A Protocol for the Interoperability of Content Distribution Networks

By

Douglas Arthur Johnson
B.Sc., University of Victoria, 2000

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr. G.C. Shoja, Supervisor (Department of CSC)

Dr. E.G. Manning, Outside Member (Department of ECE)

Dr. M. Serra, Departmental Member (Department of CSC)

Dr. P. Driessen, External Examiner (Department of ECE)

© Douglas A. Johnson, 2002
University of Victoria

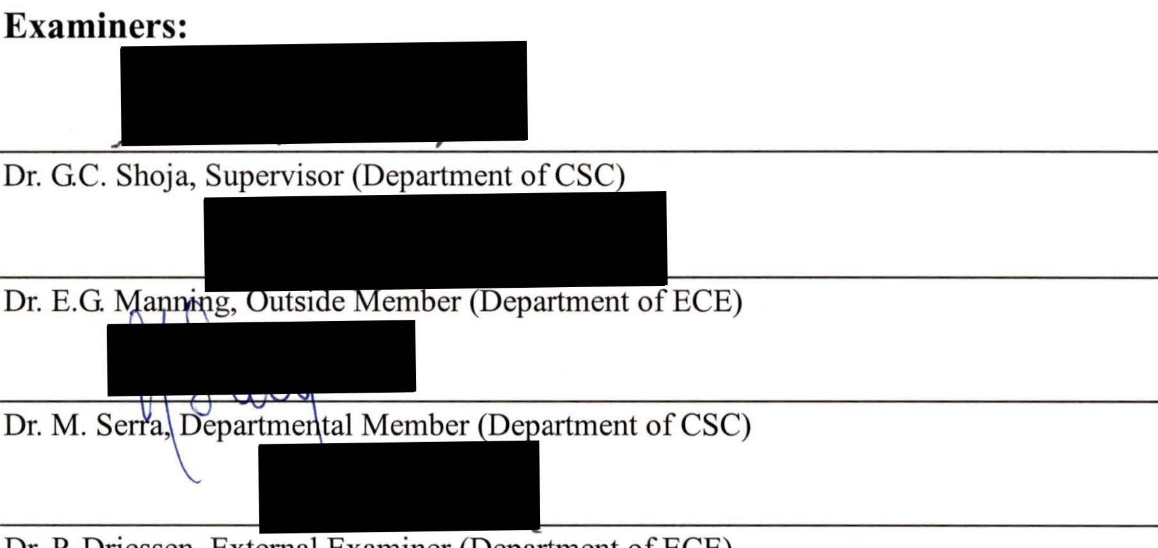
All rights reserved. This thesis may not be reproduced in whole or in part, by
photocopy or other means, without the permission of the author.

Supervisor: Dr. Gholamali C. Shoja

ABSTRACT

The primary purpose of computer networks is to facilitate communication between computers. This functionality extends to distributing content produced by media developers and information gatherers. This has become the primary focus of the Internet, with information taking many forms. Some data is delivered as web pages via HTTP other data may take on any form including delivering large blocks of data representing multimedia content. This process is costly and represents a complex problem of balancing customer demands, network and server loads, and cost. One way to address this problem is to distribute the data to many distribution points. This solution has the limitation of the network's scope and proximity to customers. By connecting several of these networks this limitation can be overcome. This thesis describes a protocol developed to the requirements laid forth by the Content Alliance Group that seeks to provide a framework for this interconnection. This thesis also describes a Best Server Choice algorithm that chooses the best server to deliver content to a client along the fastest route.

Examiners:



Dr. G.C. Shoja, Supervisor (Department of CSC)

Dr. E.G. Manning, Outside Member (Department of ECE)

Dr. M. Serra, Departmental Member (Department of CSC)

Dr. P. Driessen, External Examiner (Department of ECE)

Table of Contents

Title Page i
Abstract ii
Table of Contents iii
List of Tables vi
Table of Figures vii
Acknowledgement ix
1. Introduction 1
1.1.1 Motivation 1
1.1.2 Problem Definition 2
1.1.3 Scope and Focus 3
1.1.4 Methods 4
1.1.5 Outline 4
2. Previous Work 6
2.1 Existing Delivery Systems 6
2.1.1 Browser Caching 7
2.1.2 Proxy Caching 8
2.1.3 Server Caching 9
2.1.4 Server Farms 10
2.2 Content Distribution Networks, CDNs 11
2.3 Request Redirection 12
2.3.1 DNS 13
2.3.2 Transport Layer Request Routing 15
2.3.3 Application Layer Request Routing 15
2.3.4 Combinations of Mechanisms 18
2.4 Content Distribution Internetworking 18
3. Content Peering Gateway Protocol, CPGP 20
3.1 Primary Function 20
3.2 Protocol Scope 21
3.3 Root CPG 23
3.4 Child CPGs 25
3.5 Peering CPGs 26
3.6 CPGP Summary 28
4. Protocol Specification 29
4.1 Matrix Requirements 29

4.1.1 Format of Data Storage 30
4.2 Matrix Specifications 31
4.2.1 Internal Data (idata) 32
4.2.2 External Data (eData) 33
4.2.3 Advertised Data (aData) 33
4.2.4 Processing Data (pData) 34
4.2.5 Calculated Matrices based on Source Matrices 34
4.2.6 Summary of Metric Matrix Fields 35
4.3 CPG Data Exchange Protocol 39
4.3.1 Specification Components 40
4.3.2 Service to be provided 40
4.3.3 Assumptions about the Environment 45
4.3.4 Vocabulary of Messages 46
4.3.5 Message Format 51
4.3.6 CPGP State Machine 53
4.4 Message Sequence Charts 54
4.4.1 Simple Case Message Sequence Charts 54
5. Protocol Validation 56
5.1 Connected to Root 57
5.1.1 Connected to Root State Diagram 59
5.1.2 State Invariants (Connected To Root) 60
5.2 Connected to Peer 66
5.2.1 Connected to Peer State Diagram 67
5.2.2 State Invariants (Connected to Peer) 67
6. Best Server Choice (BSC) Algorithm Formalization 72
6.1 Algorithm environment 73
6.2 Algorithm structure 74
6.3 Algorithm Efficiency 74
6.3.1 Code Listing for BSC algorithm 76
7. Prototyping and Testing 79
7.1 CPGP Test Suite 79
7.1.1 Program Environment 79
7.1.2 Program Overview 79
7.1.3 Getting Started 80
7.1.4 Using the program 82
7.1.5 Testing Requests 89

7.2 Live Test 90
7.2.1 System overview 91
7.2.2 Setting up the Live Video On Demand Demo 97
7.2.3 Live Test Results 98
7.2.4 Live Test Conclusions 99
8. Conclusions 100
8.1 Major Contributions 100
8.1.1 The CPGP Test Suite 100
8.1.2 The Live Test: On Demand URL Rewriting 101
8.2 Future Work 101
9. References 102
10. Appendices 107
10.1 Glossary of Terms 107
10.2 CPG Code 109
10.2.1 bscListener.java 109
10.2.2 cpg.java 110
10.2.3 cpgexchanger.java 125
10.2.4 cpggui.java 129
10.2.5 cpgMessage.java 134
10.2.6 datamatrix.java 136
10.2.7 ip.java 141
10.2.8 matrix.java 144
10.2.9 matrixlist.java 147
10.2.10 row.java 150
10.2.11 socketHandler.java 154
10.2.12 surrogate.java 157

List of Tables

Table 5.1) Connected to Root state transitions. 60
Table 5.2) Creating Connection with Peer state transitions. 61
Table 5.3) Peer Creating Connection state transitions. 62
Table 5.4) Awaiting Response state transitions. 63
Table 5.5) Send Rows to Peer state transitions. 64
Table 5.6) Connected to Peer state transitions. 68
Table 5.7) Add Rows to eData state transitions. 69
Table 5.8) Send Rows to Peer state transitions. 70

Table of Figures

Figure 2.1) Local Cache: Requests are serviced from Hard Drive, so there is no network traffic 8
Figure 2.2) Proxy Cache: An intermediate server fills the request. Network traffic is decreased, as the request is not propogated to the source. 9
Figure 2.3) Server Cache: Cache exists near servers, network traffic not significantly decreased 10
Figure 3.1) CPGs participating in multiple protocol groups, and communicating with more than one Authoritative Server 22
Figure 3.2) View of overlapping content networks. Shaded areas are served by more than one Content Network 24
Figure 3.3) CPG Interaction Diagram, with two CDN's each of which having two surrogates. The Child CPG's communicate with the Root CPG 25
Figure 3.4) Content Network advertising geographically close IP Prefix ranges 27
Figure 3.5) Content Networks overlapping, and competing for IP range 137.82.x.x, their CPG's peer and exchange information about the 137.82 IP range 27
Figure 4.1) The view of the Metric Matrix to the application, eData and aData are unpopulated 32
Figure 4.2) Illustration of Matrix Calculation 35
Figure 4.3) CPG Row Format 39
Figure 4.4) A gateway intercepts a request for an IP which it is aware of and so performs appropriate redirection immediately 41
Figure 4.5) An intercepting gateway is unaware of customers IP address, thus redirection decision is made at authoritative server 42
Figure 4.6) CPGP Message format 51
Figure 4.7) State transition diagram for CPGP 53
Figure 4.8) Message sequence chart for standard connection to root 54
Figure 4.9) Message sequence chart for CPG's setting up a peering relationship 55

Figure 5.1) State diagram for states until a peering connection is made 59
Figure 5.2) Simplified state diagram for CPGP until connection to peer is made............. 65
Figure 5.3) State diagram for CPGP in connected to peer idle state................................ 67
Figure 7.1) The CPGP Test Suite IP Address Input dialog.. 82
Figure 7.2) The view of a CPG with Internal and Processing Data. There are no rows being contended thus no other CPGs have delivered external data 84
Figure 7.3) An illustration of connected content networks, where Yahoo and EA networks have peered via their CPGs, users can get content from any content server. 85
Figure 7.4) A view of a surrogate, with a row input dialog in the CPGP Test Suite.......... 87
Figure 7.5) CPG-B Conflicts with CPG-A for IP Range 92.75, thus it has populated its aData with its own information, and its eData with the row received from CPG-A . 88
Figure 7.6) CPG-B Conflicts with CPG-A for IP Range 92.75, thus it has populated its aData with its own information, and its eData with the row received from CPG-A . 89
Figure 7.7) CPGP Protocol Service architecture.. 92
Figure 7.8) Architecture of URL Rewriting Testing network 93
Figure 7.9) JSP Code used to redirect clients to different servers for embedded objects. 95
Figure 7.10) Redirection page HTML, the response to Figure 7.9 where the BSC algorithm has returned 192.168.0.5 95
Figure 7.11) The rendered HTML from Figure 7.10 for clients 1 and 2 from Figure 7.8.. 96
Figure 7.12) The rendered HTML generated from the Java Server Page outlined in Figure 7.9 for clients 3 and 4 from Figure 7.8.. 97

Acknowledgement

First, I would like to thank Dr. Ali Shoja for his supervision and motivation in this project. Secondly, many organizations have contributed to the completion of this thesis. The New Media Innovation Centre, and Nortel Networks Inc. have provided financial support. Much of this work was carried out in the New Media Innovation Centre Lab in Vancouver, B.C.

1. Introduction

Computer networks have changed the way in which the world does business, how people entertain themselves, and how they communicate with each other. Since ARPANET's inception in 1969 and the introduction of packet switching in 1970, networks have been limited in how information is delivered. As more networks were developed, most of them used the same packet switching techniques developed for ARPANET. These networks were then connected using packet switching gateways. Each subsequent revision of the Internet has maintained a degree of backward compatibility, thus continuing with not only the capabilities of the original network, but the inherent limitations as well.

As more and more hosts and customers use the Internet, many bottlenecks have appeared. Single servers can no longer handle load requirements. Service providers' bandwidth is often heavily loaded. Finally, the connection boxes between networks, known as gateways, can become congested and overloaded by inter-network communication.

The bottlenecks present with the current architecture are beginning to cause a problem in the delivery of data to end-users. The task of researchers is now to develop ways of circumventing these choke points and enabling the delivery of larger blocks of content, in more reliable and timely ways. This evolutionary step in the Internet will allow the more reliable delivery of large data streams, such as multimedia content. The technologies will also allow for reliable hosting of web sites by providing de facto redundancy. This will help ensure that denial of service attacks are less effective.

1.1.1 Motivation

As the Internet has grown, so has the vision of how it should be used. From traditional usage of reliable data transfer, such as e-mail and ftp, to streaming feature length, high quality videos, is a large step. The initial design for the packet switching technology incorporated into the Internet was meant to provide a large degree of reliability for data

integrity across unreliable communication media. The current popularity of the Internet is a direct result of this functionality. Users still expect flawless delivery of their information, yet speed is now becoming more relevant, as well as larger amounts of data and timing guarantees. Until these criteria can be met, customers and developers will have a difficult time delivering new styles of content and meeting the demands of current high traffic web services.

It is believed that many users would like to be able to quickly access new services on the Web[41]; movies, music and other multimedia services. As the Internet evolves users may see older technologies begin to be replaced. Even if appliances remain relatively unchanged, the method of delivering the data to the appliance may change a great deal. The design of a TV will remain relatively unchanged, as it's familiar and useful for watching video, but the information may now be streamed digitally. The telephone is an efficient design for people to talk to one another, but now anyone can use digital technology over wireless networks almost anywhere in the world. Even music delivery is quickly changing; CD players replaced radio and tape sets, and are now being replaced by MP3 players as the popular choice for portable music. The common factor in most of these technologies is that they accept data digitally. These appliances require fast and reliable access to the digital data in a worldwide context. Quality has also been a factor in the deployment of these technologies, and quality will continue to be needed in new services being offered.

1.1.2 Problem Definition

The key issue is that the Internet was not developed to address the problems of slow download or slow response times. The point-to-point nature of the Internet is contrary to supporting guaranteed Quality of Service, QoS, because it does not guarantee latency and throughput, desirable properties for QoS, between source and destination on a global scale. The Internet was initially developed to reliably deliver text and then images. These media require data reliability rather than timeliness. The popularity of the Internet led it to

being used for multimedia streams, audio and video. This led to many problems, server overload, and network congestion being key to a degraded quality of service. These problems were then addressed with levels of caching; local caching, then network and distributed caching; which led to content delivery networks which utilize request redirection to forward user's requests to existing caches. As the number of users, as well as the resource consumption of each user via high-speed connections, increases [40], the technology of data delivery must also advance to meet their needs.

This thesis focuses on the problem of efficient and fast delivery of content over areas of the Internet. The Internet is composed of a collection of Autonomous Systems, therefore the internal workings can follow any protocol that the owner of the network chooses. Few dedicated networks use ATM, a transport protocol which provides guarantees that would make some multimedia easier to deliver; the majority of networks use TCP/IP, which provides guarantees of delivery, but not of timeliness. This has led to a great deal of heterogeneity in the internal workings of these networks. Different entities operate these networks, with different business directives. Some networks exist to serve customer connection, some to host data for content creators; all networks of the Internet exist to move data from creator to customer.

The interaction of networks is a complex problem; the arena includes technical issues, marketing issues and issues of business interaction. Any effort to provide interoperability should include representatives from each of these groups, as well as recognition among them of each other's interest.

1.1.3 Scope and Focus

A Content Distribution Network is an Autonomous Aystem that may host duplication of data at more than one location within a given network. This duplication has the effect of allowing quicker delivery of data. By keeping the majority of data at the edges of the network, closest to those likely to view the data, bandwidth consumption can be decreased. By connecting these networks, and distributing data among them, not only can

bandwidth consumption within a network be decreased, but costly inter-network communication can be reduced as well.

This thesis develops a new protocol known as CPGP, Content Peering Gateway Protocol, to provide the necessary information among these networks in order for them to interact. It also provides a BSC, or Best Server Choice, algorithm to make use of the data that the CPGP distributes. The CPGP service simply accepts an IP address as an argument and responds with the IP address given from the BSC algorithm.

This protocol ensures that Content Networks can communicate in an effective and reliable manner. It addresses both technical and business needs.

Furthermore this thesis addresses the problems of scalability by making certain assumptions regarding the information to be transmitted.

1.1.4 Methods

By basing the protocol on well-studied and implemented network protocols, CPGP provides an acceptable solution to the problem of interconnecting *Content Distribution Networks, CDN*. This is based upon existing protocols such as BGP and the OSI network protocol stack. The protocol is further tested by an implementation in Java using reusable classes to represent the various components of Content Distribution Networks. This has provided insight into its capabilities and is useful as a tool for visualization of the overall system behaviour. It also facilitates rudimentary tests of the CDN system and shows how it can be useful for different request routing techniques. The CPG protocol is validated by reviewing possible state transitions, and showing that the state invariants are always true. Finally, by constructing an actual live system that demonstrates a complete Content Request Routing System we tested the CPGP and BSC algorithm.

1.1.5 Outline

The remainder of the thesis is as follows: Chapter 2 provides a background of the issues surrounding content delivery; including caching techniques, server farms and the advent

of content delivery networks. Chapter 3 gives an overview of the CPG work. Chapter 4 describes the specifications of the protocol; Chapter 5 discusses the validation technique used. Chapter 6 discusses the Best Server Choice algorithm, and Chapter 7 is a description of the implementation and validation work done. Chapter 8 concludes the thesis by outlining the major contributions, and a discussion of further work to be done in this area.

2. Previous Work

There have been many attempts to improve the current delivery mechanisms of the Internet. These have ranged from developing and deploying new protocols and hardware, as in ATM switching, to creating new protocols for delivering data and multimedia like RSVP, HTTP/HTML [31, 36, 39] and others. Other improvements have focused on where the data is kept or retrieved from, as in various levels of caching. Reviewing the historical settings of these improvements will aid in avoiding rehashing old ideas. It is not only important to avoid re-inventing the wheel, but equally important to avoid re-inventing less successful designs. Since the 60's these improvements have brought many benefits to the Internet. Yet the limitations of basic packet switching and server architecture remain key performance bottlenecks for content delivery. Each of the improvements focuses on some alleviating a performance bottleneck in the delivery of data across the Internet. The success of these improvements is generally measured in one of the following ways: [42]

1. Minimize the number of bytes that travel over the Internet
2. Minimize the number of hits to specific servers by distributing over many servers
3. Minimize the time that end users wait for a document to load

The methods usually focus on one of these goals, and have met with varying degrees of success.

2.1 Existing Delivery Systems

The Internet itself is the most important delivery system in use today. Its structure is relatively loose, relying on the interconnections of multiple, proprietary networks in order to route requests for data from content source to client. This method has worked thus far because of the low expectations placed upon the network. Almost everyone working with computers has heard a phrase such as "The Internet is down" or "E-Mail is broken." These common complaints are founded on the belief that if users cannot get the

information they wish, then the problem is "out there" somewhere, and that means that the whole thing is broken. Fortunately for most of the population the Internet as an entity has never been completely inoperable, only certain components of it.

The Internet uses a basic packet switching technique, each router deciding at the time of receiving a packet how best to route the packet based upon its destination. This initially served the Internet well, as there was only limited traffic and limited numbers of hosts.

