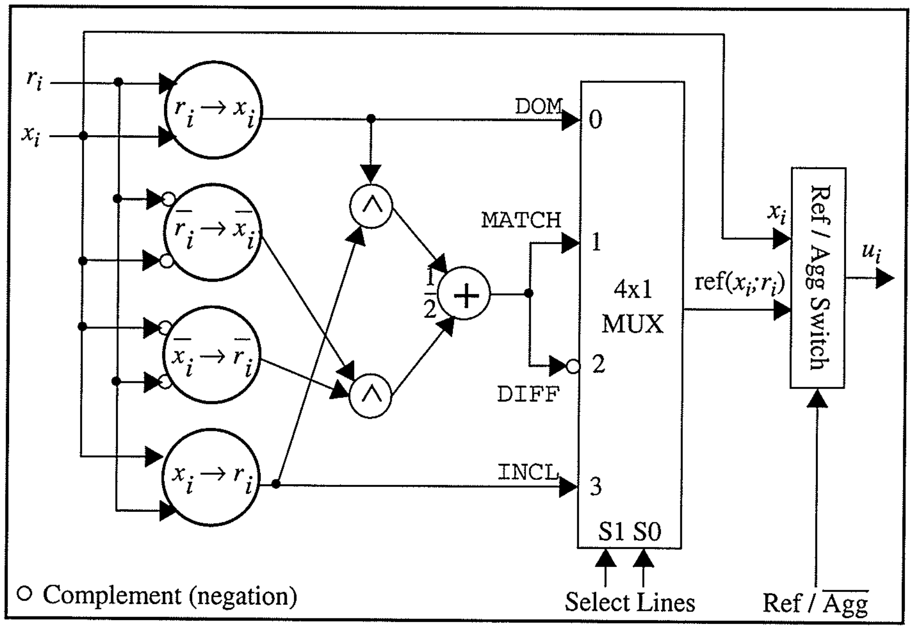


Figure C.3 : Multiplexing basic functions of referential computation - equality or matching, difference, dominance, and inclusion.

It is worth noting that the implication operation is not commutative, and as such all four implication operations are necessary.

Each neuron requires at least one referential computation unit for each input that is to be processed in parallel. This issue will be discussed more in the section on the interconnect network.

The Aggregative Unit

The aggregative unit performs the aggregation or combination of the multiple input contributions to from the output for the network.

The RFP in Figure C.2 can be configured as either a disjunctive or a conjunctive referential neuron. A conjunctive (AND) configuration is obtained by ordering the triangular norms as first **t** then **s**, while reversing the connectivity creates a disjunctive (OR) configuration. In this manner, the aggregative unit can be made up of a generalized unit similar to that illustrated in Figure C.4.

When the control signal is high, the neuron is configured to perform disjunctive aggregation. The input, u_i, and the weight element, w_i, are passed the triangular t-norm operation. The result of this operation is then provided to the triangular s-norm operation which 'sums' it with the previous result, to generate the cumulative OR, as in (4), which then becomes the neuron output. Similarly, when the control signal is low, the order of the operations are reversed, creating a cumulative 'product' as in (6).

For each input to be processed in parallel this unit requires one full set of triangular norms (s-norm, t-norm and feedback signals). In the case of a disjunctive aggregation the switch is configured so that each of the parallel t-norms are passed one of the inputs along with its corresponding weight parameter. The output of each of these are then passed to one of the inputs of each of the parallel s-norms. The remaining input for each of these s-norms is connected to the output of the preceding s-norm, with the first s-norm connected to a logical '0' signal. This forms a cascading accumulation. Note however that only one output switch is necessary; connecting only the last of the parallel triangular norms to provide the output from the cascade.

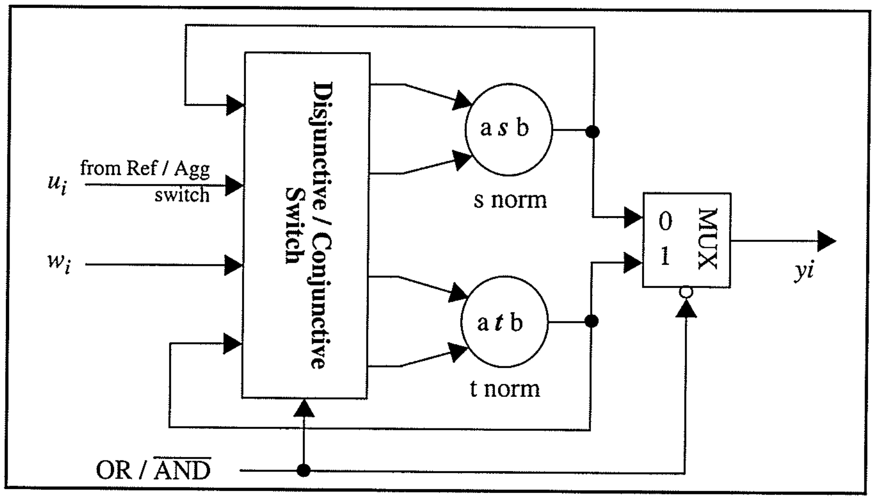

Disjunctive / Conjunctive Switch
u_i
from Ref / Agg switch
w_i
a s b
s norm
a t b
t norm
MUX
0
1
yi
OR / $\overline{\text{AND}}$

Figure C.4 : Generalized aggregative unit. Configurable as conjunctive or disjunctive.

An alternative configuration is to add a second triangular norm layer with a dedicated parallel t and s-norm circuit, to perform the accumulation for either a parallel conjunctive or disjunctive aggregation.

Ordinal Sum

An interesting generalization of t-norms emerges in the form of a ordinal sum as referred to [74]. This construct is developed as a combination of a countable number of t-norms where each of them is defined in some region of the unit square, see Figure C.5. More formally, the ordinal sum of t_j with respect to $\{(a_j, b_j)\}_{j \in J}$ is defined as

$$x \ t \ y = \begin{cases} a_j + (b_j - a_j)\left[\dfrac{x - a_j}{b_j - a_j} \ t \ \dfrac{y - a_j}{b_j - a_j}\right], & \text{if (x,y) belongs to } \mathit{some}\ (a_j, b_j) \times (a_j, b_j) \\ min(x, y) \qquad , & \text{otherwise} \end{cases}$$

Let us assume that the number of the t-norms in the above construct is fixed. Similarly these t-norms are specified in advance. Then we generalize the OR neuron by admitting the ordinal sum of the t-norms at the level of synaptic computation. In addition to the connections (**w**) encountered in the previous neuron, its generalization comes equipped with an auxiliary vector 2J - dimensional summarizing the domains of the contributing t-norms, say [**a b**] where

$$\mathbf{a} = [a_1 \ a_2 \ \ldots \ a_J]$$
$$\mathbf{b} = [b_1 \ b_2 \ \ldots \ b_J]$$

We may express the processing carried out by the neuron in the form:

$$y = \mathrm{OR}(\mathbf{x};\ \mathbf{w},\ \mathbf{a},\mathbf{b})$$

by making a clear distinction between the inputs and the parameters of the neuron.

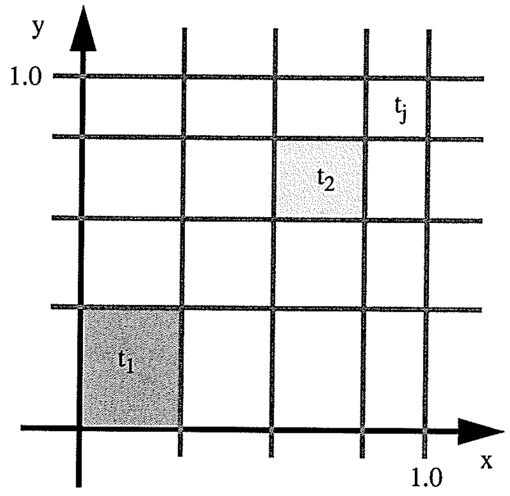


Figure C.5 : Construction of the ordinal sum.

The generalization of the AND neuron can be proposed in the same way; obviously an s-norm associated with the ordinal sum of t-norms is obtained via the De Morgan rule.

Implementing such a summation would require the implementation of not just a single pair of triangular norms but one or more families of them (in addition to the default pair). A comparator would then select which one of these norms would actually be used. Figure C.6 illustrates this manner of aggregative implementation.

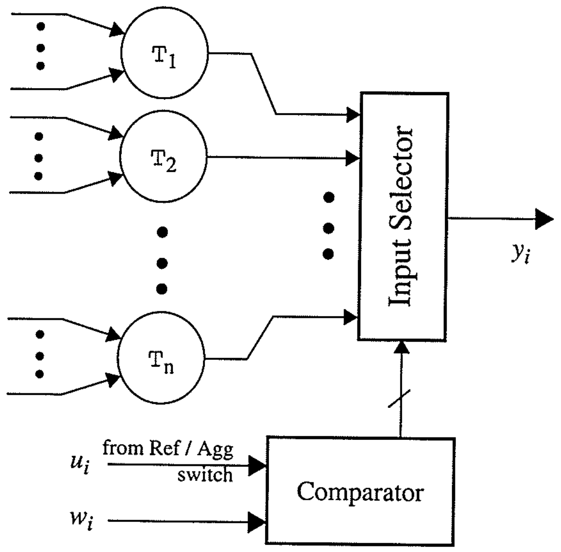


Figure C.6 : Ordinal sum of disjunctive (OR) neuron.

Since the ordinal sum occurs at the point of aggregation, it may be necessary to implement 'n' full sets of the triangular norms. One alternative is to use a configurable family of norms, where the comparator is used to choose a parameter with which the particular family member is specified. A second alternative (as with the regular parallel aggregation) is to add a second triangular norm layer with a series of dedicated parallel t and s-norm circuits.

C.1.2 The Learning Unit

The Learning Unit for the RFN is used to perform parametric learning which consists of updating weight and/or reference parameters. The LU can be subdivided into four general components for either form of parametric learning. The following discussion will highlight each of these components from the point of view of weight updates, pointing out which section(s) require augmentation for referential learning.

The four components of the LU are illustrated in Figure C.7, and correspond to (23) and (26). Notice that there are actually two similar configurations. The reason for this is due to the difference in computation of the backpropagation algorithm for neurons at the output layer and neurons at a hidden layer.

During training the error between the target value and the computed output can be calculated for neurons in the output layer. This error value can be directly utilized to alter the weighted connections to this layer. However the neurons in the hidden layer have no specific target value for comparison. It is therefore necessary to determine the error contribution of each hidden neuron to the output neuron. This is performed through the backpropagation of the partial error derivatives from the output layer successively back through each hidden layer.