As the number of hosts increased, so did the traffic. Quickly the business potential of the Internet became apparent, and consumers and businesses started to use it to the bounds of its capacity. More networks came into existence, being administered by large businesses, network providers and Internet Service Providers (ISP's). Many of these networks were constructed by existing telecommunications corporations, others were the products of start-ups attempting to break into a seemingly lucrative market. Various networks ran various routing methods, and protocols, within their administrative control, relying on standards for interconnection. Various networks began to use techniques that allowed more control over routing; ATM, or Asynchronous Transfer Mode [21], became a popular choice during the 1990's and was expected to solve many of the issues that faced IP networks, including quality of service (QoS), and resource reservation. This protocol still runs over many large networks, but failed to make inroads into home PC markets, forcing ISPs to support ATM on their infrastructure and TCP to the clients.

All of this is simply in an effort to create better service for customers which indirectly should translate into more revenue for these services. Towards these ends there have been a number of other delivery methods implemented to improve the user's experience.

2.1.1 Browser Caching

Browser Caching was among the first methods of improving content distribution response time [36, 37]. Caching displays hit rates of over 80%. For an illustration see Figure 2.1. It was intended to reduce the number of network round-trips and improve performance. The

basic concept is the same as in other areas such as instruction and data caching in CPU's, it became such a commonly accepted performance enhancement that it has been included in the latest standard for http [36]. The fundamental assumption is that when users are interested in a piece of content, then it is likely that they will revisit that piece of content within a given time span. It also relies on the assumption that most of the data is static, or if dynamic it has relatively low frequency of update.

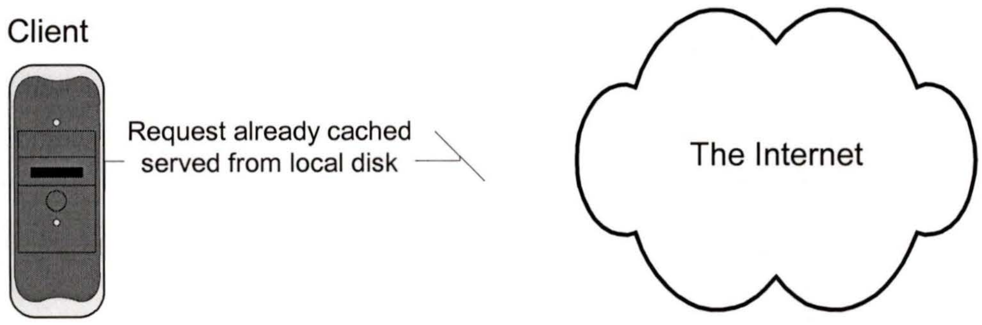


Figure 2.1) Local Cache: Requests are serviced from Hard Drive, so there is no network traffic

These assumptions have proven to be useful for speeding the response time for users demands and can be extended to provide for caching at other locations.

2.1.2 Proxy Caching

A second approach to caching occurs when an intermediate cache, called a proxy cache, is placed between the client and server. These proxy caches, see Figure 2.2, [15,18,19,28,32,40,41] are located near network bottlenecks or gateways where messages can easily be intercepted. The proxies intercept and serve http requests, and other kinds of requests, that are sources for network traffic. They are particularly useful for reducing network usage on high-volume networks. They become more useful as the number of users of these proxies increases. The further assumption is that many people will be viewing certain pages, which again change infrequently. These popular pages are stored in

the proxy cache so that when a request is intercepted it can serve the request from the local storage, rather than sending the request out of the network.

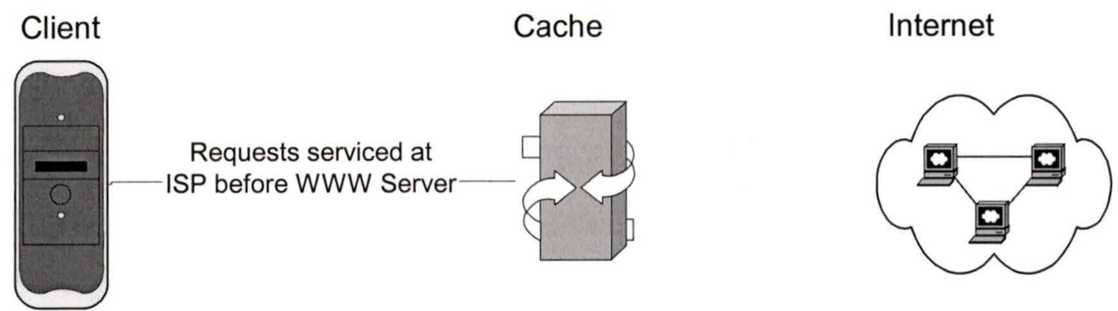


Figure 2.2) Proxy Cache: An intermediate server fills the request. Network traffic is decreased, as the request is not propogated to the source.

2.1.3 Server Caching

Server caching represents the final place where caches exist. These caches are placed immediately in front of the server [18,20,32]. Typically they will store the most popular pages served by the server in more quickly accessed local memory, see Figure 2.3. These caches do nothing to reduce network traffic but do effect quicker response for the users. These caches are also referred to as reverse caching, as they are often predictions of what will be viewed, and are simply means of decreasing the number of requests handled by the server. Server caching can be viewed as having two servers that cooperate to host some data; one of the servers is dedicated to a much smaller percentage of high-volume traffic, and the other a much larger percentage of less commonly accessed information.

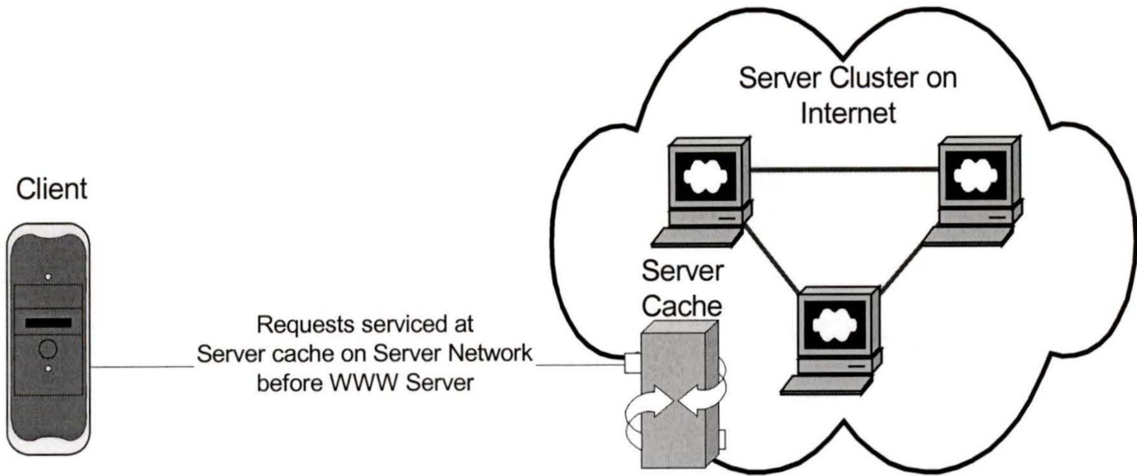


Figure 2.3) Server Cache: Cache exists near servers, network traffic not significantly decreased

This method can be one of the most effective, and most easily controlled by the content host, as the server owners administer all of the components of server caching.

2.1.4 Server Farms

There is a limit to the number of requests that a single server can handle, and regardless of the number of caches between a client requesting data and the server that responds to the request, these limits are often reached. Once this happens other solutions must be found. In some cases a better server is all that is required to handle more requests, but eventually the requests may outstrip even the most powerful of available servers. This leads to the idea of multiple servers hosting the same content. These collections of servers are called *Server Farms* which host data on different machines, all coordinated by a gateway to the server farm [20, 43]. These collections have many benefits. For instance a server farm may have greater degrees of fault tolerance, as there are many different servers. If one computer crashes there are often more to take its place. They also can go through complex software updates while still being active. This may be viewed as an extension to basic server caching which is in turn an extension of basic caching. The logic

is controlled at an administrative point that is managed by the content providers. This point has the ability to route requests not only by the requested information, as is the case in all previous caching techniques, but also by server load. The added metric of server load allows for more scalability, as more servers can be added as needed.

These benefits are not without cost. There is a large amount of administration, and the cost of maintaining so many servers is high; however charging for the large volume of data being transmitted recovers these costs. The problem then becomes not handling the server loads for the customer requests, but managing the network congestion at the points where these requests must pass through some router [20, 43].

2.2 Content Distribution Networks, CDNs

Just as network traffic has grown, so has processing speed of network switches and servers. One of the issues that the original network protocol designers had to contend with was the amount of processing cycles available to compute routing [37]. The datagram, or connectionless nature of Internet routing requires each packet to utilize some portion of network routing computation. It follows that as the number of packets rises, so does the amount of processing required to route those packets. Until recently the processing barrier has limited the amount of decision-making that can reside inside the routing switches. This has pushed the decision making for accessing the content to the edges of the network, the servers and clients that actually work with the data.

Intelligent networks [18] perform decision making beyond simply routing packets. This involves more processing within the network routers. The advent of intelligent networks [18] is therefore recent. The amount of processing that each packet or connection [20, 43] can utilize has increased. This means that caches can be placed in more locations, not solely in intermediary bottlenecks. By detecting requests a network that has enough intelligence to know where the data is stored, if it is stored in multiple locations, may choose to route the request to any one of these locations. This technique is called Content

Networking [1-4]. It relies on the ability of a switch or router to recognize requests and summarily redirect these requests to other servers. This can also be performed at the server side of a network given that there are redirection mechanisms in place. The HTTP protocol that is responsible for most web traffic today has methods to redirect these requests.

Regardless of where the redirection decision is made there are companies that build businesses out of these services. These redirection services can save thousands of dollars in network traffic per server. BC Net, a network services provider that provides services to British Columbia Universities, estimates that they save more than $16,000 each month by hosting some content servers on their network [50]. BC Net merely provides space for the servers, and Akamai, the company that owns the servers, provides all the administration and servicing. Akamai then sells the hosting service to content providers.

As more of these companies evolve they may enter partnerships to provide better service to their customers. The Content Peering Gateway Protocol, or CPGP, in this thesis is designed to support this interaction. This protocol operates as a service to redirection mechanisms, which eliminates the need to be aware of the type of redirection being performed.

Akamai and other companies have solved the problem of request redirection in a single CDN, where all the information is available to the request routing decision. We consider the case of multiple CDNs which are administered by competitors. These CDNs wish to share the minimum amount of information in order to achieve request routing.

2.3 Request Redirection

CPGP provides the service of selecting the best server to redirect a client's request to. This service is only useful in the presence of some form of redirection mechanism. Following is a review of the common known redirection methods. These mechanisms vary in the nature of the networks on which they reside, in the level at which they operate,

and in the goals of the redirection. The Content Networks which support the functionality to redirect requests and responses, often make use of one of the more higher levels of a network stack; application or transport levels.

All of these content redirection techniques do so by making use of some existing service, and providing information on where to redirect the request, these exist in some transport protocols, e.g. HTTP or RTP [48], or in some other mechanism that modifies where the user tries to receive the content from, e.g. DNS.

These methods are classified into the three main existing methodologies, DNS redirection, Application Layer redirection and Transport Layer redirection, each having a different set of metrics and objectives.

The following sections are summaries and highlights from *Known CN Request-Routing Mechanisms*, an IETF Draft from the CDI Working Group [3]. The requirements set out by this group are the basis for the work presented in this thesis.

2.3.1 DNS

The DNS [2-4, 17, 44] service is the primary method by which computers translate a *URL*, Uniform Resource Locator, to an IP Address [45]. Redirection occurs when a specialized DNS server is inserted into the resolution process. This server will make use of the Content Peering Gateway Protocol to resolve a URL, based on defined policies. This technique uses the IP address of the client's DNS server as an indicator of the client's location. The redirection can take one of many approaches.

Single Reply

In this case the DNS system is authoritative, and makes the decision locally. It fills in an A record, a DNS response that is authoritative, with the IP address, or virtual IP address of the best server hosting a copy of the requested content. This server is also known as a *surrogate*.

Multiple Replies

Here, the routing DNS returns multiple A records to the client DNS. This has the effect of creating a round-robin queue on the client side, which will be used for a series of client requests. Again, the DNS is authoritative

Multi-Level

In this case the DNS server does not make the decision locally. Instead it uses one of the DNS query mechanisms to refer to another, more authoritative server. This may occur to redirect within a Content Network or between Content Networks. It may be applied when a DNS with national scope refers to a DNS with territory or local scope for more fine-grained resolution. It utilizes the NS or CNAME DNS records, each of which corresponds to a redirection mechanism.

NS redirection involves resolving various levels of a domain name incrementally. A request for x.y.z.ca will first be queried at the z.ca level, which may refer the request to the DNS authoritative for y.z.ca and so on. This method can be effective and allows for a distributed approach to DNS redirection; however it has some drawbacks. The number of parts of the domain name being resolved limits the number of redirection iterations, and the final DNS used in the resolution determines the time to live of the entire process.

CNAME records are used to redirect resolution to an entirely different domain. This can be quite powerful as it may be used for even minimal length URL's with more than one redirection. The drawback to this is that it therefore involves the resolution of even more domains.

Other DNS redirection mechanisms include Anycast and Object encoding.

Anycast utilizes a broadcast mechanism or Anycast server to discover a reasonably local server that provides a service. This is applied in situations where a service is being requested, and the client is ambivalent to as the source of the service.

Object encoding is a technique of adding information to the DNS name by some schema. The domain name then contains an identifier, or clue, as to the content. This can be used in conjunction with one of the other mechanisms to provide for more comprehensive solutions.

Limitations of DNS based redirection

The DNS based routing techniques have some inherent limitations as follows:

1. Resolution occurs only at the domain level, not per object
2. Servers may be required to return entries within short TTL (time to live) values, increasing the volume of requests, as a DNS request may be resolved with every transaction, rather than resolving only once then cached.
3. Reliant on DNS implementations, which do not always follow the standard.
4. Routing is based on the client's DNS server, which isn't necessarily an indicator of the client's proximity.
5. Recursive DNS resolution further hides the originator of the request, making intelligent decisions hard or contingent on additional data.
6. Users from a specific client site DNS will be redirected to the same surrogate within the TTL for the first request which may flood servers.
7. DNS timeouts may occur while handling particularly complex resolutions.

2.3.2 Transport Layer Request Routing

Transport layer routing can support finer levels of redirection. It may inspect the information in setup packets to discover more information with which to make the selection. The inspection may provide data about the client's IP address, port, and protocol. Acquired data could be used to more reliably choose the best server. Transport Layer Request Routing is utilized by proprietary network routers that have the ability to intercept and modify TCP session requests on behalf of a user.

2.3.3 Application Layer Request Routing

Application layer redirection has the benefit of access to the most amount of information on which to base any redirection decision. The application layer protocol has information

not only of the client's IP address and protocol, but exposes data about the content being requested. This provides the most comprehensive control for determining what surrogate should be used for servicing this request. The mechanisms by which Application Layer Request Routing varies depending on the protocol being used by the client. Following is an outline of the most common variants of Application Layer Routing.

Header Inspection

This type of inspection is based upon information passed in the setup of a communication session. The information passed about the content is often found in the MIME request header, or complete URL.

The Uniform Resource Locator, or URL is a naming scheme that qualifies the object being requested. Elements of a URL usually include information on the transport method being used, where the information is located on the network and where the information is located within a server, as well as often including what sort of information is being requested. In most cases this information is sufficient to uniquely identify the content, and choose the server to route to based on this data.

Although URL is one type of header redirection, it is not the only. Many protocols now have some built-in method of redirection. HTTP uses the code 302 to redirect a user to another server by changing the requested URL. These methods are extremely easy to implement, but have the drawback of added latency.

The In-Path redirection method is another header-based redirection. This method is most easily understood with respect to a network cache. An intermediate element, which exists in the network, performs some logic based on the client's content request. This intermediate node does not have the information on hand, as in the case of a cache, but decides from where to draw the information.

Mime headers often encode extra information in the form of cookies, data collected by a browser during the current, or a previous session. These headers may have additional

information not found in the other redirection mechanisms such as language being used, and session information. This added information allows for more flexibility in supporting persistent sessions even through multiple redirections.

Content Modification

Another method of content redirection occurs after a session is already negotiated. This may then involve redirection portions, or new sessions to other servers. This redirection is thus done on object level, and several objects could be streamed from different servers in order to balance load. This method is best applied when object requests are embedded within another content request, as in objects in an HTML document. Re-writing object encoding within an HTML document is known as URL-Rewriting, and can be done before the HTML is requested, or while it is being served. It involves changing the references to objects in HTML code in order to refer to different URL's to retrieve the object from.

A-Priori URL rewriting does not take into account information about the user request. It involves changing the reference to the location of objects before they are requested, and thus redirecting the requests for those objects to new locations. This can be used in conjunction with a DNS method in order to redirect customers to one of the copies of a replicated HTML document that then has references to other locations for the embedded objects.

On the other hand, on demand URL rewriting takes place when the client request reaches the point at which the original document is being served. In the case of an HTML page being sent to a client that contains references to objects, those objects can be rewritten while serving the HTML document to direct the client to other surrogates. This method makes effective use of information about the client, and the request.

Although content modification is an effective request routing technique, there are limitations to its use. The most severe limitation of Content Modification is that the

original request must be served from the server that the user originally requested information from, also known as the *origin server*. This allows the origin server to redirect much of the work such as serving embedded objects, to other servers but does not reduce the number of sessions, only the complexity of each session.

Another de facto limitation is that the objects being served from surrogates should have very small or no cacheable periods; during which other caches, such as browser caches are allowed to hold the data. This is because if a user is to view the information a short time later, they may be required to communicate with another surrogate, that surrogate may have different, and newer data. If the user has cached that object they will still be viewing the old content.

2.3.4 Combinations of Mechanisms

The most powerful redirection systems will make use of combinations of different mechanisms. A DNS redirection could direct a client to a server that performs On-Demand URL rewriting. The objects themselves may then be a part of some content network that performs any one of the redirection mechanisms to find the best server for that object. The danger then lies in performing so much redirection that the overall response time is slower than if no redirection were done at all, or other methods such as caching are utilized.

2.4 Content Distribution Internetworking

Each of these methods relies on administrative control over an entire network. There is no currently defined mechanism for exchanging information describing how these redirections should take place. The proposed CPGP presented in this thesis takes on the role of exchanging information respecting which server should be chosen, but is only part of a larger system which must include distribution mechanisms and accounting mechanisms. By connecting many content networks these systems can then make use of a

much more global scope. Companies can then focus more on their national or local customer base while still meeting levels of committed global service.

3. Content Peering Gateway Protocol, CPGP

Content Peering Gateways, CPGs, are the communicating elements of CDNs. A CPG is the CDN's interface to the outside world, and hides the internal structure of the CDN. Each CPG has complete knowledge of the internal workings of its CDN. It is responsible for the Content Network and its relationships with other CDNs.

3.1 Primary Function

The *CPGP* is primarily meant to provide enough information to each Content Peering Gateway, so that with high probability it can make good routing decisions locally, thereby decreasing traffic, and sending other decision making requests to the *Authoritative Server*. The Authoritative Server is an element of a group of CPG's that maintains a global view of all the CPG's. This can only be accomplished by hosting enough information so that most requests intercepted by a Content Network can be resolved without leaving the network.