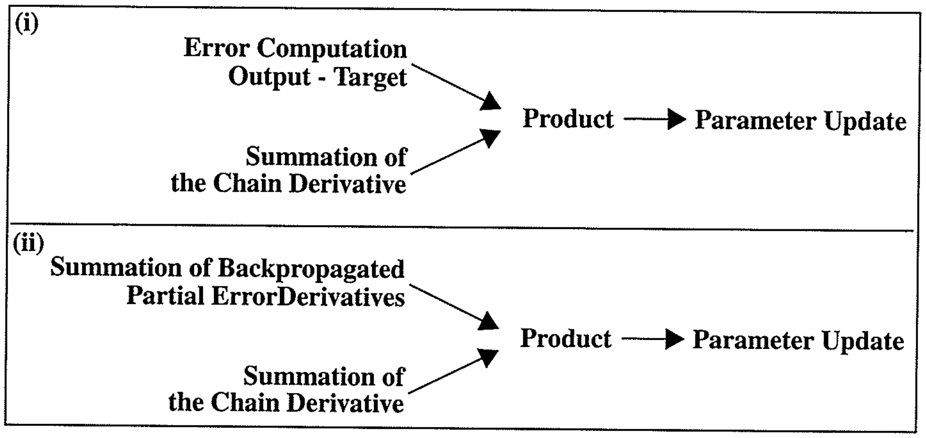

(i)
Error Computation Output - Target
Summation of the Chain Derivative
Product
Parameter Update
(ii)
Summation of Backpropagated Partial ErrorDerivatives
Summation of the Chain Derivative
Product
Parameter Update

Figure C.7 : The components of the learning unit for (i) a neuron in the output layer, and (ii) a neuron in a hidden layer.

The first component represents the first part of the derivative of the error metric given by (26) (for an output neuron). The idea is the same, although the circuit is different for this component in a hidden neuron. The hidden neuron will need to perform a summation with contributions from each neuron in the next layer.

The next component performs the accumulation of the chain derivative, the multiplicand of (26). This must be computed for each interconnect weight to be modified.

It is this component where weight and referential learning differ. The structures for this component is illustrated in Figure C.14 for the weight and reference parameters.

The third component produces the error derivative with respect to each weight, as specified by (26). This is then used to perform weight update as specified by the learning rule, (24).

At this point learning can be performed off-line, as an algorithm running on a serial microprocessor. Alternatively it can be implemented in whole or in part in hardware. In part the hardware is already present in the RFP, which performs all of the intermediate summations with the addition of simple control logic and a few registers for intermediate values, with the product and parametric update performed off-line. This is most useful for highly parallel systems where the MU is also off chip, in standard memory (thereby allowing considerably more area to be allocated to the RFPs).

C.1.3 The Memory Unit

The Memory Unit is the simplest unit, consisting of a standard RAM. The MU can often be the limiting factor in network design if implemented on-chip. That is due to both the data size, and potentially the layer size being preset by the MU.

For each input to be processed (in series or in parallel), a neuron requires storage for a weight parameter, a reference parameter, and with in-situ learning an input value. The MU can be saved and reloaded to add storage, but this is at a cost of time. Also with additional hardware the MU can be flexible in a trade-off between data size and the number of inputs it can handle.

When implementing a neuron with both excitatory and inhibitory components, as discussed in [57], it is not necessary to fully replicate the MU. Thus, each input value for the neuron needs to be stored only once to accommodate both structures.

C.2 The Network Design

A generic network consists of three components; (i) Logic Units - consisting of general or dedicated processing unit(s) and/or memory elements, (ii) Interconnection Network - providing communication between Logic Units, and to any external I/O as required, (iii) Interconnection Interface or Controller - providing the connection between the Interconnection Network and the Logic Units.

C.2.1 The Logic Unit

Many designs already exist for a wide variety of Artificial Neural Networks (ANN's). These designs incorporate a variety a learning methodologies, either on or off line, utilizing analog, stochastic, or digital circuitry [75][76][77][78][79][80][81][82][83]. While the analog circuits offer advantages in the areas of speed and size, their design process is typically far more complex, generally requiring full custom designs. Digital circuits tend to promote reusability and ease of design, but can be severely limited by high precision computational overhead. Stochastic designs offer a middle ground reducing computational overhead and offering scalable precision.

A fuzzy processor is an approximate reasoning element. It is therefore acceptable to utilize low precision operations on fuzzy sets, which are more readily implemented in a digital environment, than a traditional ANN. Furthermore, being a logic processor, the operations needed to perform fuzzy computations are easily and economically implemented without the need for custom analog parts. Therefore the design of the RFP will utilize digital circuits. This offers the additional advantage of being able to re-use existing "off the shelf technology", to implement the circuit.

This still leaves us with the choice of using standard digital or stochastic circuitry. Both offer ease of implementation utilizing digital components, and lend themselves to the basic logic operations being performed by the RFN. Stochastic processing offers two major advantages, data precision scalability and parallelism. Each stochastic signal regardless of precision requires only a single line, and basic logic components for both logical and arithmetic product and sum, allowing for large parallel structures to be implemented. Digital implementation requires multiple parallel lines to carry each data and the precision must be set to construct the computational units.

However, stochastic circuitry have other overheads, including the need for pulse generators to create the stochastic streams. When implemented on-chip, the feasibility of large parallelism is reduced. Furthermore, as already stated, fuzzy processing is implemented in low precision computations and fixing the precision to a small size (say 8 bits) is quite reasonable[1].

The RFN will therefore be implemented in digital hardware. However we feel the potential of stochastic processing should not be abandoned completely, and will be more fully investigated in the future.

1. In fact most fuzzy inference controllers are implemented in 8-bit microprocessors such as the MC68HC11.

C.2.2 The Interconnection Network

Many network architectures have been proposed to perform interconnection between processing units in a network. These range from the fully connected [82], to linear systolic arrays [84]. While the fully connected network would seem to be the natural choice in theory, in practice they are not feasible for a generic scalable architecture. Moreover, the linear systolic array has extreme communication latency. This is particularly true for a layered network implementation of the backpropagation algorithm that passes information bidirectionally.

The FNN being considered is a multi-layered feed-forward network. A typical feed-forward neural network is shown in Figure C.8. The network consists of layers of processing units and weighted interconnections between these layers. This architecture lends itself directly to a multi-bus based connection, allowing broadcast between layers. That is, a two bus system for each layer, the first an input bus for layer *i* and the second an output bus for layer *i*.

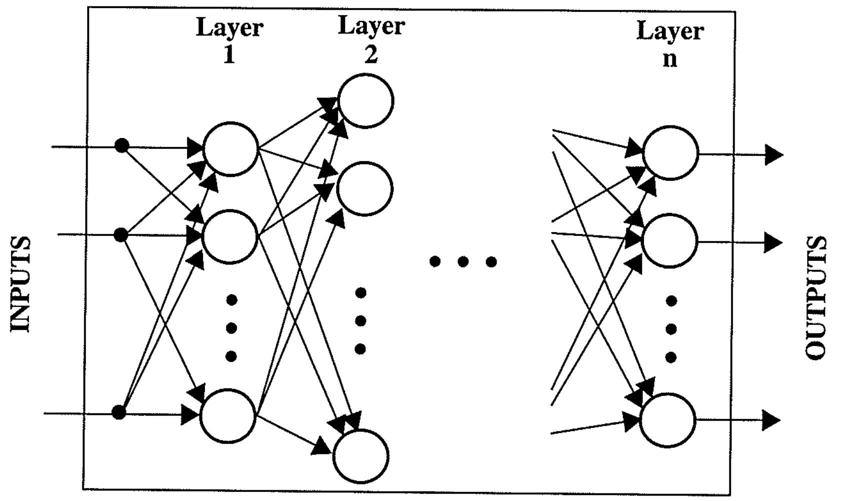


Figure C.8 : A feed-forward neural network.

It is clear from the figure that the input bus for layer *i* is also the output bus from layer *i-1* under regular feed-forward mode. Less obvious is that during backpropagation it is connected to the output bus from layer *i+1*. Similarly the output bus for layer *i* is also the input bus for layer *i+1* (in regular mode), and the input bus to layer *i-1* (in learning mode). As this implies, the data and address lines of every second bus are connected, with independent control lines for each bus, to determine which layer is the recipient of the transmitted information.

The buses are connected by individual bus controllers. These controllers provide local bus control signals (to transfer temporary control of the data and address portion of the bus to individual neurons) and to provide global control signals determining the modes of operation, which include;

- *initialization* - to activate and configure the neurons in a layer, and pre-load weight values,
- *regular* - feed-forward neural processing of inputs, and
- *learning* - backpropagation of errors and weight updates.

These controllers are then connected in series forming a linear array of buses (or layers), and acting as a finite state machine in either linear fashion (regular fuzzy processing), or bidirectional (training mode).

The content of the buses fall into three categories of signals; data, address, and control. Figure C.9 illustrates these signals, specifying the control signals in the two major modes of operation; *initialization* and *processing*.

Figure C.9a and Figure C.9b show that there are seven shared control lines, providing seven control signals in each mode, five of which are standard across both modes. These are:

- *Mode_S1,* used to toggle between the initialization and processing modes,
- *Bus_Input*, which enables a particular layer to access shared bus information,
- *Data Ready*, a handshaking signal, automatically reset after each clock,
- *Token In* and *Token Out*, used to form a serial line connecting all neurons in one layer, for the purpose of transferring control of each layers' output bus.

In addition to these common signals are two signals utilized during initialization (Mode_S1 high),

- *Use / Load*, which enables an unused neuron (Use), or can signal pre-load of the weight and reference memory ($\overline{\text{Load}}$), and
- *Data Select*, toggles access between weight and reference storage,
- and two signals used for the processing mode (Mode_S1 low),
- *Sign bit*, which indicates the sign of the backpropagated error signal, and
- *Mode_S0*, which toggles between the regular processing and learning modes.

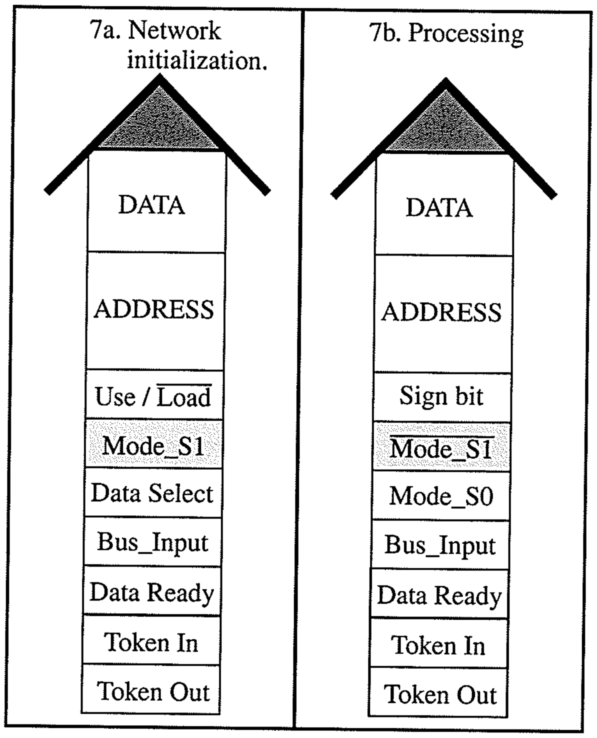


Figure C.9 : The network signals for (a) initialization, and (b) regular processing.

C.2.3 The Interface

The interface unit, shown in Figure C.10, consists of a general control unit and four registers. The registers are used to buffer the data and addresses from the input and output buses.

In this implementation only a single data bus is used between layers, therefore only a single input can be processed at one time. Thus, the purpose of the network interface is to:

- receive the input data, address, and control signals from the network,
- generate the internal signals for the individual neural processor,
- regulate communication of the data to and from the neural processor, and
- prepare and co-ordinate transmission of the output data packet.