First we assume that there is a default delivery server. This server is used for any customer who is not explicitly served by a CDN. The Authoritative Server, which is the coordinator of the content peering system, can always refer to this server in the event that no other surrogate exists.

The next assumption used in CPGP is related to locality of the customer. If a Content Network that can provide the content locally intercepts a large number of requests from a given IP range, for instance from the UVic Class C IP block, then it will *advertise* service provision for that IP range, this means it will inform the Authoritative Server that it wishes to serve content to the IP Addresses within that IP range. This reflects a principle of locality. If a network intercepts requests from a user, then the network is between the user and the data. Therefore if the intercepting router can choose a surrogate on its own network then it will be closer to the user.

This allows us to have a dynamic and automatic way of updating which IP ranges a CPG advertise. By identifying and logging at interception points which IP ranges are being intercepted, the CPG can set metrics on when to begin servicing those IP ranges. For instance, if a University network gateway intercepts many requests for a specific content then if it has that content locally it may decide to advertise provision of that content to the IP ranges for which it is the gateway. The IP range being served is also known as the *footprint*. If it then decides to service a footprint it would join the content peering group for that content and thus be able to make decisions based on a view of every node which serve that range.

3.2 Protocol Scope

CPGP is meant to operate within the context of a single Content Unit. Take the example of streaming media; the CPGP servers would participate in exchanges for each movie, or group of movies. In the example of DNS entries, there could be a separate CPGP group for each domain name, or group of domain names. This means that CPGP does not identify the content unit within its framework, as this is a function of the service using CPGP. This information could be held at the Authoritative Server if an implementer so wished and would be useful from a management and information point of view.

Thus if a Content Provider such as Disney wishes to stream many movies, e.g. Mulan, Aladdin etc., and host these movies on separate servers with some servers hosting both movies, as illustrated in Figure 3.1, then it must host an authoritative system for each movie. Also, each surrogate must participate in a separate instance of the CPGP structure for each movie for which it streams, as illustrated in Figure 3.1.

This allows the protocol to be applied to any redirection mechanism. It acts as a service irrespective of the redirection mechanism. A DNS server would use the CPGP API to discover the IP Address to put into a CNAME or A record. An HTML rewriter could simply use the IP returned as a reference for an object, still operating in real-time.

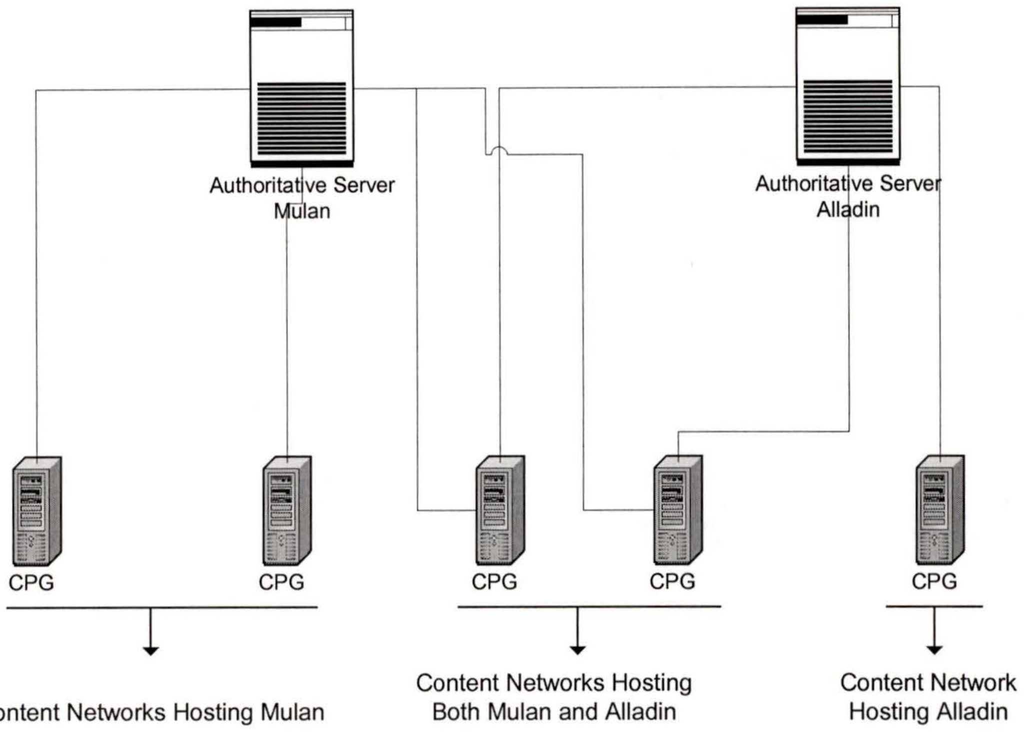


Figure 3.1) CPGs representing one or more content units, and communicating with more than one Authoritative Server

This model maintains a distinct data structure and communication topology for each content unit in order to achieve a high level of efficiency. If the content being hosted is extremely popular, then the surrogate is unlikely to be participating in many relationships at once, thus decreasing the additional overhead of managing the protocol. If, on the other hand, the content is not highly solicited, then the protocol will not be overly utilized, thus ensuring that the protocol management does not become a performance concern.

Business agreements are likely to follow suit. There is little justification to negotiating individual contracts for small units of content. As such CPGPs will likely reflect the agreements between companies.

3.3 Root CPG

In order to be useful, CPGP must be scalable to work on connecting networks across the Internet. This means that no centralized service will be able to effectively handle the volume of requests. A means to distribute the workload, including the decision-making, must be incorporated. Furthermore it must work to minimize the time taken to make decisions while still ensuring that those decisions are informed and accurate. Content Peering Gateway Protocol accounts for these requirements. The protocol distributes all relevant information to its interested participants, so that they have a complete view of all content networks with which they compete to service customers. Thus each network has the capability to route requests that it may service but is not overwhelmed with information in which it has no interest.

At the heart of this system is the Authoritative Server, which we also define as being the Root CPG. The Root CPG is responsible for identifying the footprints which are being advertised by more than one CPG. These overlapping areas are the points of contention that each of the CPG's involved must have a complete view of. The Authoritative Server does this by maintaining a complete view of the Content Peering Group coverage.

The Authoritative Server must also be aware of a default server that can also serve the content if no other choice exists.

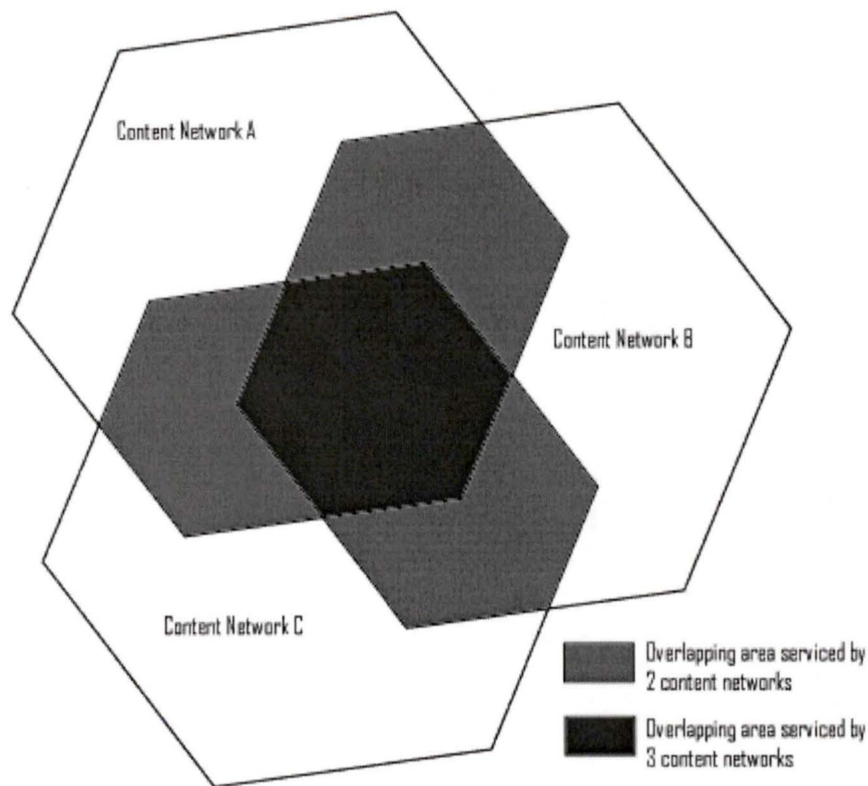


Figure 3.2) View of overlapping content networks. Shaded areas are served by more than one Content Network

By interaction with the Authoritative Server CPG, each other CPG makes intelligent routing decision based upon a complete view of any user for which it advertises service. In Figure 3.2 we can see where each content network overlaps with each other. The shaded areas represent service coverage, where the darker areas are served by more than one content network These views are managed by the Authoritative Server, and then communication between peers is left to the CPG's involved.

The functionality of the root CPG is limited to coordinating the other CPG's and managing the default destination. It is not allowed to host its own surrogates.

3.4 Child CPGs

The other type of participant in Content Peering Gateway Protocol is the representative CPG of a Content Network. This sort of CPG coordinates communication with the CDN's surrogates and its peers. The representative CPGs are called child CPG's or children.

For the purpose of the protocol, each CDN is represented by exactly one CPG, thus each CPG appears as a separate Content Distribution Network. It disseminates any knowledge that it is willing to reveal of its internal structure.

The connection between the representative CPGs and the root is initiated by the child's gateway. Refer to Figure 3.3 for a diagram of the architecture of various CPGs interacting.

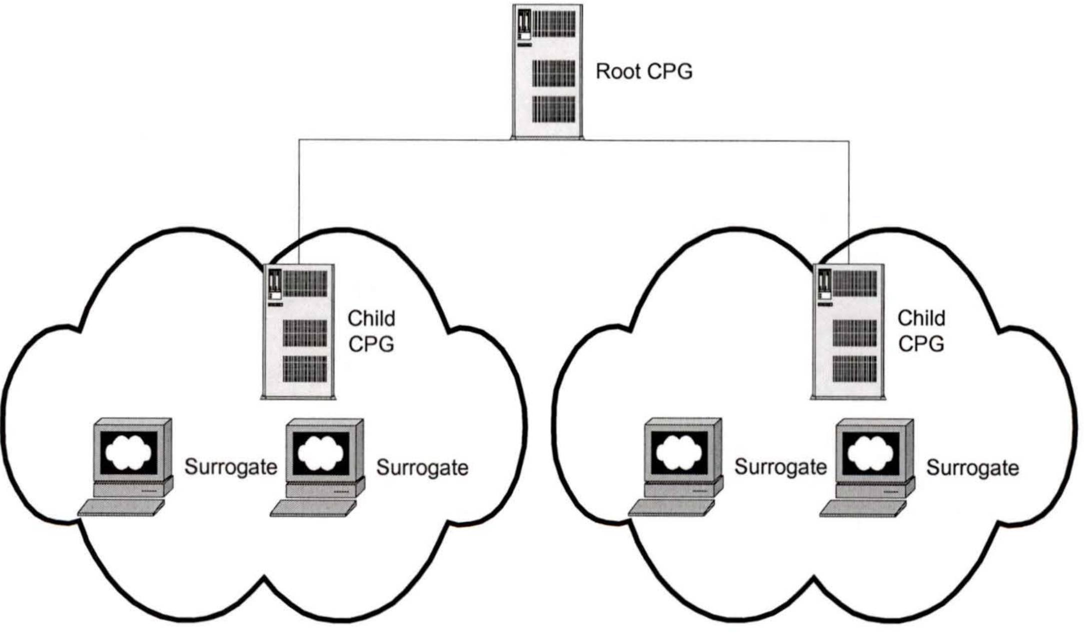


Figure 3.3) CPG Interaction Diagram, with two CDN's each of which having two surrogates. The Child CPG's communicate with the Root CPG

3.5 Peering CPGs

The most interesting aspect of the CPGs occurs when they contend for an advertised IP prefix range. The root must identify when this occurs, communicate the intersection to the child CPG which then will either initiate, or utilize an existing communication link to the other interested CPG or CPGs in order to maintain a complete view of its interested areas.

Logically, IP ranges can be construed as geographic locations. The model for Internet Address Allocation, as described by the Internet Assigned Numbers Authority states that blocks allocated to ISPs should remain intact [49]. ISPs as business models will therefore have geographic significance and although there is no specified mapping of Internet Address to ISP, this geography or topology of gateways and connections points could be described spatially. This does not imply that with no further knowledge an IP address can be translated into a geographic location; however it gives us hope that a given location will likely have blocks of IP addresses. For instance, a company or university may purchase a class of IP address block, and manage that locally. A Content Network wishing to advertise that range is then likely to advertise IP ranges which are both geographically and network topologically close, see Figure 3.4.

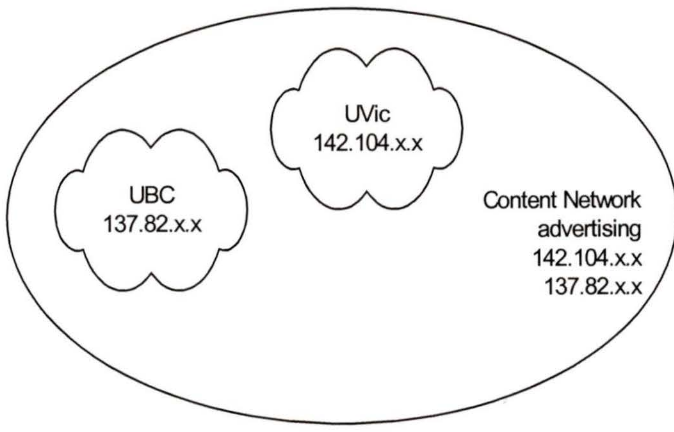


Figure 3.4) Content Network advertising geographically close IP Prefix ranges

Thus it is possible to construct a map of geographically close IP networks and overlay the Content Networks that advertise them, see Figure 3. 5. These may be visualized as maps.

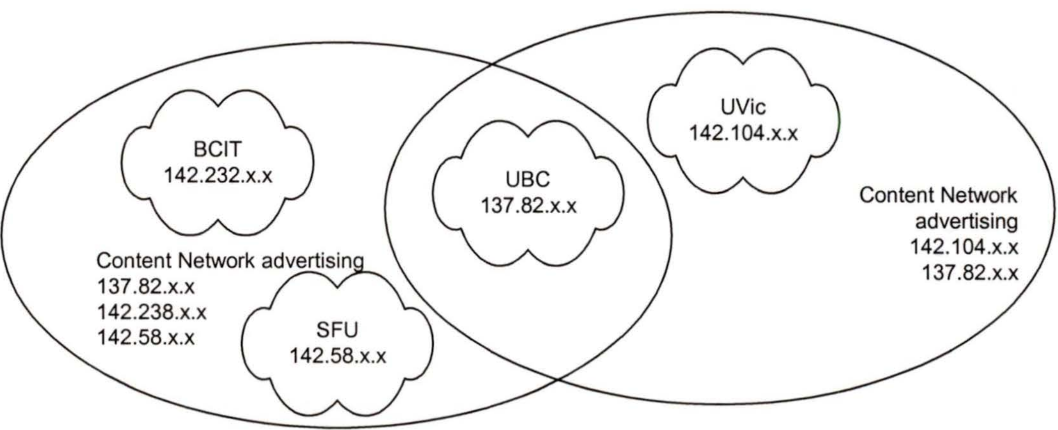


Figure 3.5) Content Networks overlapping, and competing for IP range 137.82.x.x, their CPG's peer and exchange information about the 137.82 IP range

When this situation occurs, in order for either Content Network to make an intelligent decision for routing any request it intercepts, it must know the state of the other Content Networks with respect to their overlapping IP range or ranges. This is what the peering relation represents.

3.6 CPGP Summary

By maintaining views based on a specific content unit in a specific locality CPGP has the ability to effectively route requests with assurance that any routing decision is accurate. These views are created and maintained automatically. The use of CPGP significantly decreases the traffic and load on the authoritative system. This will result in better global network traffic management by use of more intelligence within the network. The peering relations allow for an appropriate amount of data exchange between interested networks. The implementing of the protocol as a service for other redirection mechanisms allows for the maximum amount of flexibility while providing the necessary functionality. Identifying content units of appropriate size and business potential is therefore left to the industry implementers and their business needs.

4. Protocol Specification

In order to be effective the specifications for the CPGP protocol encompasses not only the messages to transfer information, but the state variables to be used as well. These state variables are an integral part of the system, and warrant discussion in their own right. Although the protocol itself does not rely on the implementer's details of the internal state variables it is most reasonable to assume that various implementations are similar.

The most important state variable set is the *Metric Matrix* which is the state of the advertised IP ranges, their advertisers and associated information. This thesis refers to a single group of this information as a *row* of data. These rows are uniquely identified by the addresses being advertised, and the advertised destination. This matrix stores all the information for choosing a server to serve a given IP Address. The matrix includes information about a CDN's servers, information about peered CDNs, information sent to peered CDNs, and information used to process requests.

The protocol makes use of the internal states and relies on them to discover what needs to be done at a given time. As such they have strong dependencies upon each other and must be discussed hand in hand.

4.1 Matrix Requirements

The metric matrix is the collection of internal state and row information. It is necessary to the operation of the Protocol, representing the data used in the procedure rules upon arrival of a message.

This chapter qualifies some of the choices as to how the matrix is structured and its storage format. These issues are dependant on a few factors; what the state is used for, how the matrix is arrived at, or calculated and how it is administered.

The Metric Matrix will be storing the raw data that is:

1. Proprietary – gathered within the CDN or from client surrogates
2. Exchanged by the protocol with another CPG
3. Used by the BSC algorithm

There are a number of available formats, each with its own properties. The choice of how to store the data is left open. The protocol must specify how the data is encoded for transmission, and as it is convenient the implementer may wish to store the information in the same manner. However any implementation may choose to translate to and from a proprietary format for performance reasons.

4.1.1 Format of Data Storage

One of the biggest questions regarding the Metric Matrix is what format the data will be stored in. There are several choices, each with advantages and disadvantages. The three main contenders are

1. ASCII
2. BYTE
3. XML

The first option, ASCII, has the benefit of being human-readable, is generally quick, and is easier to parse than BYTE, but more difficult than XML. ASCII storage would be roughly readable and leads to a good deal of debugging potential. Furthermore, some of the information will be the application level headers and will be compared with URL's, which are ASCII based.

The next option, BYTE storage, saves space and leads to a large degree of efficiency, both in memory and speed. This is considered important for system applications. Furthermore, every system that is developed will have the same representation for bytes.

Finally, XML is a newcomer on the storage scene. There are parsers already available for viewing XML encoded data and it may then be incorporated into general-purpose administrative tools more readily. Unfortunately, parsing XML adds a level of complexity and also performance hits.

CPGP specifies a mixture of BYTE and ASCII; this will ensure the performance of the system remains high while still providing the readability. IP's will be stored in BYTE format and URL's and other variables will be stored in ASCII. The added complexity of XML is not justified in this case. Using XML would necessitate parsing functionality into the CPGP system.