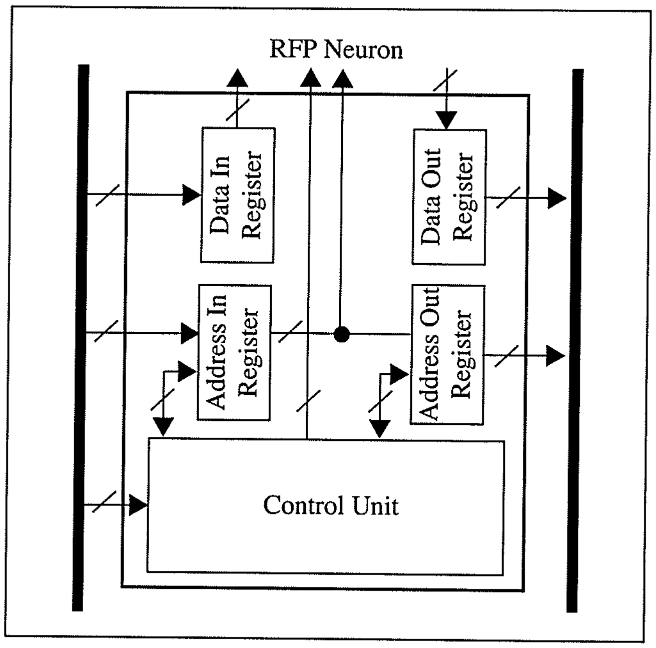


Figure C.10 : Schematic of the network interface.

The control unit contains the local address (with respect to the bus layer), a simple counter to generate addresses during learning, and the control logic of the neuron. Based on the mode, bus input control signals and the input address, the control logic determines whether the data is for 'it' or not. In regular mode the input address is the 'from' address, and is always valid. For the initialization mode, the input address is the 'to' address, and must be checked. During learning mode the input address is first treated as 'to' (for backpropagation from higher layer), and then generated (using the counter) to perform local weight updates. Similarly, in regular mode the output address is the local neuron address, while in learning modes it is the counter address.

C.3 Hardware Implementation

The hardware implementation will focus on the RFN (illustrated in Figure C.11) with in-situ learning. The particular design of the RFP and LU are presented in the following sections.

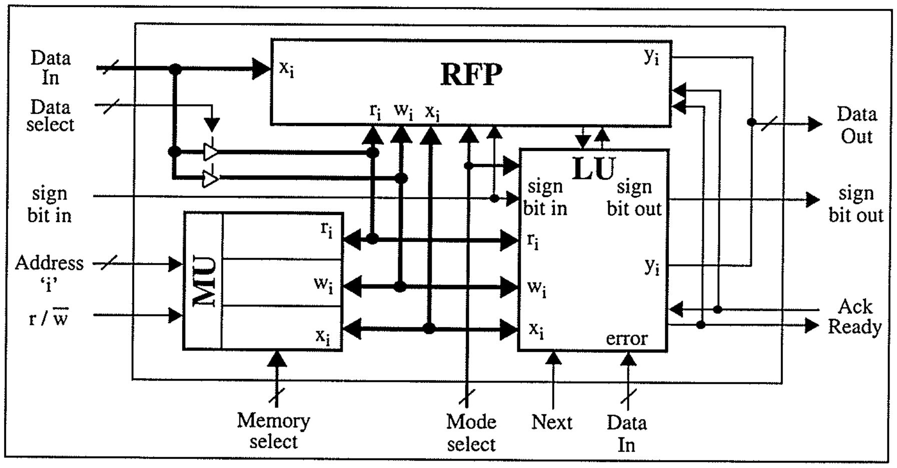


Figure C.11 : Complete RFN design flow.

C.3.1 Referential Computation Unit

The reference neuron is implemented using the Godel implication (operator) defined by (28). The four implication operations indicated in Figure C.3 are performed by analysis of a single comparison of the operands (shown in Table 5). The MATCH function may be directly implemented following the computational flow of Figure C.3. However an analysis of the operations that take place reveal that the two AND functions will always combine one datum with a logical '1'. This results in the original datum from each AND being summed and weighted (multiplied) by 1/2. Rather then duplicating effort, the operators are combined through the use of two multiplexers which choose the appropriate set of inputs to be summed. This is illustrated with the circuit in Figure C.12. Thus if $r_i > x_i$ then x_i is summed with $\bar{r}_i$, otherwise $\bar{x}_i$ is summed with r_i. The weighting is accomplished here with a simple hardwired shift of the m-1 MSB's (assuming m bit precision), one position to the right (becoming the m-1 LSB's), and the carry out from the adder becomes the m'th bit. The DIFF computation is then simply the inversion of the MATCH operator.

	$a > b$	^	+	$a \leq b$	^	+
$a \rightarrow b$	b	**b**	b + a	1	**a**	a + $\bar{b}$
$b \rightarrow a$	1			**a**		
$\bar{a} \rightarrow \bar{b}$	1	a		b	b	
$\bar{b} \rightarrow \bar{a}$	a			1		

Table 5. Evaluation of the implication operator and MATCH function.

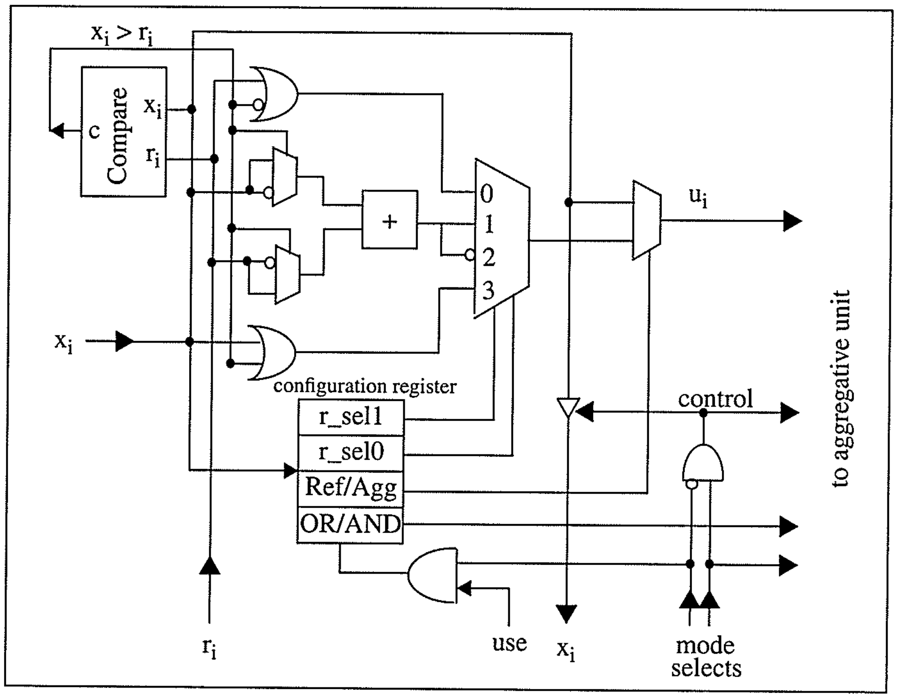


Figure C.12 : The Referential Computation Unit

While this simplifies the `MATCH` operation, the `DOM` and `INCL` do not emerge as easily. These two operators perform a single implication operation as described above. Again the result will be either a datum (if the implication is true), or a logical '1' (if false). By utilizing an `OR` operations of the appropriate datum and control signal (as illustrated in Table 6), we may easily generate the correct referential computation for either `DOM` or `INCL`.

	$a > b$	$a \leq b$
$a \to b$	$c = 1$ $\mathbf{a \to b = b}$	$c = 0$ $\mathbf{a \to b = 1}$
$a \to b$	$b + \bar{c}$	

Table 6. The INC or DOM function.

The Godel implication was chosen simply because the 'datum' referred to above is simple and required no additional operators. Other implications may similarly reduce, such as the Lukasiewicz implication (27), or reduce in other ways or even not at all, requiring the full circuit described in Figure C.3.

The result from the desired operation, u_i, is then selected and passed to the Ref/Agg switch, along with the original input. The input value is stored in the MU to be used again if learning is performed. The reference unit has control of the lines connecting itself to the MU (and the LU) only during regular processing mode. During learning the input value may come from either the MU or the LU.

The configuration of the neuron occurs during initialization. The configuration bits are loaded into the configuration register at the same time that the interface is set in use, and loads its neuron address. In addition to specifying the type of neuron being implemented the configuration bits are used to indicate the learning mode. That is, whether to perform learning on the weights, the references, both, or neither if learning were to occur.

C.3.2 Aggregative Neuron Design

The aggregative neuron implements two basic operators; a t-norm and an s-norm. These triangular norms are implemented using the Lukasiewicz norms, equations (29) and (30). Specific triangular norms were chosen over less complex alternatives because they accommodate interaction between their arguments [54].

Figure C.13 shows the aggregative neuron with its two triangular norm operators. Both use the carry signal as a control of the respective result. The s-norm is an upper limited sum, meaning that it cannot exceed a maximal value of 1 (represented by all bits high). If a carry occurs, the result must be changed to 1. This is accomplished by using an OR operation over each of the sum bits with the carry bit. Similarly, the t-norm is a lower limited sum, (a+b-1), with a minimum value of 0. In this case, the carry must be set in order for the sum to be non-zero. This is accomplished by using an AND operator to combine the sum and the carry bit. Additionally, the t-norm operator has an initial carry in value of '1'. This is needed to ensure the property of:

1 t a = a

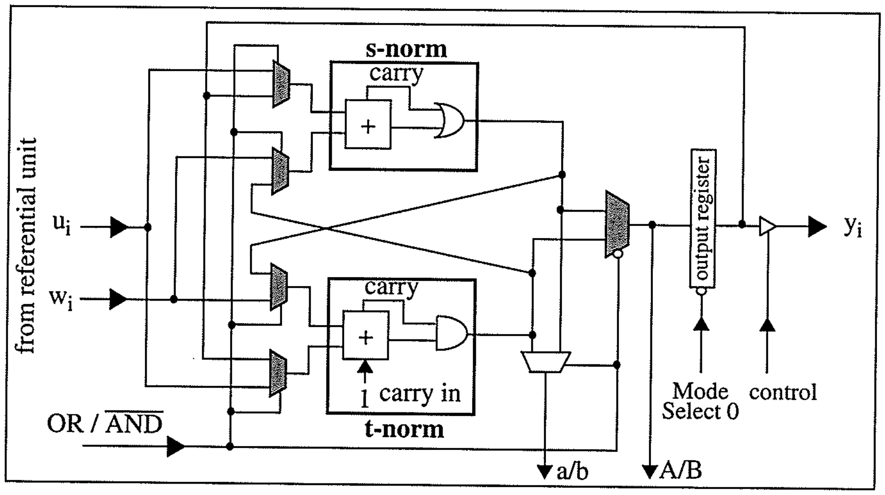


Figure C.13 : The aggregative computation unit.

The switch used to configure the neuron as either conjunctive or disjunctive consists of the five highlighted multiplexers (an additional multiplexer is used for learning). The first four are used to select the input to the t- and s-norm operators, and the fifth chooses which operator's output is summed. During regular processing, the output is stored in the output register, which acts as a cache for the partial sum. After summing all *n* inputs, the result waits to be passed to the output bus. Once a layer has completed passing its results to the next layer, the Mode_S0 line is strobed to reset the output registers (By the identity transform an OR neuron is initialized to 0, and an AND neuron to 1). In the learning mode, two registers from the learning unit are used for holding the partial sum (A/B) and the i'th term (a/b) specified by equations (33) and (39).