As this is a protocol specification there is no need for the parsing functionality behind XML.

4.2 Matrix Specifications

As in the example of BGP [1] there are a number of different categories of data involved in a routing system. Internal Information, Information to be shared, Information acquired from another CPG, and Information that is used for processing. As in BGP, the Matrix is divided into four distinct categories; (1) Internal Data (iData), that information a CPG wishes to keep secret or hidden within the black box; (2) External Data (eData), the information garnered by advertisements from other CPGs; (3) Advertised Data (aData), the information available for advertisement, and (4) Processing Data (pData), information used in processing requests. This information will be referenced by the algorithms to determine best route choices and will be calculated based on the collection of other data. Figure 4.1 features the view of the matrix from the live test application.

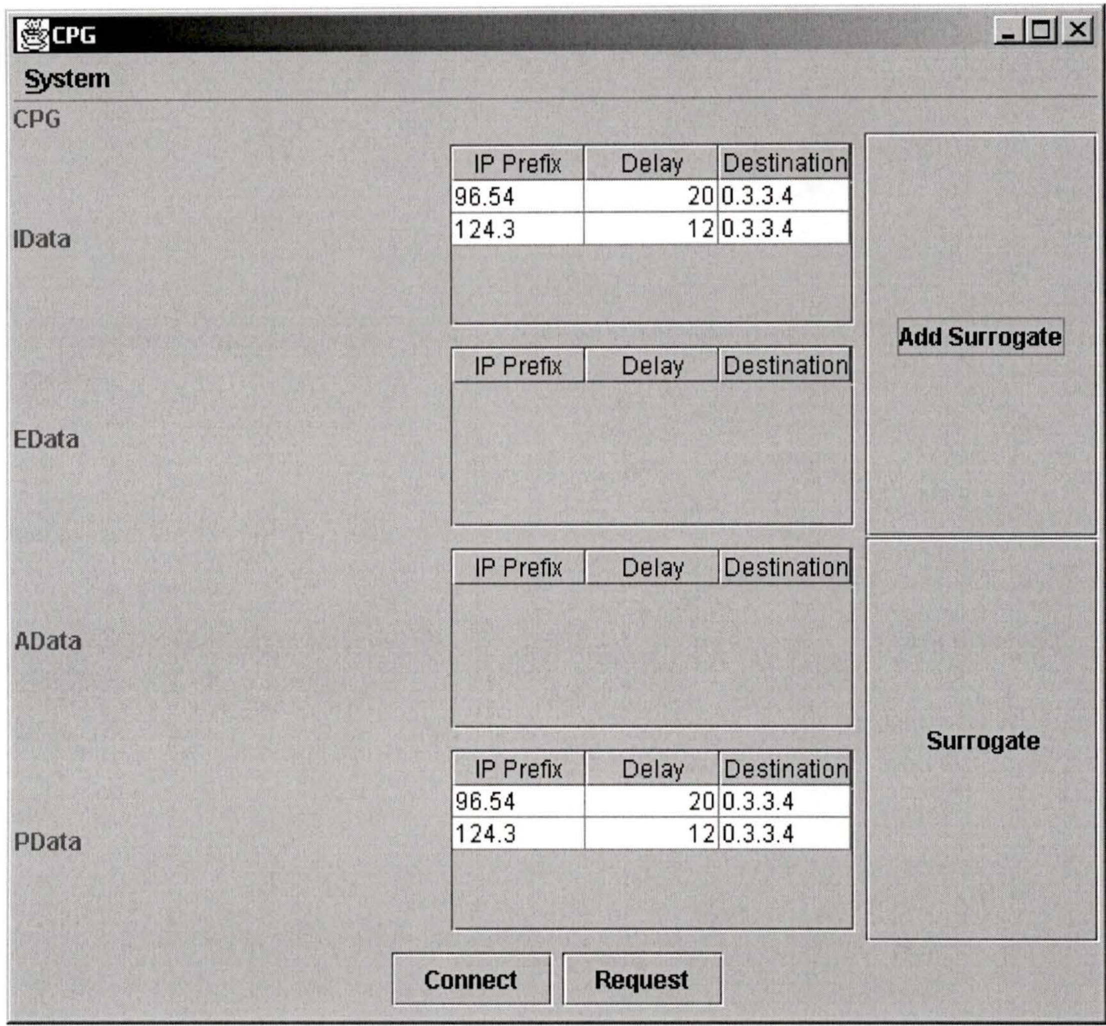


Figure 4.1) The view of the Metric Matrix to the application, eData and aData are unpopulated

4.2.1 Internal Data (idata)

Because one of the primary requirements of the RRS system is to ensure that each content network can be treated as a black box, it is logical to ensure that a CPG is aware of its own internal data, in a place where it can treat that information separately. In designing the routing system, it is then advisable to consider this in the design. By ensuring that some of the data remains internal a CPG can treat it differently. The CPG may process

internal data separately, change values for advertisement to reflect SLAs, Service Level Agreements, or even choose not to disseminate large portions of this data on whatever managerial criterion may be chosen. The Internal Data portion of the metric matrix addresses these needs.

As the iData is private information, and never directly shared or viewed by external bodies, it may be retained in whatever structure is convenient to the managing system. This may include data not normally useful to the proposed BSC algorithm, or supersets of the data that need to be processed before advertising.

The iData matrix may also be used to store rows that have been changed, but not dramatically, in order to report them all in one large bundle. This would have the effect of decreasing the number of messages being transmitted, and therefore increase the likelihood that the matrix is not in a critical section when a request from the BSC comes in.

4.2.2 External Data (eData)

External Data is the collection of all data that is gathered by the system. This is raw unprocessed, and therefore will generally not be used for the calculation of destinations. This will instead be used by the protocol to calculate the processed data fields. These entries are received from peered CPGs.

4.2.3 Advertised Data (aData)

Advertised Data is the collection of rows which will be advertised to peered CPGs. This is based on where the CPG's internal rows intersect externally collected rows. The information on which rows need to be advertised is either calculated when the CPG receives new information from a peer, or reported to the CPG from the root. The root calculates these when it receives new information from the CPG. The resulting matrix is the superset of information for which this particular CPG is responsible for advertising to its peers.

The aData must therefore also hold the information about which peer the data is for. In our implementation this is kept as a reference to a *cpgexchange* object, which is responsible for the lines of communication between peers.

4.2.4 Processing Data (pData)

Processing Data matrix contains union of internal and external data sets. It is the portion of the data of interest to the protocol for determining destinations. Ideally it would hold solely the computed best choice for any given IP range; however this would create a great deal of processing with every update. For certain types of content, small units of dynamic data, it may very well be more efficient to preprocess the processing data set, and finding the best server for each IP prefix, rather than processing each request separately.

In the trivial case this processing would simply be determined by footprint; however there may be instances where other factors, such as SLAs, come into consideration.

In a distributed environment pData is then the intersection of the CPG's internal rows with the advertised rows from all other CPGs. The rows that a CPG exchanges with a peer are solely those rows that both CPGs are interested in. This minimizes the amount of data transferred, while creating an environment for making intelligent decisions.

CPG's will focus on the requirements of the BSC algorithm in order to best define this table; the protocol will then logically follow.

4.2.5 Calculated Matrices based on Source Matrices

The Internal and External Data matrices are sources of information. The internal data is gathered by the CPG from its surrogates where the external data is gathered from other, peered, CPGs. From these two sources of information the processing data or the advertised data fields can be calculated.

The processing matrix is the combined matrices, and the advertised data is the intersection of the matrices, as illustrated in Figure 4.2.

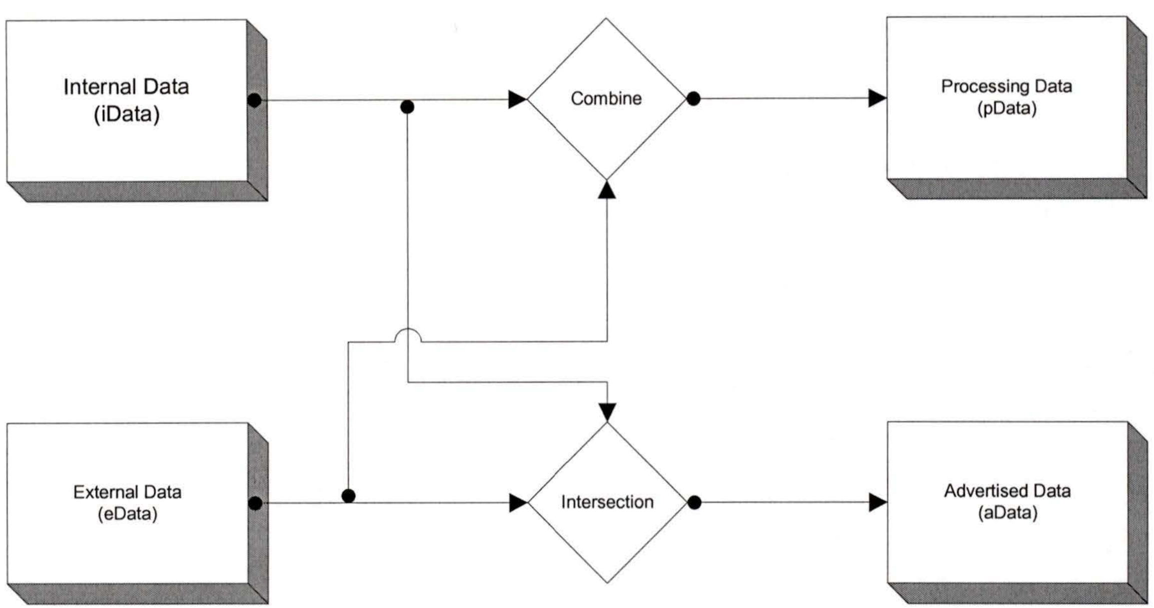


Figure 4.2) Illustration of Matrix Calculation

4.2.6 Summary of Metric Matrix Fields

The amount of data needed to support intelligent decisions is an open problem. There are a number of considerations, including the requirement that it is an extensible set. This extension may be contracted on the basis of individual content units, and may be set by the content provider when it negotiates contracts.

The following discussion is of what fields are important to the operation of the protocol. The CPGP protocol is meant to define solutions to the current bottlenecks of network traffic and serve as a solution that operates on new networks as well. IP v6 has been defined and will be the standard used in networks in the coming decade.

In order to provide backwards compatibility to IP v4, all fields representing IP addresses will necessarily have another flag representing IP version number. These fields are shown in Figure 4.3, and defined here:

Foot Print:

- The IP Ranges which may be serviced
- As these prefixes will be of varying length, there is a companion byte which determines the length of this footprint.

This will be a 16 bytes series representing an IP v6 number

TYPE – BYTE

SIZE – 16

Destination: what the destination for routing is

- An IP Address, or an alias of an IP Address (*Virtual IP Address*)
- The CPG or the actual Surrogate
- The destination will be chosen by the CDN doing the advertisement, and will be the handler of the request for that CDN. It may be the surrogate, in which case it will service the request itself, or an intelligent routing switch, or another router that chooses the surrogate appropriately, and provides the rerouting mechanism.
- Provides a mechanism to maintain the black box of an external network by allowing peered CPG's to provide an entry point to the network, rather than the IP address of the server.

This will be a 16 bytes series of data for compatibility to Ipv6

TYPE – BYTE

SIZE - 16

Data Source: What host the information is from

- Stores the IP of the remote CPG from which the local CPG got this row of data
- Useful for accounting
- Useful for updating
- This will be better implemented as a unique CPG-ID which identifies the CPG as well as its CDN. This is needed in the case of CPG's which coordinate multiple IP interfaces.

  TYPE – ASCII
  SIZE – 8 (4 and 4)

Domain ID:

- The domain ID of the destination, as assigned by an elected authority.
- Unique ID to perform accounting, and possibly to eliminate loops
- Shall be administered by a governing body or by common acceptance. And will necessarily be unique among peering groups. This will be kept as short as possibly as it will be concatenated with other identifiers to provide unique ids.

TYPE – ASCII

SIZE – 4

DTF – Delay to Footprint

- Estimated delay, this may be a calculated field

- This is necessarily just an estimation, as it is a metric on an area (footprint)
- Incorporates communication and network delays as well as server load.

TYPE – BYTE

SIZE – 4 (will be interpreted as a 4 byte integer)

SLA-Info

- Including information as to priorities and tolerance levels
- It is likely that a simple preference metric can be applied here.
- Since each table will represent a single URL space that the system is authoritative for, then this can be an internally defined metric.
- This is currently a tie-breaking mechanism, and therefore is simple
- E.g. Content Network A, with CPG-A, agrees with Content Authority that for the footprints that CPG-A advertises for, it will stream for any client that it deliver to with a delay no more than 100mSec greater than the next closest CPG, therefore this field is set to "100" which is subtracted from the DTF field during processing.
- E.g. Content Network A, with CPG-A, agrees with Content Authority that each movie streamed will represent 2Mb/sec. of bandwidth. CPG-B agrees that each movie streamed will represent 3Mb/sec. of bandwidth; so calculating server capacity is based upon this modified metric.

TYPE – BYTE

SIZE – 1

Options:

- This will be a string with length X, X being the first word in the string.
- This may vary depending on needs of the content unit, but is not used in the CPG calculation, and should be made available to the upper layer upon request
    - In the case of URL re-writing a relative URL to the destination IP.
    - For DNS redirection would be a TTL (Time to Live) for caching.

TYPE – ASCII

SIZE – calculated

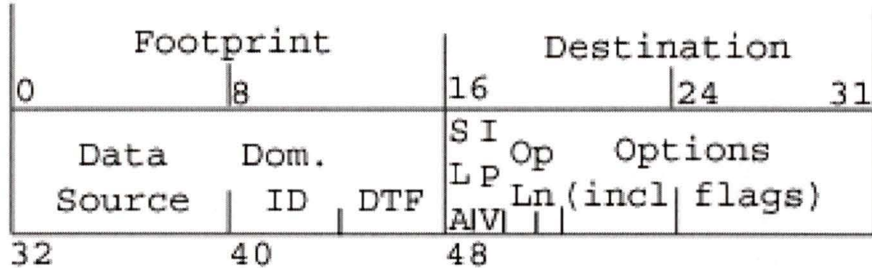


Figure 4.3) CPG Row Format

4.3 CPG Data Exchange Protocol

We need a protocol to govern data exchange. The CPG data exchange protocol, CPGP, serves to provide this functionality. The system is to be used as a service to other programs.

A DNS server may use the system to choose what IP should be used to resolve a request. If it chooses an internal surrogate it may return an A record, using the IP address returned; however if it found an external re-route address it may utilize the *NS* or *CNAME* records to partially resolve the address, to report the rerouting address as the appropriate location to find the IP address.

In order to most effectively define the protocol five components of the specification, as laid out in [5] are used.

4.3.1 Specification Components

The necessary components of a protocol definition includes: what the service does, how it manages itself, how it connects to other networked services and the way in which it communicates. This is summarized by:

a. Service to be provided
b. Assumptions about the environment
c. Vocabulary of messages
d. Format, or encoding of each message
e. Procedure rules upon reception of messages

4.3.2 Service to be provided

The CPG system will provide an API to associate a client IP address with the best server to attend to that client. The CPG system will run constantly as a service on the CPGs and respond to requests. The service simply accepts an IP address as an argument and responds with an IP address. This functionality allows greater flexibility than trying to perform the redirection as well.

If the CPG has no specific knowledge of the client's location in the range of IP addresses then it responds with a default answer. This will usually correspond to the authoritative system. At the authoritative system there may be a server designated to handle all otherwise unsolicited IP ranges, or a further redirection may take place. Figure 4.4 shows

an example of a redirection in which the intercepting gateway is aware of the client, whereas in Figure 4.5 the intercepting gateway is unaware, and thus the decision is made at the Authoritative Server.

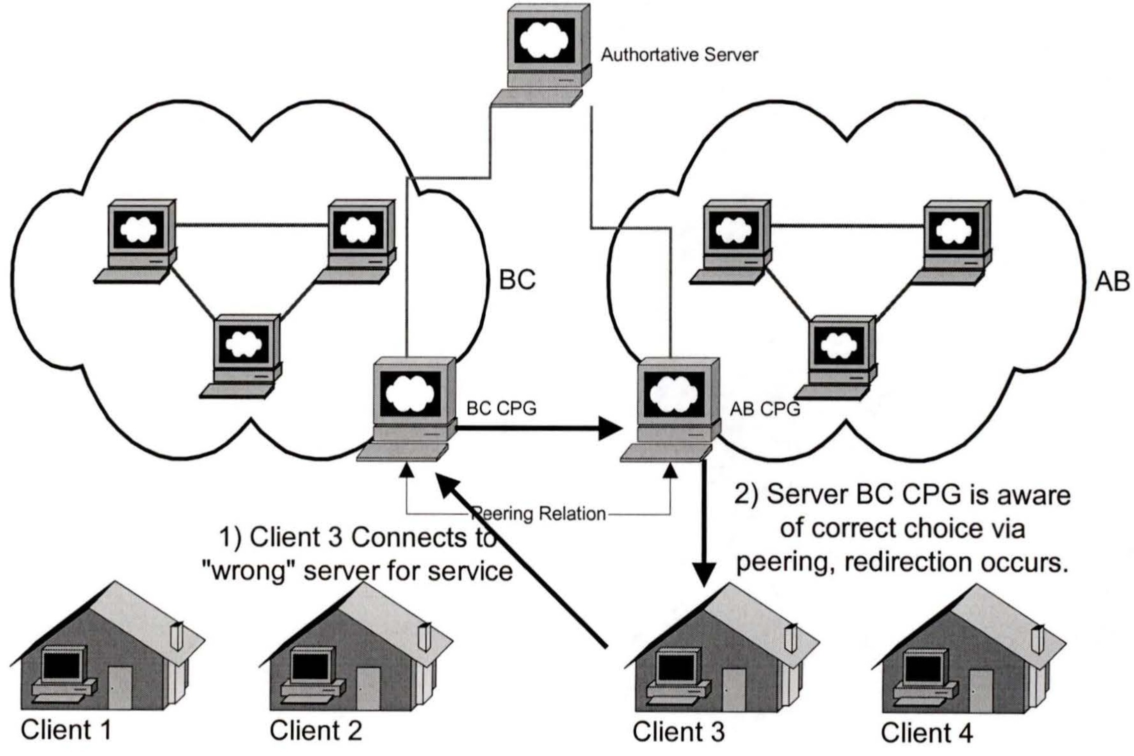


Figure 4.4) A gateway intercepts a request for an IP which it is aware of and so performs appropriate redirection immediately.

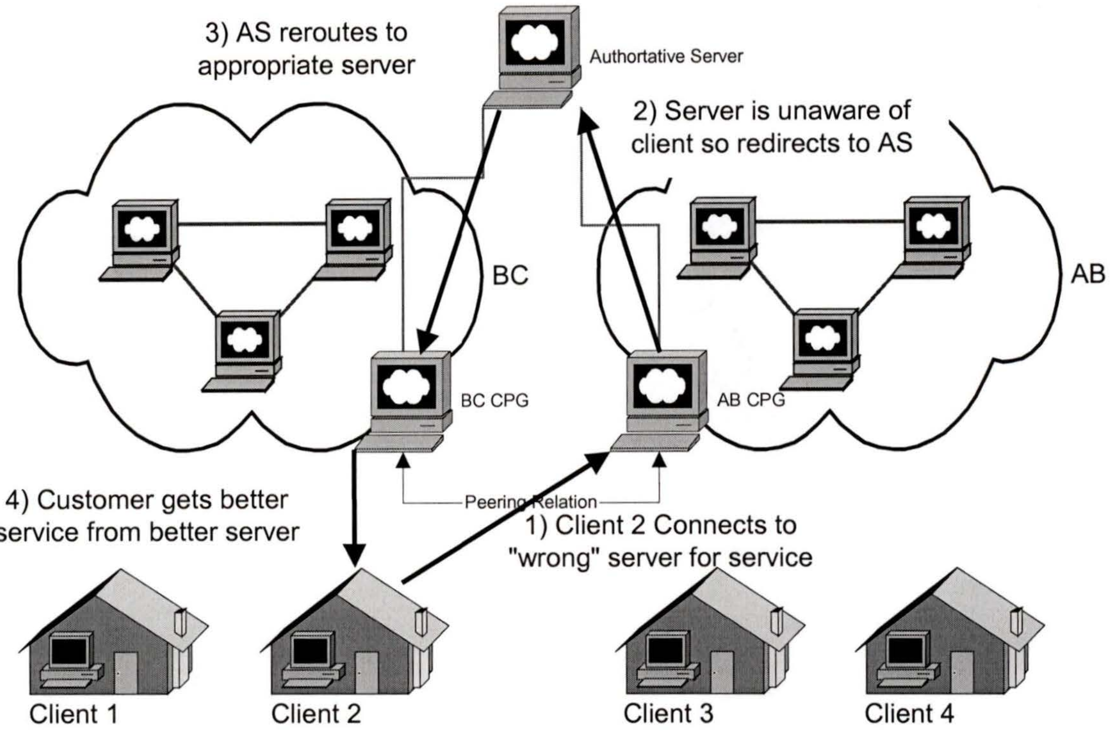


Figure 4.5) An intercepting gateway is unaware of customers IP address, thus redirection decision is made at authoritative server

The CPG system may also be embedded into a more specific request routing system that takes into account factors specific to the rerouting mechanism. For instance a DNS specific service could quickly be developed as a wrapper around the CPGP protocol. The request routing system (RRS) would then be a hybrid of DNS functionality and CPGP functionality. The DNS service could have more knowledge of how to create the A record or CNAME record.

A Simple DNS Scenario:

Following is a simple DNS scenario.

1. A client request for content is intercepted by DNS RRS system "X"
2. "X" uses the client IP address (which will probably be the DNS server for the client) to map to an appropriate server for that client IP range.

3. If "X" knows the server
    a. "X" finds the best server for this IP range in its own surrogate set, thus returns an "A" record to the clients DNS server, with "TTL" set to 0.

    else

    b. "X" finds the best server outside of its IP range, and knows to redirect the client DNS to there, thus returns an "NS" record with the destination filled in from its server
4. Client's DNS server continues with address resolution – returns address to client.
5. Connection is made to best server found.

This service also provides some guarantees of validity. These guarantees can only be provided with some description of the age of information. This is provided by a mechanism for updating. The age of the data can be guaranteed by ensuring updates happen within a given periodicity. The interval provided can be enforced by means of periodic updates, or pulling the data from the CPG's peers. The system administrator can set the frequency and therefore have a great deal of control over the rate of data flow.

Using periodic updating should not preclude sporadic updates from the peers, which have push functionality, in which the creator of the data initiates the communication, for important exchanges of data which effect the outcome of decision making.

These two types of updating, periodic and sporadic, provide assurance the current states match those of the CPG's peers.

Periodic Updates:

The information stored in the Metric Matrix will be the primary source of data for routing decisions in the system. Because of this, it is necessary to include a periodic system for ensuring consistency. This has some desirable effects:

- Ensures consistency at given time intervals, T_u
- Provides an upper limit to the age of information, useful for verification
- Acts to ensure that peers or children are still up and running
- Decreases network traffic by alleviating the reliance on sporadic updates

The periodic updates are accomplished by a pull mechanism, in which the recipient of the data requests the information to be sent. These will be done over reliable TCP connections that are already negotiated by the CPG's. Transmission will be as follows:

1) Time T expires, CPG sets all rows for the peered CPG to dirty
2) Sends RESET message
3) Receiver constructs and sends UPDATE message containing rows for sender (from aData)
4) Sender gets UPDATE message, and modifies its rows, removing dirty from each row
5) Remove remaining rows with dirty flags

This mechanism places the upper bound *T* on the validity of the data.

Sporadic Updates:

Sporadic updates will be used as the primary concurrency mechanism. There will be times when a content holder will wish to stop service of a particular server, or notify the CPGs of the existence of a new server. This will be done by advertising new values for the Delay to Footprint of a given footprint. These may fall between Periodic updates. Hence, this mechanism will be developed to ensure the update reaches the Authoritative system. These will also be used to address dramatic variations of server load and congestion, or any notification that changes the outcome of a decision.

Sporadic Update Transmission will be as follows:

1) UPDATE comes from a Surrogate that changes the outcome of server choice

2) UPDATE (row) message is sent to root and peers that this is advertised to

This information is pushed, and as such the message originator controls the rate of data flow.

4.3.3 Assumptions about the Environment

The CPG Data Exchange Protocol is the protocol used to update CPG Metric Matrices. The protocol must support existing data fields of the matrix, as well as newly defined data fields. The protocol's requirements are as follows:

- Support a pushing of data to reflect internal state change
- Support pulling of data to provide updating
- Correctness guarantees
- Efficiency and speed

In order to meet these, the protocol must also draw upon data from an administrative component to know who to communicate with, what information to get from them, and what information needs to stay private.

Because this is an application layer protocol, it has the advantage of being able to use transport layer attributes. This then raises the question of which transport protocol is most appropriate. The main concerns to be addressed are speed and correctness. These are embodied in the two main transport protocols in use, TCP and UDP.

CPGP specifies TCP. This protocol provides the most applicable transport layer correctness guarantees. This means that CPGP dos not have to be concerned with acknowledgment, but will need to provide other error and exception handling problems, henceforth called state-machine errors.

Another advantage of using TCP is related to authentication. By ensuring that CPGs must negotiate TCP connections, authentication may be done at the onset of a TCP connection once, and this will remain valid for the period of the session. TCP connections will then be maintained, and kept up via *Keep Alive* messages or some similar mechanism.

Furthermore, CPGP assumes that the messages (TCP segments) arrive in order, and either arrive, or an exception is raised. This behaviour is not standard in TCP, but is of orthogonal functionality to the protocol. These problems have been solved for multiple platforms and by multiple mechanisms so this is a valid assumption. Ensemble, a transmission controller produces these desirable results [46, 47]. It has also been shown that these concerns can be separated in order to more readily verify a protocol [5]. In a CPG system, this could be accomplished with the use of a pre-developed system, such as Ensemble, or with a simple sub-layer.

4.3.4 Vocabulary of Messages

This section provides a vocabulary of the messages involved in CPGP communication. Each message is named, then a description of the message is given, followed by an explanation of the procedure to follow when that message arrives. Each message solicits different behaviour depending on the state of a connection with a given peer. For any message arrives unexpectedly, the defined behaviour is to raise an exception and continue as if it were a TERM message.

The TERM message can occur during any state, and signals a removal of the peer, all its rows in eData, and all rows being advertised to it in aData.

HELLO

Message Explanation:

When a CPG wishes to initiate a connection either to the root (authority) or a peer, it must begin by making a TCP connection, and then sending a HELLO packet. This is done as an initial step to peering between two CPG's. There can only be one connection between a parent/child.

The connection will occur over a well-known TCP port.

Procedure Rules:

This message has dual functionality. It may be used to initiate a new connection with a peer or child, or to synchronize views with an existing peer. It is immediately responded to with a RESET message.

RESET

Message Explanation:

When a CPG wishes to indicate that it wishes a new copy of each of a peer's rows, it sends a RESET message. At connection time that all rows belonging to that peer are dirty, thus a RESET message is the first sent over a new connection. In the case of periodic updating, the rows are set to dirty before the RESET message is sent.

The response is therefore an UPDATE message completed with all the rows for the RESET originator from the CPG's aData section, as well as a NEW message completed with all new rows from the CPG's iData section that are in contention with the rows from the RESET originator in the CPG's eData section. This ensures that the recipient of the data rows does not need to perform any processing to identify which rows are updates to its current eData rows, and which rows are new additions, which require more processing.

Procedure Rules:

If the recipient is a peer:

This message solicits a complete listing of all rows for this peer, as listed in aData. If this message is from the root, the listing is complete from modified iData rows. Except for new rows, which are transmitted in NEW messages, the existing rows are sent in UPDATE messages.

The root should never receive a RESET message.

NEW

Message Explanation:

NEW messages are used to advise a peer, or the authority, that a NEW row is being advertised. These may be the result of a new surrogate coming online, or a surrogate deciding to advertise a new IP range, perhaps due to intercepting a large number of requests from the given IP range.

NEW messages contain row data. As an efficiency measure, and because TCP is an acknowledged, blocking protocol, multiple rows to be advertised are allowed in one NEW message, and there can be as many as kMaxRows in each message.

The rows are contained in the data portion of the packet, and are prepended with their size.

Procedure Rules:

If the recipient is a peer:

Upon receipt of a NEW message from a peer (CPGs cannot receive NEW messages from the root), a CPG checks to see if the rows contained intersect with any of its internal rows. If there are any intersecting rows, it ensures that they are in the aData matrix. If they are new to the aData matrix for that Peer, it creates a NEW message of its own, sending it back to the Peer. If the intersecting rows are already being advertised to that peer, it continues. After sending any data that it may have to send, it adds the row to its eData field, and updates the processing information.

This ensures that any contending rows that the recipient CPG has for the NEW rows being reported are added to the aData matrix for that Peer. Without this step the views of the common IP Range would not be complete to both CPGs.

If the recipient is the root:

The root behaves quite differently: it adds the row to its eData fields, and then checks for any intersecting rows from other CPGs. If any rows are found to conflict, it sends a CONTEND message to the originator of the NEW message, with a complete list of contending rows and contact information for each row, to initiate peering relationships.

UPDATE

Message Explanation:

UPDATE messages also contain the row data and their sizes. Again multiple rows to be advertised are allowed in one UPDATE message, as many as kMaxRows per message.

Procedure Rules:

UPDATE messages are much easier to handle than NEW messages. The receiving CPG merely needs to update the entry in the eData and pData matrices. The behaviour is identical for the root and peers.

TERM

Message explanation:

TERM packets will be used in uncommon scenarios when a CPG is aware that it will be shutting down its service. They may serve as notification messages.

Procedure Rules:

TERM messages are integral to exception raising in the CPG system. If a state synchronization error is detected, the CPG that discovers this error creates a TERM message and sends it to the peered CPG, or root. This should be used as a last resort to remove itself from the entire system. If a synchronization problem is discovered with a peer, a HELLO packet can be used to solicit a complete listing of the peers contending rows.

Upon receipt of a TERM packet, the peer should remove all references to the sending peer from aData and eData (which should also remove it from the pData fields).

KA – KEEPALIVE

Message explanation:

KA packets will be sent over connections in order to ensure that the connection is still live. They are immediately acknowledged with a companion KA message. This introduces another variable for each state.

Procedure Rules:

KA, or Keep Alive message are used to ensure the transmission connections are long lived, and have no effect on the running of the protocol.

CONTEND

Message Explanation:

Contend messages are used by the authoritative system to inform a CPG that one or more of the rows that it has advertised is in contention (intersects) with one or more rows that another CPG has already advertised.

Upon receipt of a NEW message, the authority examines the eData section for rows which each row in the NEW message intersect, and populates a CONTEND message with the results, while inserting the contact information for each row with that row. The CONTEND message is then sent to the CPG which has advertised the new rows.

A CPG receiving a CONTEND message must then create, or use an existing connection to the CPGs which it now has rows in common with, and inform that CPG of its new rows via a NEW row packet.

Procedure Rules:

The root is the only CPG that may send a CONTEND message, and so all recipients are peers. When the CPG receives a CONTEND message it extracts the rows in contention, adds them to its own eData matrix, also adding the intersection of these rows from its iData matrix to the aData with the peered CPGs reference. It then sends a HELLO packet to the peer, and when it receives the obligatory RESET packet, responds with all the data in its aData matrix that is destined for that peer.

REMOVE

Message Explanation:

Remove messages are used by CPG's to withdraw a row from both the authority and peers. The recipient is responsible to remove the specified row, and recalculate which existing rows in the receiving CPG's aData matrix are there solely because of the row being withdrawn. Care must be taken to not remove rows that are still in contention by other rows.

Procedure Rules:

Removing rows is nontrivial, as the CPG must also identify any rows in its aData that are solely dependent on the row being removed.

The root may simply remove the row from its eData and pData fields, as it has no advertised surrogates, however peers must identify these dependencies and remove them.

4.3.5 Message Format

Messages will have the following fields; TYPE, SENDERID, DATALENGTH, DATA, OPTIONS.

The packets will then have a format as shown in Figure 4.6.

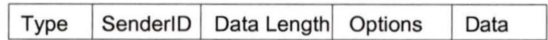


Figure 4.6) CPGP Message format

where Type is from Σ=(HELLO, RESET, UPDATE, TERM, KA, NEW, REMOVE, CONTEND) and will be represented by a byte.

In an UPDATE or NEW packet, row data will be stored almost identically to the matrix entry.[1]

---

[1] In our Java implementation this will actually be a precise mapping of objects, one row to one data option.

4.3.6 CPGP State Machine

Figure 4.7 presents a state diagram representing a single link of the CPG's as constructed from the procedure rules.

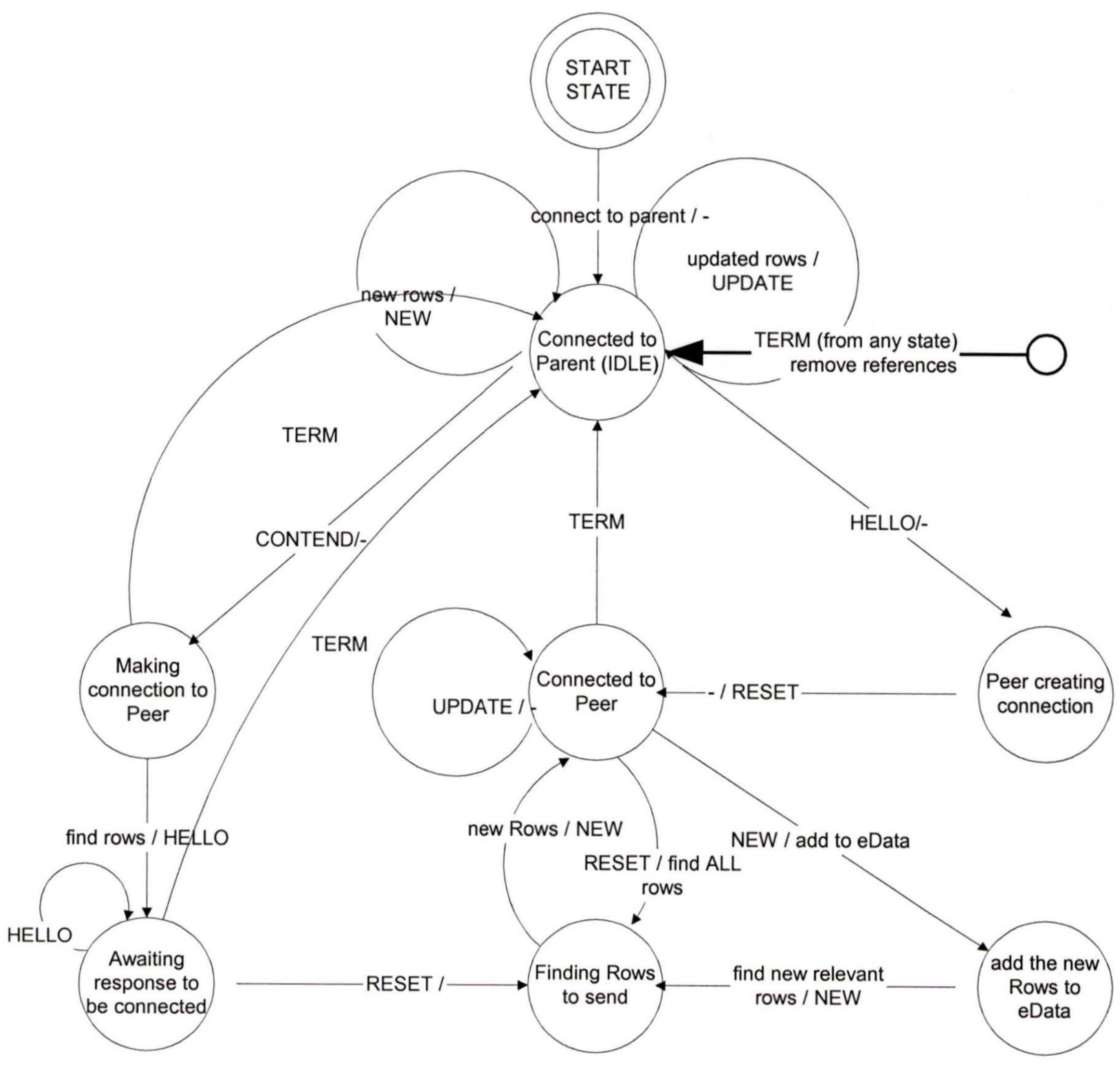


Figure 4.7) State transition diagram for CPGP

4.4 Message Sequence Charts

Message sequence traces are useful ways to visualize the packet transmissions and message interaction through the time axis. These are also interesting ways to show problems with a protocol, and to explain the procedures. Following are a number of message sequence charts showing scenarios of protocol interaction. See Figures 4.8 and 4.9. As indicated by the state transition diagram, once connected most interactions are unidirectional, without eliciting a response.