C.3.3 Implementing the Learning Unit

A dedicated hardware learning circuit for this architecture was proposed in [56] for the weight parameter. Sections 3 and 4 illustrate the symmetry between learning of the interconnect strength for both aggregative and referential types of neurons. Furthermore, the detailed calculations for referential learning uncovered that they differ (between each reference type as well as from the weight parameter updates) only by the final evaluation of the chain derivative. That is, the evaluation of the derivative of the reference function, for the disjunctive and conjunctive forms respectively. The computation of these chain derivatives can be quickly and easily summarized as a conjunction of two or three inclusions, as illustrated in Figure C.14. The `INCL` and `DOM` neurons add an additional inclusion during referential learning, while the `MATCH` and `DIFF` neurons add a sign function (± 1). The sign function is incorporated separately by the LU configuration controller (Figure C.15), along with the global sign bit indicating an increase or decrease to the referential parameter.

The LU, as illustrated in Figure C.15, conforms to the four design components of Figure C.7. The first component depends on which layer the neuron belongs to (hidden or output). In order to design a single general purpose neuron it was necessary to add a configuration bit to the LU to indicate the current layer. This is then used to configure the full adder to first load register 1 (with the target value) and then subtract that value from the output of the neuron. Alternatively, it can give control to the sign bit signal to perform the summation of the backpropagated error derivatives. The resulting multiplier is then stored in register 1. This value is then used for each of the parametric updates performed at that neuron.

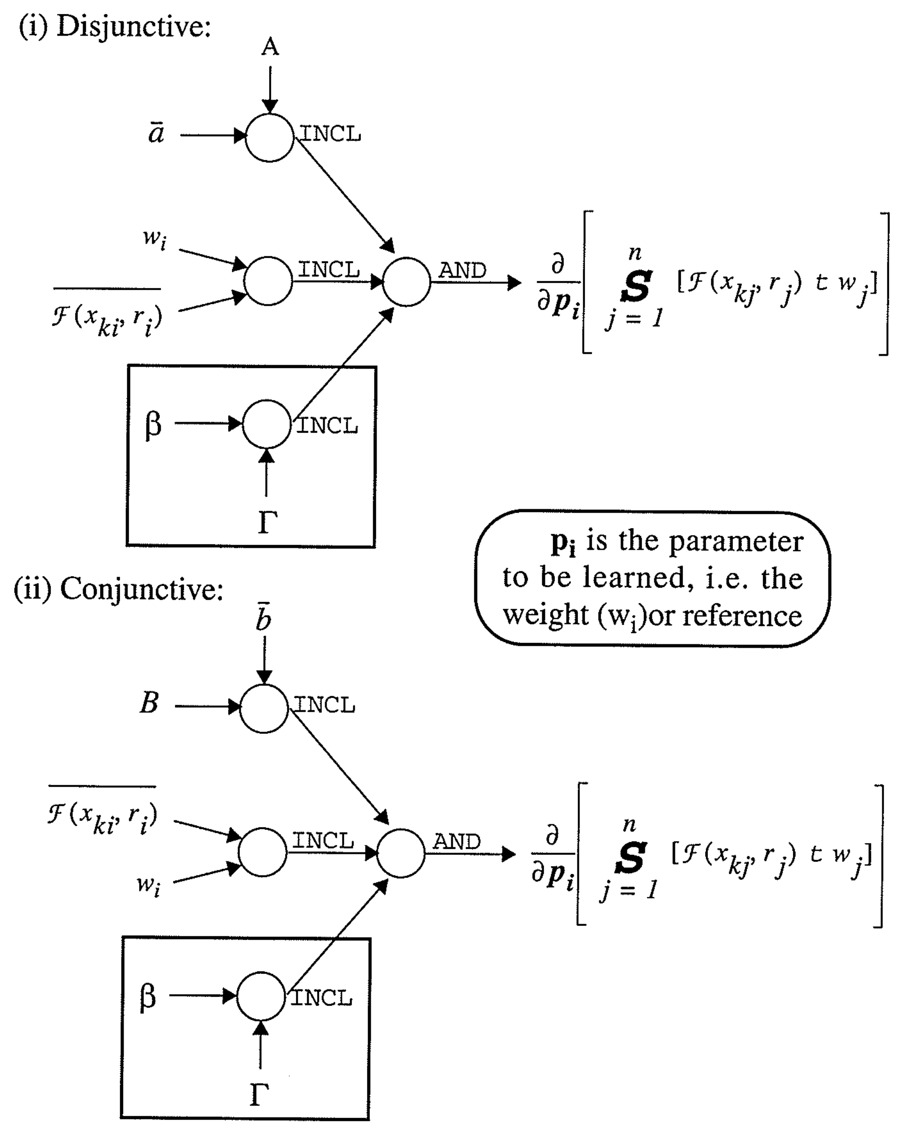
The second component is calculated for each of the updates to be performed. For update to the i'th parameter (either weight or reference) we begin by cycling through each input (stored in the MU) and calculating A/B and a/b. Then the reference function is computed for the i'th input and reference parameter and stored in the u register. The configuration controller then reconfigures the RFP to be a conjunctive `INCL` neuron to perform the aggregation of these registered values and the i'th values stored in the MU to compute the chain derivative (see Figure C.14). This value is then stored in register 2.

The controller also determines the sign function, using the comparator on the RFP, and the reference function type (implemented in a simple lookup table structure). When completed, the controller restores the RFPs original configuration.