4.4.1 Simple Case Message Sequence Charts

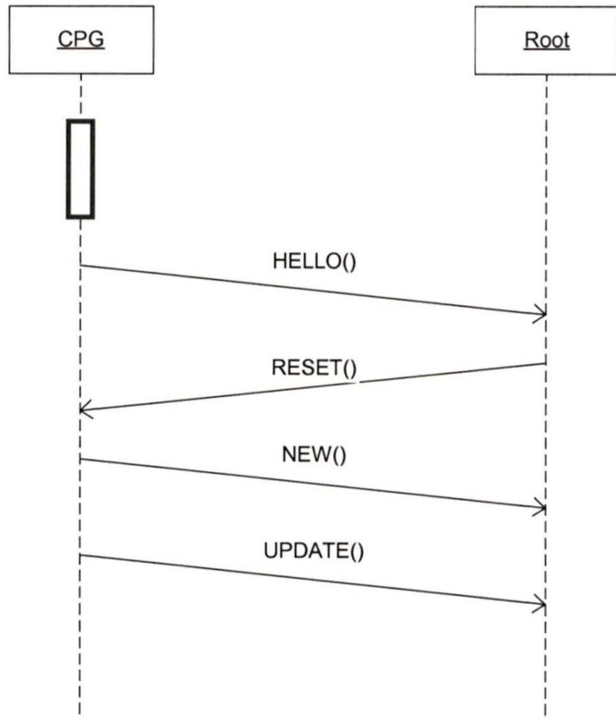


Figure 4.8) Message sequence chart for standard connection to root

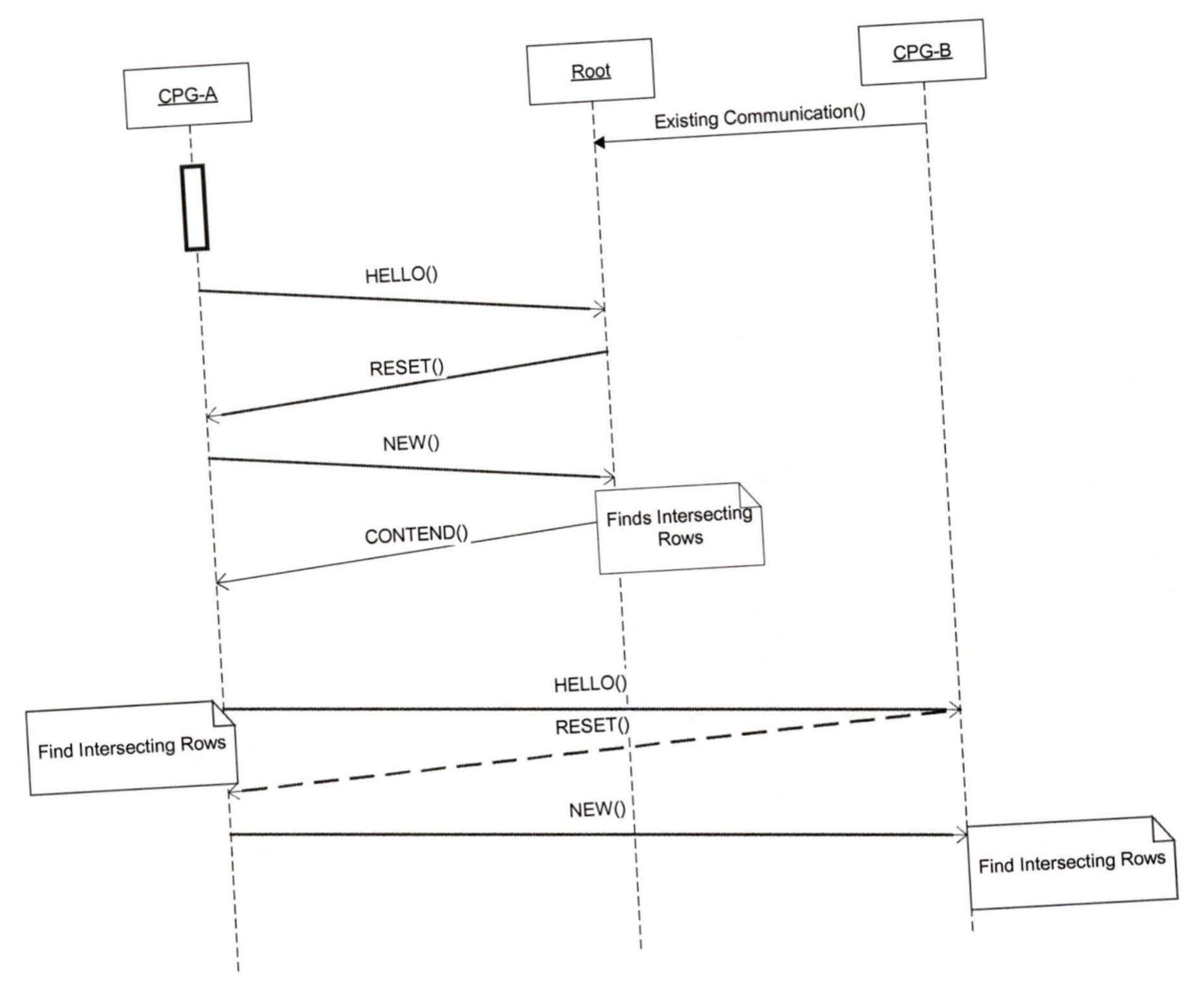


Figure 4.9) Message sequence chart for CPG's setting up a peering relationship

5. Protocol Validation

In order to validate the CPG protocol, this chapter performs a review of the possible state transitions. These states must have state invariants which are always true when the state machine resides in that state. The state transitions must only be possible if the transitions do not violate these state invariants. Once this has been shown we must argue that the state invariants are sufficient for the purpose of the protocol to guarantee that the protocol executes the intended function [5].

In order to validate the CPG protocol it is useful to break it down into some manageable pieces. Many protocol developers attempt to test their protocols only by running trials, hoping that bugs and cycles are exposed through testing. This is important, and helps to show how an implementation conforms to the requirements, however it does not provide any insight into the protocol itself, rather into a programmer's model of the protocol. By investigating the protocol outside of a computing environment the definition and validity of the protocol it may be more clearly shown.

Complex protocols may be virtually impossible to explore completely. As the number of variables rises, then the number of states increases as well. The Metric Matrix is a variable in the CPG protocol, and as its size is not logically bounded, so the protocol's states remain unbounded in number. As such it is easier to identify state transitions, and assert that by following these rules, state invariants hold true.

The state transition diagram relates to a single peering connection. It has been shown how the collaboration between peers can achieve a highly distributed and effective decision making system, however the protocol itself relates to a single peer, and its connection with the parent, or another peer. Therefore there may be many of these state machines

operating in conjunction on a single system. This creates the need for the system to co-ordinate the shared data with well-established mutual exclusion mechanisms.

Dead-lock and Live-lock are detected and handled through the use of the same time-out mechanism use to manage staleness of data. The variable T is the maximum time that the protocol waits for responses. This ensures that the protocol is never frozen in a state of non-progress.

The proof begins by focusing on how the peers become connected. There are two idle states, where internal variables can produce visible effects; Connected to Root, and Connected to Peer. From Connected to Root, to get to Connected to Peer, there are two paths. There is only one return path, if the peer disconnects. These two state machines may be viewed as being connected by these paths.

As each of the connections accept messages for only one peered CPG, the state transitions apply to a single connection, and the interaction of only those two connected CPG's. This means that while one peer may be setting up a connection, another peer may still be updating its rows, or even advertising new rows. The state transition diagrams should be thought of reflecting this, otherwise the number of states increases exponentially with each peer.

5.1 Connected to Root

While connected only to the authoritative server, a CPG may be operating at its peak efficiency. Requests can still be routed to it, and its surrogates may be active and productive. This could occur on a localized network that has a single point of entry via some gateway. If there were a surrogate at that gateway, it would then become the best choice for every IP address within the network, and quite possibly not wish to advertise for any IP addresses outside of its network. In this case there would be no contention, and it may operate smoothly and in the state of *Connected to Root*. If a CONTEND or HELLO packet arrives, it will then move towards the *Connected to Peer* state. This may

occur if another provider puts a surrogate within the network, or simply believes they have a faster server. See Figure 5.1.

As the protocol is a means of connecting precisely two CPG's for the purpose of participating in a network of CPGs it is sufficient to validate the protocol for sharing information between two such CPGs and assume that the results remain true for using the protocol to co-ordinate many CPGs. In order to validate the protocol, first assume the transport method cannot insert, reorder or distort messages [46, 47]. Next assume that lost messages are detected within a time frame, and are resent accordingly.

5.1.1 Connected to Root State Diagram

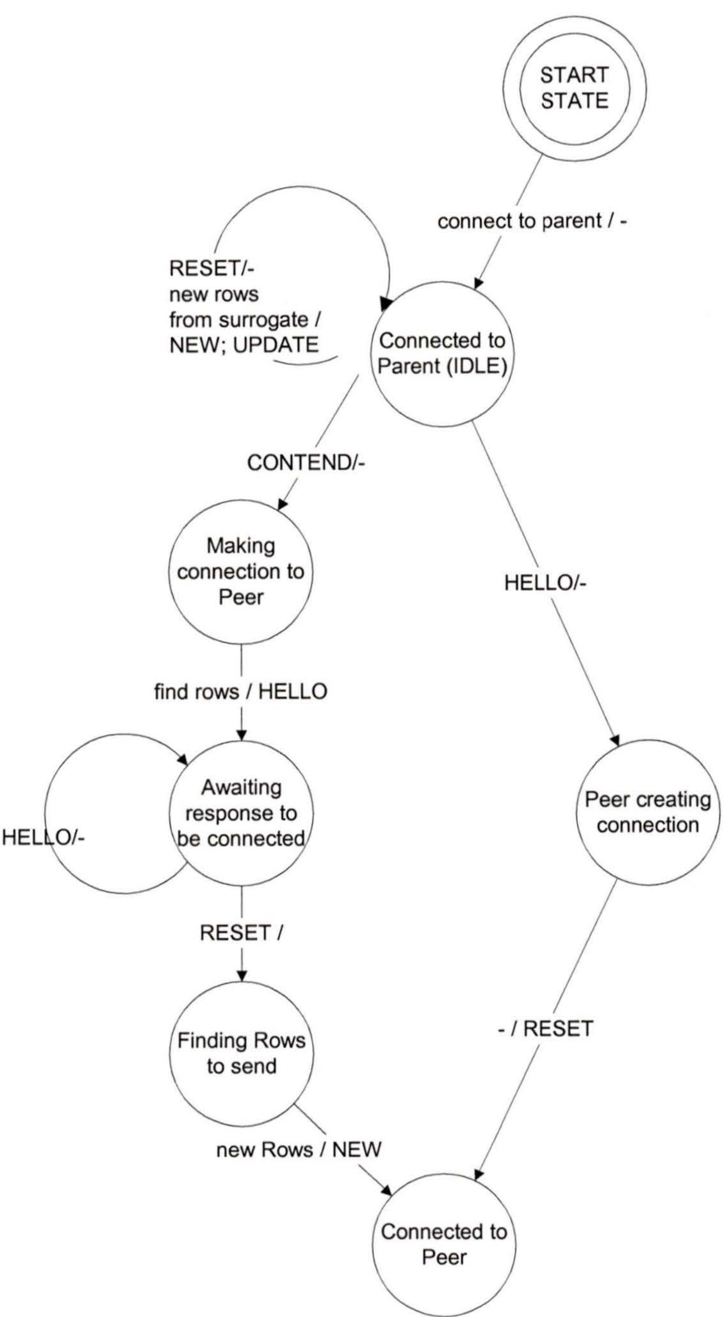


Figure 5.1) State diagram for states until a peering connection is made

5.1.2 State Invariants (Connected To Root)

State: **Connected to Root**	
Invariants: No incoming rows, therefore no updating of internal variables. Communication is solely with the root, thus the CPG cannot get UPDATE or NEW messages.	
Messages:	*Transition Rules:*
HELLO	Go to "Peer creating Connection"
RESET	Resend all rows to Root as "UPDATE"
UPDATE	Raise exception
NEW	Raise exception
CONTEND	Go to "Creating Connection with Peer" populate aData with rows
TERM	Move to Stop state

Table 5.1) Connected to Root state transitions.

State: **Making Connection to Peer**	
Invariants: No incoming messages, therefore no updating of internal variables. The CPG has just moved from an unconnected state to a setup state. Except for aData, the internal variables relating to this link are all clear. AData has the information pertaining to what the CPG needs to share on this link.	
Messages:	*Transition Rules:*
HELLO	No incoming messages
RESET	No incoming messages
UPDATE	No incoming messages
NEW	No incoming messages
CONTEND	No incoming messages
TERM	No incoming messages
Send HELLO	Move to "Awaiting response from Peer"

Table 5.2) Creating Connection with Peer state transitions.

<table>
<tr><td colspan="2">State: Peer creating Connection</td></tr>
<tr><td colspan="2">Invariants:

No incoming messages, therefore no updating of internal variables.

The CPG’s internal variables relating to this link are clean (eData and aData)</td></tr>
<tr><td>Messages:</td><td>Transition Rules:</td></tr>
<tr><td>HELLO</td><td>No incoming messages</td></tr>
<tr><td>RESET</td><td>No incoming messages</td></tr>
<tr><td>UPDATE</td><td>No incoming messages</td></tr>
<tr><td>NEW</td><td>No incoming messages</td></tr>
<tr><td>CONTEND</td><td>No incoming messages</td></tr>
<tr><td>TERM</td><td>No incoming messages</td></tr>
<tr><td>Send RESET</td><td>Move to “Connected With Peer”</td></tr>
</table>

Table 5.3) Peer Creating Connection state transitions.

State: **Awaiting response**	
Invariants: Connection, no internal rows shared, aData is populated, eData is empty (From this peer). Both Peers may have received a CONTEND approximately simultaneously, as signaled by receiving a HELLO packet at this time. This is solved by sending a reset immediately. This RESET cannot arrive before a HELLO packet as messages are not inserted or reordered.	
Messages:	*Transition Rules:*
HELLO	Send RESET : stay in this state
RESET	Move to "Finding Rows to Send"
UPDATE	Raise exception
NEW	Raise exception
CONTEND	Add those rows to aData, perform no other action.
TERM	Move to Stop state

Table 5.4) Awaiting Response state transitions.

State: **Send Rows to Peer**	
Invariants: This is a "non-existent" state that is shown to denote non-trivial activity. The CPG exits this state when it sends a NEW packet to its' newly negotiated Peer.	
Messages:	*Transition Rules:*
HELLO	Nothing accepted
RESET	Nothing accepted
UPDATE	Nothing accepted
NEW	Nothing accepted
CONTEND	Nothing accepted
TERM	Nothing accepted

Table 5.5) Send Rows to Peer state transitions.

This shows that in terms of state transition and invariants the state machine can be simplified to show only those states participating in the transmission of data. This does not illustrate other internal states, where variables are updated and as such is displayed for the sake of clarity and simplicity, as shown in Figures 5.2 and 5.3.

Because each path from *Connected to Root* to *Connected to Peer* is uninterruptible, the state invariants hold. Unless a message is re-ordered, and a data packet arrives before the completion of the connection, the state invariants hold. As we have assumed that messages arrive in order, by the intervention of a sub-layer, this cannot happen. Therefore this portion of our state diagram is valid.

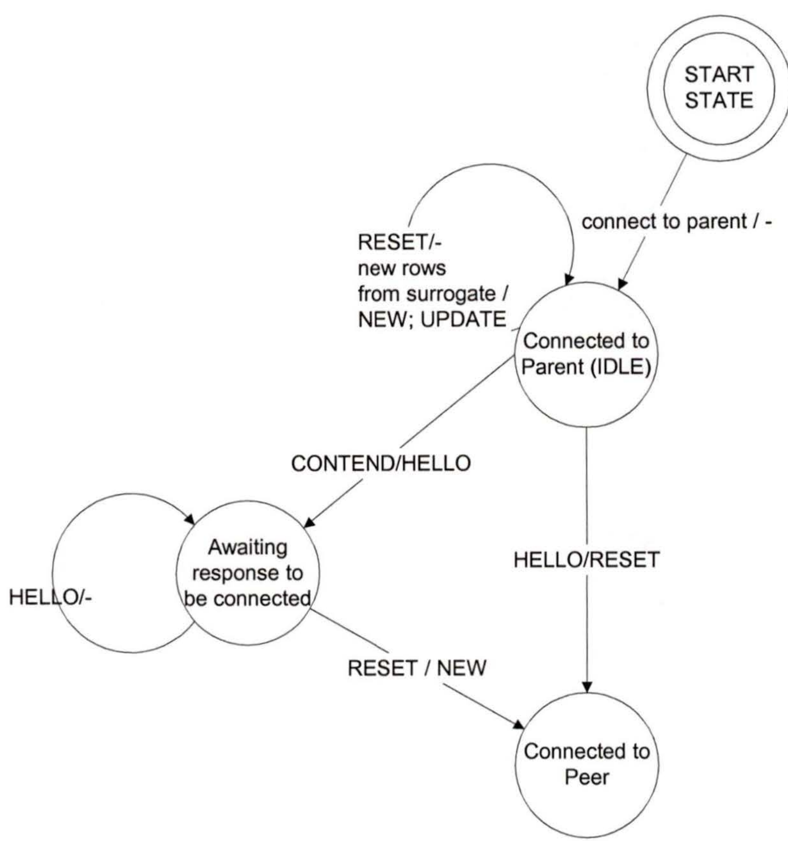


Figure 5.2) Simplified state diagram for CPGP until connection to peer is made

Once the CPG has reached the *Connected to Peer* state it idles, awaiting more messages. Virtually every message has significance while the CPG is connected to a peer; these are examined in the following section.

5.2 Connected to Peer

Connected to Peer is the state in which the CPG idles most of the time, awaiting messages from the protocol. There is no processing to be done during this state, and it responds to internal events by updating internal variables, and in some cases sending messages. These messages are what spark transitions through the states, those received due to internal updates on remote CPG's and those sent to the CPG by surrogates that it represents, see Figure 5.3.

All paths through the CPGP state machine eventually end in the *Connected to Peer* state, this is because this state is the goal of both CPGs, a distributed, consistent view of each other's relevant information. The only way to break from these loops is with the reception of a TERM packet, which brings the protocol to a stop state. This packet can be received at any time, and has the effect of resetting all internal variables related to this connection, and returning to an unconnected state. If a CPG loses a connection to a peer completely, it can also view this as receiving a TERM packet with no arguments.

5.2.1 Connected to Peer State Diagram

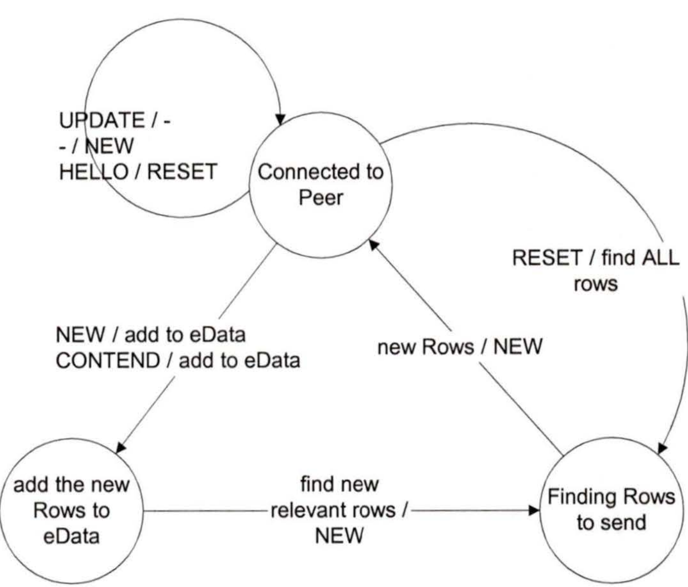


Figure 5.3) State diagram for CPGP in connected to peer idle state

5.2.2 State Invariants (Connected to Peer)

State: **Connected to Peer**
Invariants: This is the state the CPG idles in, waiting for messages from its Peer. During this state it accepts any message, and each of them has meaning. While in this state the CPG's view is synchronous with the peers view, and any decisions may be made with authority. The variables invariant is that the CPG's eData and its' Peers aData are synchronized, and that the local aData and remote eData are synchronized.

Messages:	*Transition Rules:*
HELLO	The CPG's peer has detected inconsistency, and has requested resetting the connection. Send RESET The CPG's eData is now invalid, send all rows in aData as NEW and remove all references in eData, to resynchronize rows.
RESET	Find all rows in iData that relate to the rows the CPG has in eData. Place these rows in aData. Go to Send Rows to Peer to send these rows. The CPG's eData and aData views are now unsynchronized.
UPDATE	Update the row in eData, as this is not new the CPG does not need to update aData, as views are synchronized.
NEW	New row for eData, check to see if there are any rows for that are unreported and contend for this new row. Move these to be sent. Move to "Add rows to eData"
CONTEND	Place the new rows in the eData, now the eData and remote aData are out of sync. Move to "add rows to eData"
TERM	Move to stop state.

Table 5.6) Connected to Peer state transitions.

State: **add rows to eData**	
Invariants: The CPG exits this state when it's added all the incoming rows to it's eData matrix. This state finds the correct rows to send to the Peer in the next state. This state exists when the local views of eData and aData are not synchronized, which means that the CPG's aData is not synchronized with the remote eData	
Messages:	*Transition Rules:*
HELLO	Nothing accepted
RESET	Nothing accepted
UPDATE	Nothing accepted
NEW	Nothing accepted
CONTEND	Nothing accepted
TERM	Nothing accepted
Find appropriate rows	Move to "Send rows to Peer"

Table 5.7) Add Rows to eData state transitions.

State: **Send Rows to Peer**	
Invariants: This state is responsible for sending the outgoing message of NEW rows. After this message is sent the CPG has regained consistency between its' aData and the remote eData. Upon sending the message the CPG exits this state and return to the "Connected to Peer" idle state. The rows to be sent are found in one of the states leading to this state.	
Messages:	*Transition Rules:*
HELLO	Nothing accepted
RESET	Nothing accepted
UPDATE	Nothing accepted
NEW	Nothing accepted
CONTEND	Nothing accepted
TERM	Nothing accepted
Send rows to peer	Move to "Connected to Peer"

Table 5.8) Send Rows to Peer state transitions.

This shows that a CPG can move from any state to a state of agreement of views between peered CPG's. This satisfies the function of the Content Peering Gateway Protocol, and as such validates the requirements. When the state invariants are met there is agreement within the CPG's and therefore they can make intelligent decisions on where to route requests based on user metrics.

This has been shown by the assertion of these state invariants. The transitions allow for logical flow, and movement of between the possible states. The state invariants are not violated by any of the state transitions, and under the message passing constraints this shows that the protocol is valid. This information exchange provides enough data to make an intelligent decision on where to route the requests. This is done by the proposed Best Server Choice algorithm, which is discussed next.

6. Best Server Choice (BSC) Algorithm Formalization

The primary objective of the BSC Algorithm is to discover the best choice for servicing a request that a CPG has intercepted. The best choice is defined by the needs of the content provider, and is then calculated from the metrics available to the algorithm. For example, some content providers will define best to be the server that can provide the data most quickly. The BSC algorithm uses CPGP as a means to collect enough information to support distributed decision-making. This algorithm takes into account several factors, including the distributed nature of the environment, the limitations of the data available, and assumptions about the size and nature of the data set. The BSC algorithm also assumes that an IP Range represents a geographic area, which leads to assurance of locality of the servers involved.

The algorithm is required to choose the best server based upon the user's IP prefix. Other data regarding the user may be used in more complex versions, including sessions and preferences. Different implementations will undoubtedly vary depending on differences in the technique used to store the data matrices.

The distributed algorithm can create significant efficiencies beyond merely distributing processing. By distributing the decision-making data with our algorithm, we also decrease the search space, and therefore the algorithm may run more efficiently. As each node that performs choices does so with a complete view of the data set it is interested in, it remains deterministic.

There is also a default mechanism which defines the server to be used in the event of no appropriate choice, this may be the content authority site, or another redirection node.

6.1 Algorithm environment

In order to refer to the algorithm, there are some key definitions of the environment. Begin by assuming the algorithm is operating on a sorted list of rows. The rows have multiple columns, and some of these columns may be identical, e.g. an IP Prefix advertised by CPG-A may be identical to an IP Prefix advertised by CPG-B. Therefore there is a linear order between rows defined as follows:

First by the first digit of the IP Prefix, all IP's of the form

M.x.y.z < N.u.v.w
If, and only if M < N.

Secondly by the second digit of the IP Prefix

a.M.y.z < a.N.v.w
If, and only if M < N

And so on for digit 'i' where 'i' <= length of the shorter of the two IP prefixes. If one IP prefix is identical to this point, the longer IP Prefix is considered larger.

a.b.c.M > a.b.c where a, b, c, and M are any arbitrary numbers

If IP prefixes are otherwise identical, the rows are sorted by delay, the smaller delay is considered "less than"

(a.b.c with delay x) < (a.b.c with delay y) : where x < y

If the rows are still considered equal, the final arbitrator is the destination IP of the rows, by the same rules as applied to the IP Prefix.

After all of this, if the rows are still considered equal, then the rows are considered identical and only one of the rows may be admitted into the data matrix.

6.2 Algorithm structure

The algorithm should provide the same results regardless of the location where it is run. This is true outside of tie-breaking mechanisms. Tie breaking may be allowed to favour the CPG that is performing the calculation, but should only be used in the event of a tie on all metrics of importance to the session.

Step 1: Select all rows from the data matrix that are larger than the shortest matching IP prefix that represents this IP, and smaller than or equal to the longest matching IP prefix that represents this IP.

1.1: Find the first Element that represents this IP address O ($\log_2 n$) (binary search for an element in a sorted list)

1.2: Find the last Element that could represent this IP address O ($\log_2 n$) (binary search for an element in a sorted list)

The algorithm now has a list of size m, where element 0 is the shortest matching IP prefix that matches this IP, and element m-1 is the longest matching IP prefix that matches this IP prefix. As this list is sorted by IP prefix, there is likelihood that there are rows within this list which do not represent this IP prefix.

Step 2: Filter the non-representative rows out of our list. O (m) where m is the size of the sub-list (worst case m = n)

Step 3: Apply rules to each row of the matrix, finding best possible match. Currently the only rule is minimum delay, and taking the first element found as a tiebreaker.

6.3 Algorithm Efficiency

These steps are divided into different functions, Step 1 deals directly with the datamatrix. As it is desirable to minimize time that the data is accessed this is done in the datamatrix class itself, and returns a reference to a new datamatrix which can then be searched. This

means that the maximum time that the matrix is locked in a critical section is a function of $2(\log_2 n)$, where n is the number of rows in the pData matrix. This portion of the algorithm should be kept at a minimum, as the matrix must be locked to access it, and represents the most important critical section in the algorithm.

Step 2 then finds the rows to which we want to apply our rules; it is a filter that could then output into another thread directly if the implementer wished. Step 2 may be incorporated into Step 3; the filter could be applied and any rows that pass the filter could immediately be passed through the rules. Doing step 2 separately we find the filter takes time $O(m)$. The worst case is that $m = n$, and as such the filter step may be as bad as $O(n)$.

Step 3 uses the output from Step 2, and applies the selection rules to it. Implementations will differ on the information available, Service Agreements, and arbitration mechanisms. Again, if all the rows pass the initial filter, this step could once again be $O(n)$. This brings the algorithm complexity to $O(2(n) + 2(\log_2 n))$.

There can also be many variations on the implementation; this merely outlines the requirements of the protocol. For instance if an implementer had access to reliable iterators, and the matrix rows were protected with semaphores, the entire process could be reduced to simply finding the first possible record $(O(\log_2 n))$ and then iterating through the records one at a time, first checking if it is a representative and then applying the rules to those that are, stopping as soon as the rows become larger than the clients IP Address. The worst case of this would be $O(n + \log_2 n)$. For worst case analysis, the initial search could be left out entirely decreasing the worst case complexity to merely $O(n)$.

Since Step 1 separates the data being searched from the main repository, Steps 2 and 3 may be performed on entirely different processors. This allows for a high degree of multithreading and parallel processing. If multiple processors are used, then the critical section of Step 1 becomes the primary concern, with a complexity of $O(\log_2 n)$.

The implementation used in the CPGP test suite keeps the various steps separated, so as to allow for cleaner testing and verification, and more clear definition of the algorithm.

The algorithm implementation can use any method to make these steps more efficient that the implementer likes. The rules of selection must remain the same to preserve deterministic behaviour, however, depending on the nature of the peering group it may be natural to run the algorithm upon receipt of updates, or at the time of user requests. The proposed implementation is focused on a less dynamic group that is hosting large, multimedia data.

6.3.1 Code Listing for BSC algorithm

```
//FROM MATRIX.JAVA Steps 1 and 2
/** findMatches
find matches will search through the given data matrix
and return a new datamatrix with all matching IP addresses
this takes into account subset matches (prefixes)
as this is syncrhonized, it cannot be recursive
	**/
synchronized public datamatrix findMatches(ip baseIP) {
// first find a matching first digit
// this could be done via an nlogn algorithm for better efficiency
datamatrix returnMatrix = new datamatrix();
int left = 0;
int right = size();



// we'll try to match up to the baseIP length
// and we ensure that our search range >= 0
//while(left <= right && baseIP.compareTo((ip)((row)elementAt(left)).footprint) > 0) {
while(left <= right && (((ip)((row)elementAt(left)).footprint).compareTo(baseIP)) >= 0) {
// find matching ip left as first index
left++;
		}

// if left >= right, then we are out of elements, no matches
if(left == right) return returnMatrix;
//now make right the last element
right--;
// now find right index
// while our right is right of left, and is bigger than our ip
```

```
while(right >= left && baseIP.compareTo((ip)((row)elementAt(right)).footprint) <= 0) {
right--;
            }

// from left to right, see if the footprint represents this ip
// add it to the list if it does
for(int i = left; i <= right; i++) {
if(((row)elementAt(i)).represents(baseIP)) {
returnMatrix.add(elementAt(i));
System.out.println("adding element");
                    }
            }

System.out.println("made list left: " + left + " right: " + right);
return returnMatrix;

    }



//CPG.java THIS IS where STEP 3 is implemented.
public void BSC(ip clientIP) {
datamatrix matching;
long mindelay = Long.MAX_VALUE;
int minrow = 0;
// this will get the processing data, then find the rows.
System.out.println("matching");
matching = (metricmatrix.getp()).findMatches(clientIP);
System.out.println("matched");
System.out.println("Got the matching matrix: " + matching.size());
for(int i = 0; i < matching.size(); i++) {
if(((row)matching.elementAt(i)).delay < mindelay) {
mindelay = ((row)matching.elementAt(i)).delay;
minrow = i;
            }
System.out.println("entry: " + ((row)matching.elementAt(i)).footprint);
    }
if(mindelay == Long.MAX_VALUE) {
System.out.println("default choice");
return;
    }

// the minimum row is the best match we have
System.out.println("minimum delay: " + mindelay);
System.out.println("destination ip: " + (row)matching.elementAt(minrow)).destinationIP);
return;
}
```

In order to further test our protocol, a limited live test was conducted to show its functionality.

7. Prototyping and Testing

In order to show the functionality of the CPGP protocol, we have implemented a suite of applications to simulate the interconnection of content networks. The simulation mimics the various components of a content network, and embodies the regular behaviour of the protocol. It serves as a prototype inter-CDN system, showing that CPG's can make effective choices in a coordinated peer-to-peer system.

7.1 CPGP Test Suite

Following is a brief description and user manual of the CPG protocol test suite. This set of programs work in conjunction to show the feasibility of the CPG Protocol Specifications. Further tests were accomplished with a live request routing system using On-Demand URL rewriting that communicates with the CPGP Test Suite.

7.1.1 Program Environment

The CPGP Test Suite was written in Java 1.3.1. Therefore it is cross-platform, and may be run on any system supporting Java 1.3.1. The system must have a Java interpreter installed. We have tested the code under Linux and Windows 2000, running on processors ranging from 500Mhz to 1.5Ghz. The program performs its communication through network sockets, but can be run on a single computer communicating with itself.

7.1.2 Program Overview

The purpose of developing the CPG Protocol Test suite was to adequately examine the feasibility of the CPG Protocol.

The main window of an instance of the program is the representation of a CPG, the communicating component of a content network. The CPGs display their internal states in the window, and have buttons to perform various actions.

The CPGs can communicate with each other over a network connection, or with their surrogates. The surrogate representatives are simply threads running within the CPG's program space; they are meant as a way to simulate real surrogate's information, namely delay to a certain IP space. The method of communication between real surrogates and their representative CPGs is not part of the concern of the protocol, however the CPGP test suite does it with callbacks between threads of a process.

The CPGs accept information from their surrogates as internal data. This is dealt with by adding the data to their iData matrix; most CPGs will then also add the data to their own processing matrix. The processing matrix is essentially a union of the internal and external data matrices, whereas the advertised matrix is an intersection of the external matrix and modified versions of the rows in the internal matrix.

The CPGs communicate with each other through long-lived TCP connections. The connections are maintained by KeepAlive messages (see Protocol Specifications). Any data collected from another CPG is marked as external data, and stored appropriately.

7.1.3 Getting Started

In order to run the CPG test suite, a Java Virtual Machine must already be loaded on the computers to be used in the simulation. The program was developed using Java 1.3.1, but is compliant with the specifications from Java 1.2. If Java 1.2 is used, the Java Swing libraries must also be available; these are standard in Java 1.3 interpreters.

With a working Java environment, one may start any number of CPGs on any number of computers. One may wish to start by loading several CPGs on the same computer, using different port numbers. The program is normally begun from the command line by typing.

Type:

java cpg <name of this CPG> <y/n> <port Number>

where the name is a one-word name for the CPG, y/n indicates whether or not the cpg is a root, and port number is the port to which other CPGs may connect to this CPG. It is suggested to start at least 3 CPG's as well as the root so that one can see how they communicate with each other.

The first CPG should be the root, and have a "y" as the second argument.

Two CPGs and the root may be started on ports 1500, 1501 and 1502 with these commands:

java cpg root y 1500
java cpg CPG1 n 1501
java cpg CPG2 n 1502

It is not a requirement that to start with port 1500; rather they must follow the standard port scheme. Ports < 1024 are known as privileged and as such are reserved: any port in the range 1024 – 65535 which is not used by another service on the computer can be used.

When the program is started, the user will be asked to enter an IP address; this is a mock IP Address and represents the reported IP Address for this gateway in rows. This may be anything the user likes. This is not the IP Address that other test programs will connect to. Enter all IP Addresses as dot-delimited numbers between 0 and 255, see Figure 7.1. This IP Address will be only be used for display purposes, as one may run more than one CPG

on a single computer. To more accurately simulate a network environment, CPGs should be run on many computers, and their actual IP Addresses entered.

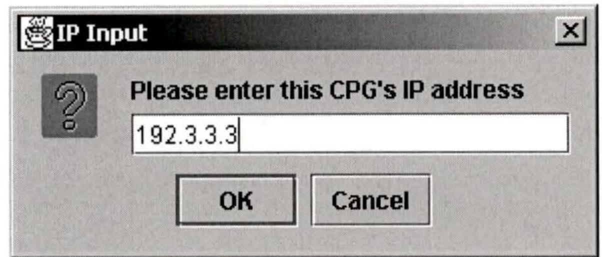


Figure 7.1) The CPGP Test Suite IP Address Input dialog

7.1.4 Using the program

Setting up the CPGs:

Now that some CPGs are active, in order to be useful they will need to be connected. Furthermore, each CPG will need at least one surrogate.

The labels are as follows:

iData – Internal Data.

This represents the CPG's view of its own network. The destination here is the IP address of the surrogate. This is valid because the CPG may know the internal state of its own network. The surrogates populate the iData matrix when they create new rows.

eData – External Data

eData is the CPG's view of the world outside. The destinations here are the designated IP's of the other CPG's. This information is collected via peering relationships with other CPGs. The root is the only exception, where it is gathered from its nodes.

aData – Advertised Data

aData is the view of the internal structure that this CPG shares with its peers. The destinations listed are modified to reflect the designated routing gateway for this content network. The routing gateway is responsible for re-routing requests to internal servers.

pData – Processing Data

pData is the aggregate of External Data and Internal Data, it stores its own destinations here, as well as the external destinations, unchanged. This is the information the BSC algorithm will reference in order to make an intelligent decision of how to best route a request.

The user should start with the root CPG; this is the CPG that has a "y" as its third argument. This CPG is the Authoritative system for the peering group being constructed. It will have no internal rows, thus no advertised data. Each of the other CPGs will connect to the root, and subsequently report their new rows to it. The user will not interact with the root at all, but must be aware of the IP address and port that the root is connected to.

See figure 7.2 for a view of a preset CPG.

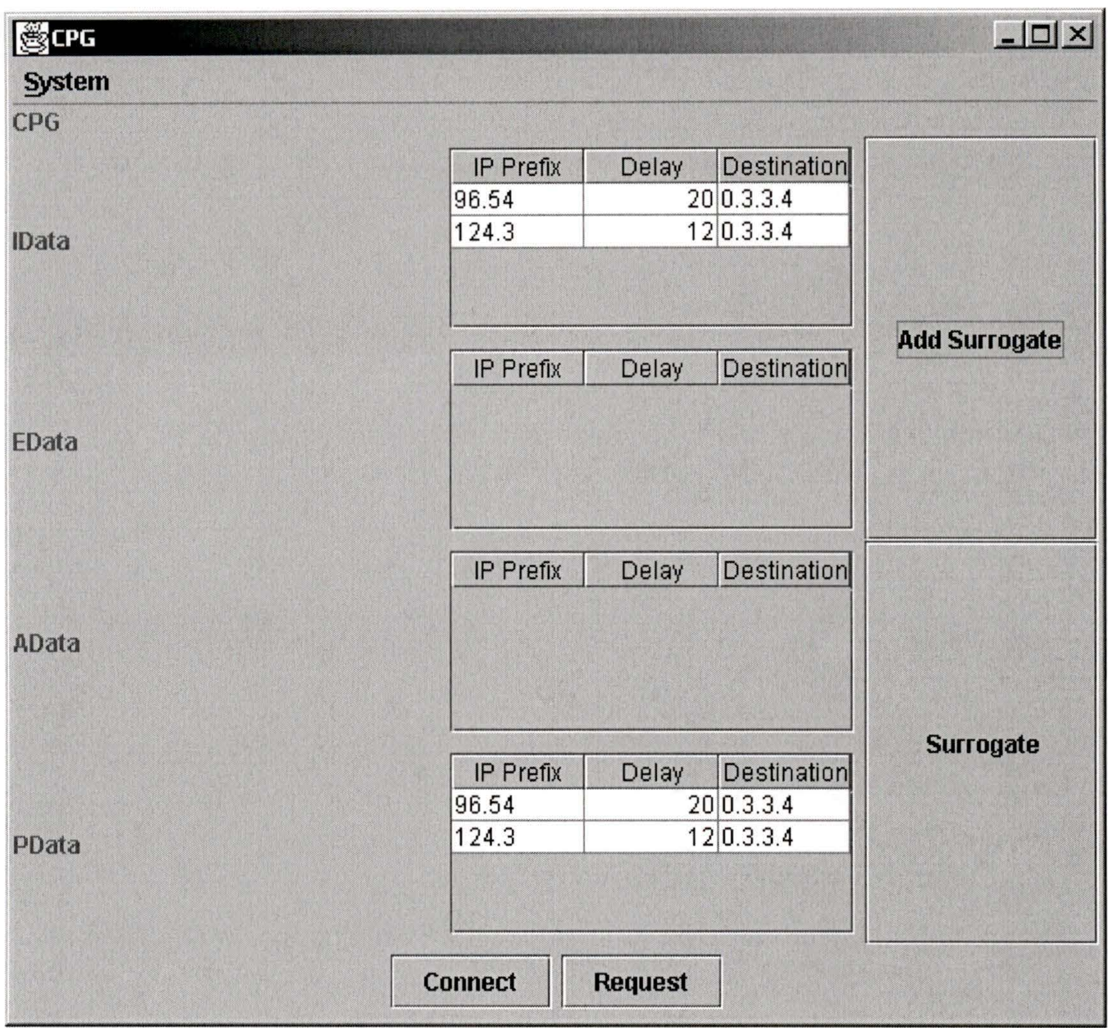


Figure 7.2) The view of a CPG with Internal and Processing Data. There are no rows being contended thus no other CPGs have delivered external data

The content network connections should create some areas where contention is present. These are identified by the authoritative system, and communicated to each other through a peering relationship. This is illustrated in Figure 7.3.