The next component is the multiplier. It is used to multiply together the values in the two registers. This calculates the partial derivative of the error due to the i'th parameter, and is ready to be used for both weight update and backpropagation to the previous layer. Due to the low precision, the multiplication is performed using a simple shift-add multiplier. Since the values represent fractions between 0 and 1, it is actually performing division (right shift instead of left shift).

The last component performs the parametric update. The error derivative is multiplied by either a constant (hardwired) or a programmable bit shift (shift register). A similar hardwired shift can be used with the output data to actually equal the error derivative. This value is then used to increment or decrement the parameter being updated. The new parameter value is then stored in the update register, until it can be written to the MU.

There is an additional register in the LU to indicate for which parameter learning is to occur (if it is to occur). Any neuron can still feedback contributions to previous layers partial error derivatives, even if the parameters of that neuron are fixed (i.e. no learning is


(i) Disjunctive:
A
$\bar{a}$
INCL
w_i
$\overline{\mathcal{F}(x_{ki}, r_i)}$
INCL
AND
$\frac{\partial}{\partial p_i}\left[\sum_{j=1}^{n} [\mathcal{F}(x_{kj}, r_j)\ t\ w_j]\right]$
β
INCL
Γ
p_i is the parameter to be learned, i.e. the weight (w_i)or reference
(ii) Conjunctive:
$\bar{b}$
B
INCL
$\overline{\mathcal{F}(x_{ki}, r_i)}$
w_i
INCL
AND
$\frac{\partial}{\partial p_i}\left[\sum_{j=1}^{n} [\mathcal{F}(x_{kj}, r_j)\ t\ w_j]\right]$
β
INCL
Γ

The third INCL is used only for reference parameter learning of the inclusion and dominance neurons as follows:

INCL	$\beta = x_{ki}$	$\Gamma = r_i$
DOM	$\beta = r_i$	$\Gamma = x_{ki}$

Figure C.14 : Multi-valued representation for the chain derivative during parametric learning for (i) a disjunctive neuron., and (ii) a conjunctive neuron.

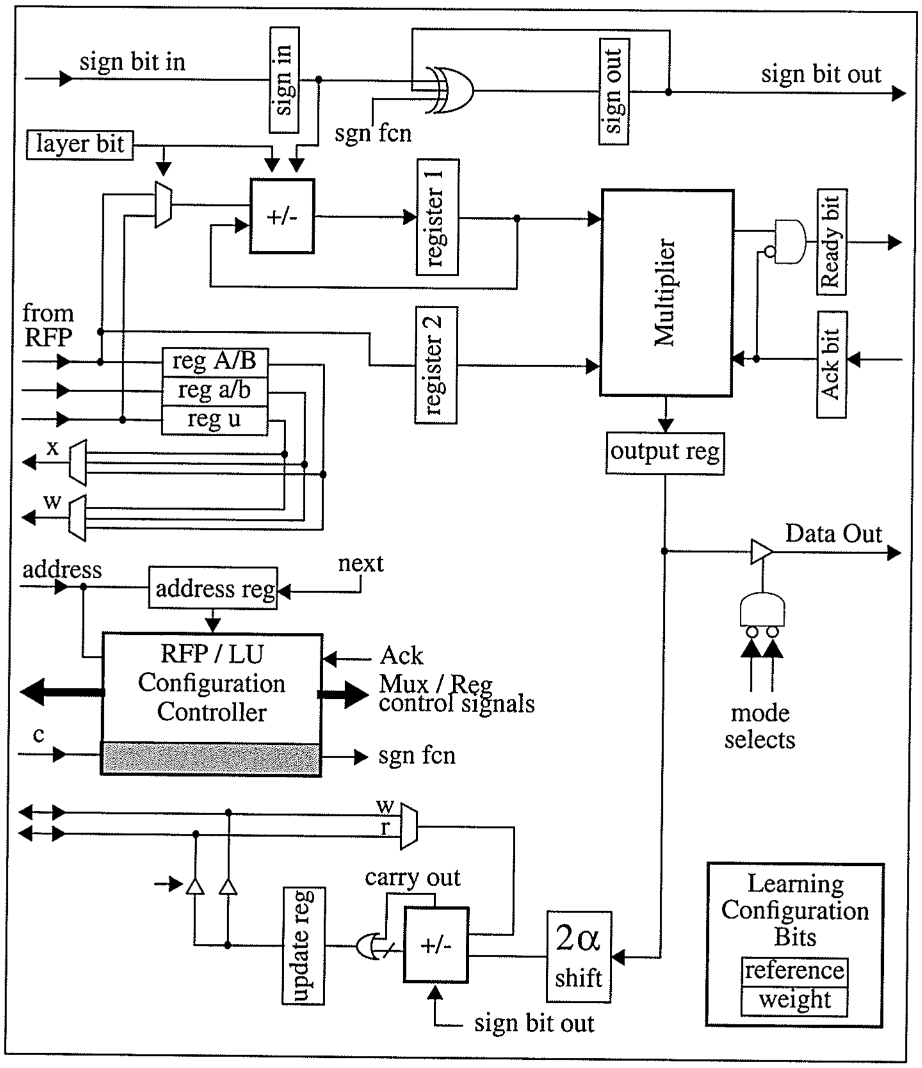


Figure C.15 : The learning unit (LU).

performed). Alternatively, either of the two parameter types can be chosen for individual learning.

Last, is the learning of both parameters together. In this case, Figure C.14 again illustrates the similarity between the two types of learning. The only salient point is to indicate that for networks with hidden layers, the sign bit registers and register 1 must be constructed to contain two values; one for each of the parameters. This is because even though the two parameters begin with the same error, they may of course diverge with each layer that they are fed back. Register 2 needs only to hold one value at a time, reusing the same calculated values for both parameters to determine the chain derivative (although the result may differ).