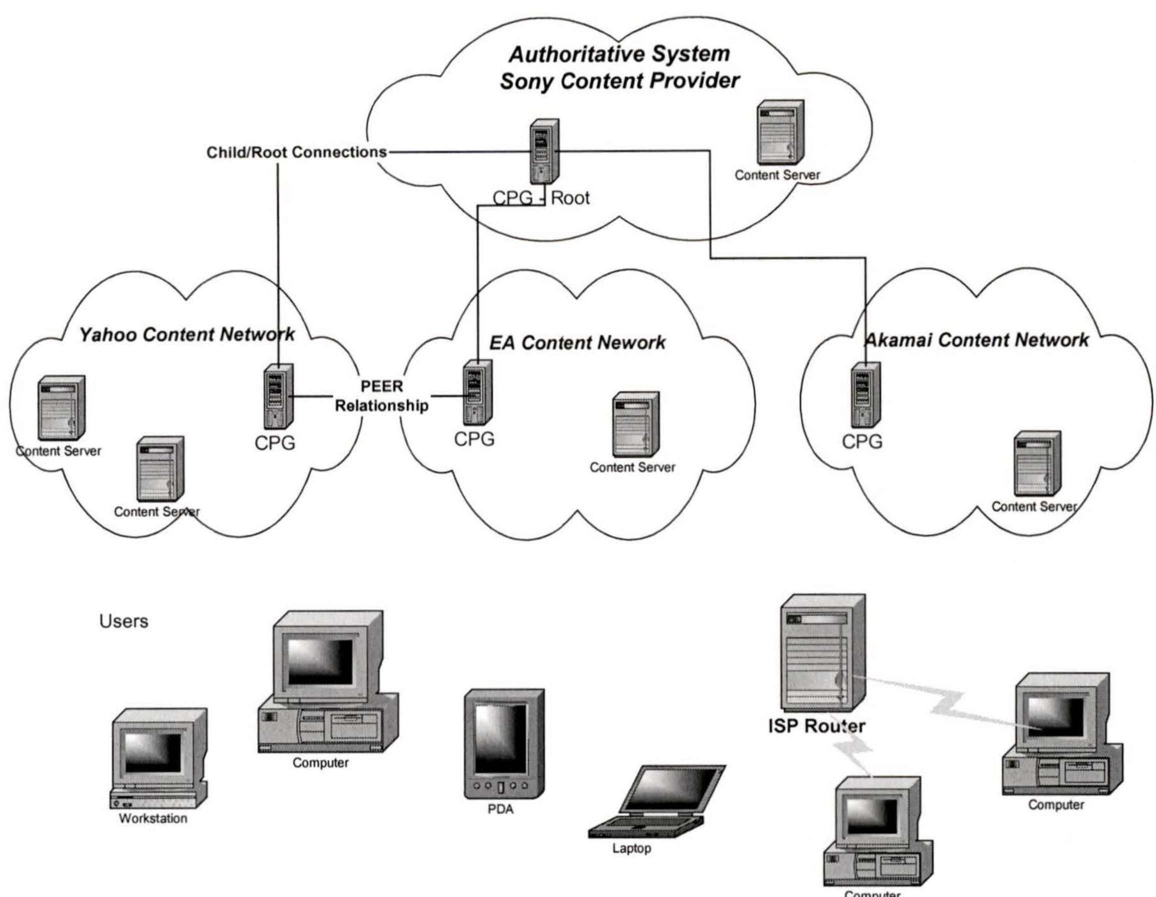


Figure 7.3) An illustration of connected content networks, where Yahoo and EA networks have peered via their CPGs, users can get content from any content server.

As in live systems, the children are responsible to connect to the root. The root has no internal surrogates, and as such has no iData. If an implementer of the root wishes to have surrogates running within the root, they must instantiate a new CPG to reflect the content network of the authoritative system, and have it connect to the root as normal. Due to its role as the coordinator the root is separated.

In order to connect a CPG to the root the *Connect* button is clicked. A dialog box will appear asking for the IP address of the root. If the root is being run on the same device, nothing needs to be put into the box, simply enter should be pressed. If the root node is being run on another computer, the IP address of the other computer should be entered into this box followed by pressing enter.

The command windows should now show some output, typically it will be "message sent…" and "packet received"

Once the CPGs are connected, they are ready to communicate. Clicking on one of the CPG's *Add Surrogate* button will prompt the user for some made-up information. The surrogate will represent a server that hosts data for this CPG's network. The user may designate any IP they wish, as this is just a representation for the surrogate internally.

There is no requirement that CPGs are connected first. Surrogates may be created and populated within a CPG to see what happens when UPDATE or NEW messages are batched.

It is recommended that initially only one surrogate is created for each of the nodes. Surrogates should not be added to the root. Any surrogates added to the root will not cause a malfunction, but they will not be visible to other CPGs. Any information added or removed is replicated to the CPG and stored appropriately, the eData field at the root has modified IP destination to indicate that it is now the CPG that the requests will be routed to, whereas in the rows in pData and iData on the local CPG the information is identical.

Rows entered in the surrogate are the advertised footprint of that surrogate. It may have many footprints. The footprints are IP Prefixes and the delay is a number representing the delay time to that footprint, see Figure 7.4.

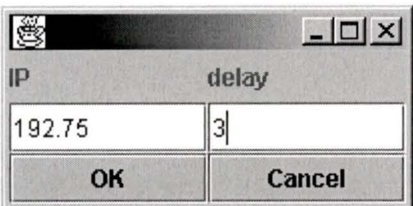


Figure 7.4) A view of a surrogate, with a row input dialog in the CPGP Test Suite

Once the surrogate's behaviour has been observed, creating a surrogate for another connected CPG will demonstrate the inter-CPG communication. Following the same steps to create a surrogate; when a row is added to the child's surrogate the information from iData is propagated to the root. Now the root is able to recognize external data from the second CPG (represented in the eData field) and check if it conflicts with any rows from the first CPG. The collection of those is the processing data, which is the repository that will be referred to for routing decisions.

If any rows do conflict the CPGs create links to each other and begin exchanging the information which conflicts, this will appear in the aData and eData sections. Figure 7.5 and 7.6 show the views of two CPGs which contend on one of their IP Ranges, they peer, or share, with each other the rows that are relevant to that IP Range, but not their other rows. The pData in each case is populated with a complete view of their advertised areas,

their eData mirrors the aData of the other CPG, and the iData contains the CPG's view of its own network.

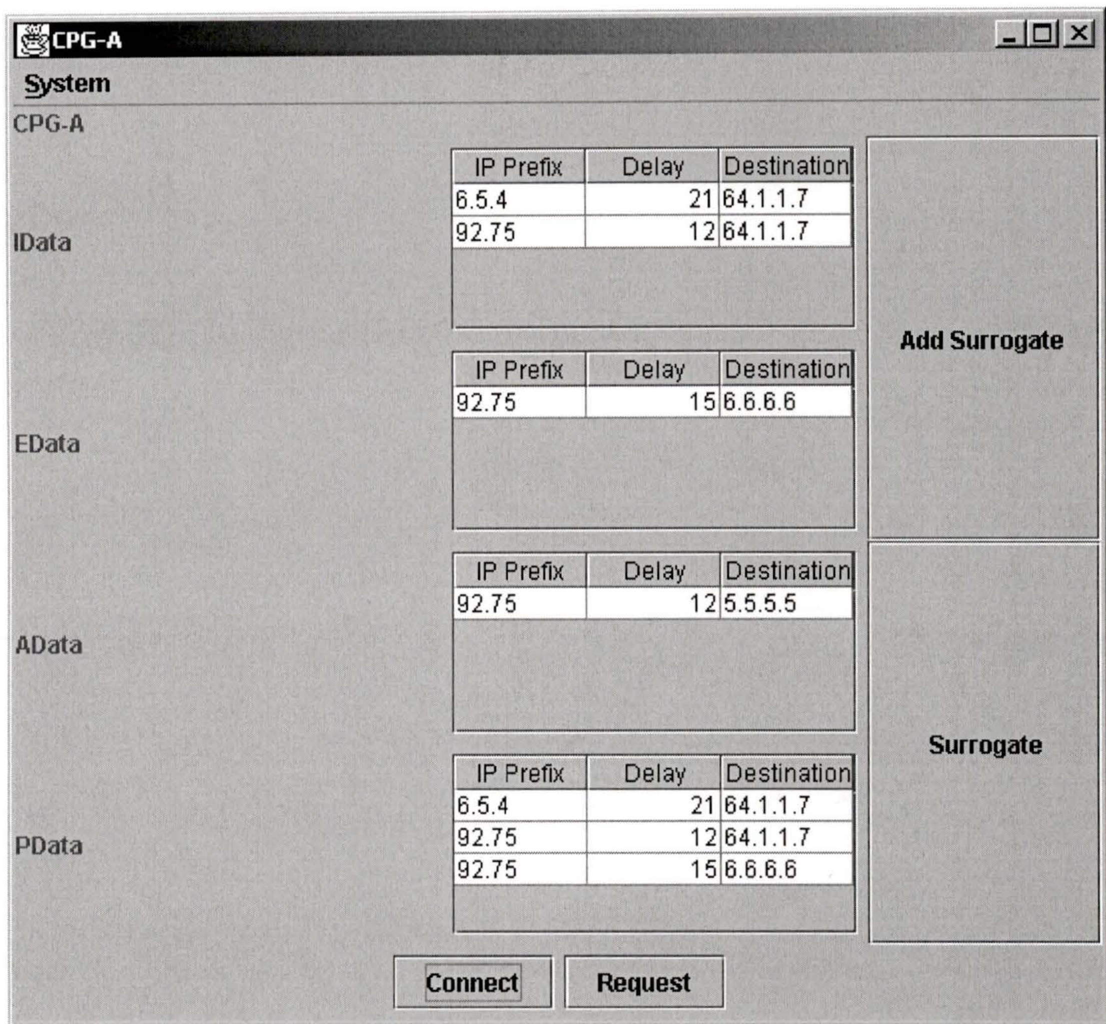


Figure 7.5) CPG-B Conflicts with CPG-A for IP Range 92.75, thus it has populated its aData with its own information, and its eData with the row received from CPG-A

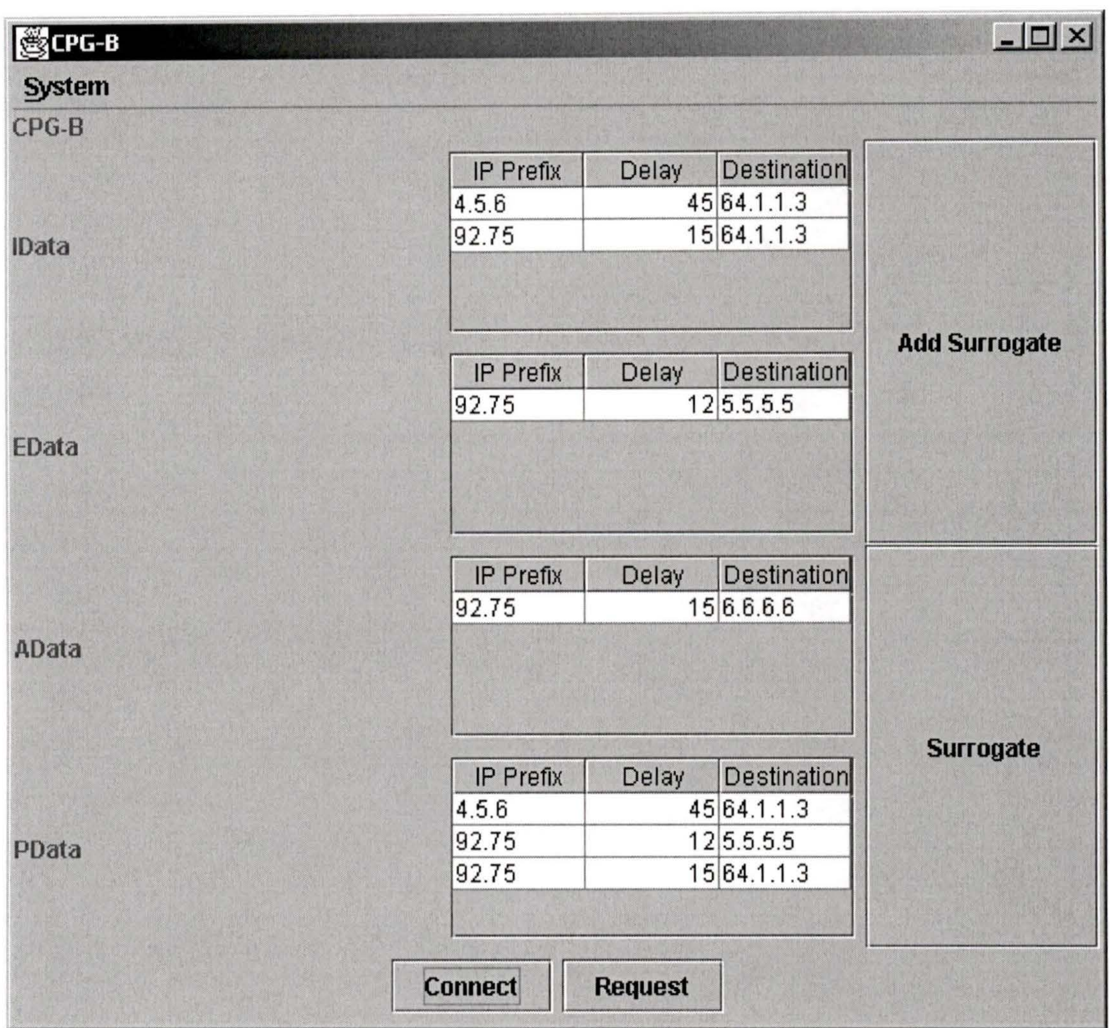


Figure 7.6) CPG-B Conflicts with CPG-A for IP Range 92.75, thus it has populated its aData with its own information, and its eData with the row received from CPG-A

Figures 7.5 and 7.6 further illustrate the ability of CPGP to limit peering of irrelevant information.

7.1.5 Testing Requests

As there are many levels of the Request Routing System, the only level with a complete view is the root. This represents the Authoritative System. One of the requirements listed

in [1] that there should be an Authoritative System that can consistently route requests, or pass the routing decision to another system in either an iterative or recursive manner.

Currently the AS (Authoritative System) will make a choice itself depending on the information at hand, however it is equally feasible to have it recursively ask a destination IP for a new answer. Also, any CPG can make an intelligent decision for any data which it advertises for, and if asked to make a choice for a destination which it does not participate in, it will deliver a default answer, which would logically be matched to the root of the system. Any system that does recursively resolve routes should ensure at the redirection mechanism level that there are no loops.

One of the primary benefits of the CPGP system is that it may have a designated IP Address to route requests to. This may be the CPG of the CN that the chosen surrogate is in, or it may be another decision-making server (like an intelligent switch). Any iterative or recursive request should be routed to that IP address, by whatever routing mechanism is in place.

In order to test the Request Routing System constructed, the "Request" button should be pressed. A dialog will appear to ask for an IP address representing the client. A 4 byte IP address separated by dots should be entered.

The program then outputs the rows that can serve the IP address making the request. It then chooses the best row based upon the delay metric.

This can be done at any of the CPGs, as they all are capable of making decisions.

7.2 Live Test

Further testing was done by constructing a live test to show a complete Request Redirection System in action. This system is implemented with a great deal of functionality that shows not only the Content Peering Gateway Protocol in action, but also a common means of request redirection being used. The implementation uses well-known products which are freely available.

7.2.1 System overview

This CPGP system is best demonstrated by integrating it into a live system. The World Wide Web (WWW) makes an exceptional test-bed for this, as URL rewriting is already recognized as one of the primary means of request routing, and there are many tools for performing these operations. We have implemented an On-Demand URL rewriting system that directs clients to different web servers for certain objects depending on the clients IP address. This demonstration is meant to be a functional request routing system, using On Demand URL Rewriting (see Section 3.3.3). The system uses the CPGP BSC algorithm, along with java server pages to change the source of an embedded object in a web page depending on the IP address of the client viewing the page. The same page can be hosted on multiple computers, with inline reference to objects such as images or other media content, updated while the page is being loaded. The test implementation uses Apache web servers, and Jakarta-Tomcat Servlet Engines as the web delivery mechanisms, both of which are free programs. Java Server Pages (JSPs) and servelets were used to perform the communication with the CPGP system. The CPG protocol suite was extended to accept connections to resolve BSC (Best Server Choice) requests. This connection occurs on port 5005 and used as a service layer to the Tomcat Servlet Engine as illustrated in Figure 7.7.

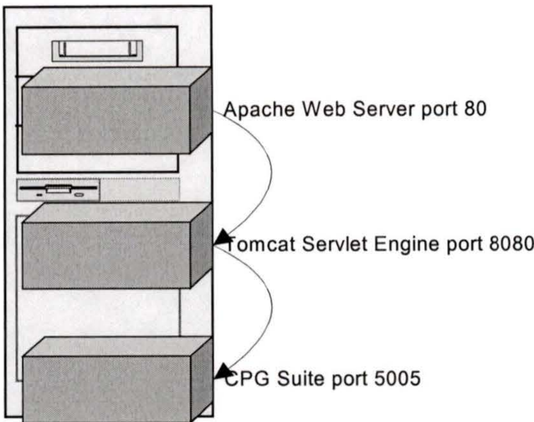


Figure 7.7) CPGP Protocol Service architecture

The servers were then configured to advertise different IP addresses on a local area network, as displayed in Figure 7.8, and direct requests to one of two servers depending on which client was requesting data. The clients were generic PC's running either Microsoft Internet Explorer, or Netscape Communicator. These web-browsers both support functionality beyond interpreting HTML 1.0, however the page was intended to merely display the client IP and which server the client would be accessing for content.

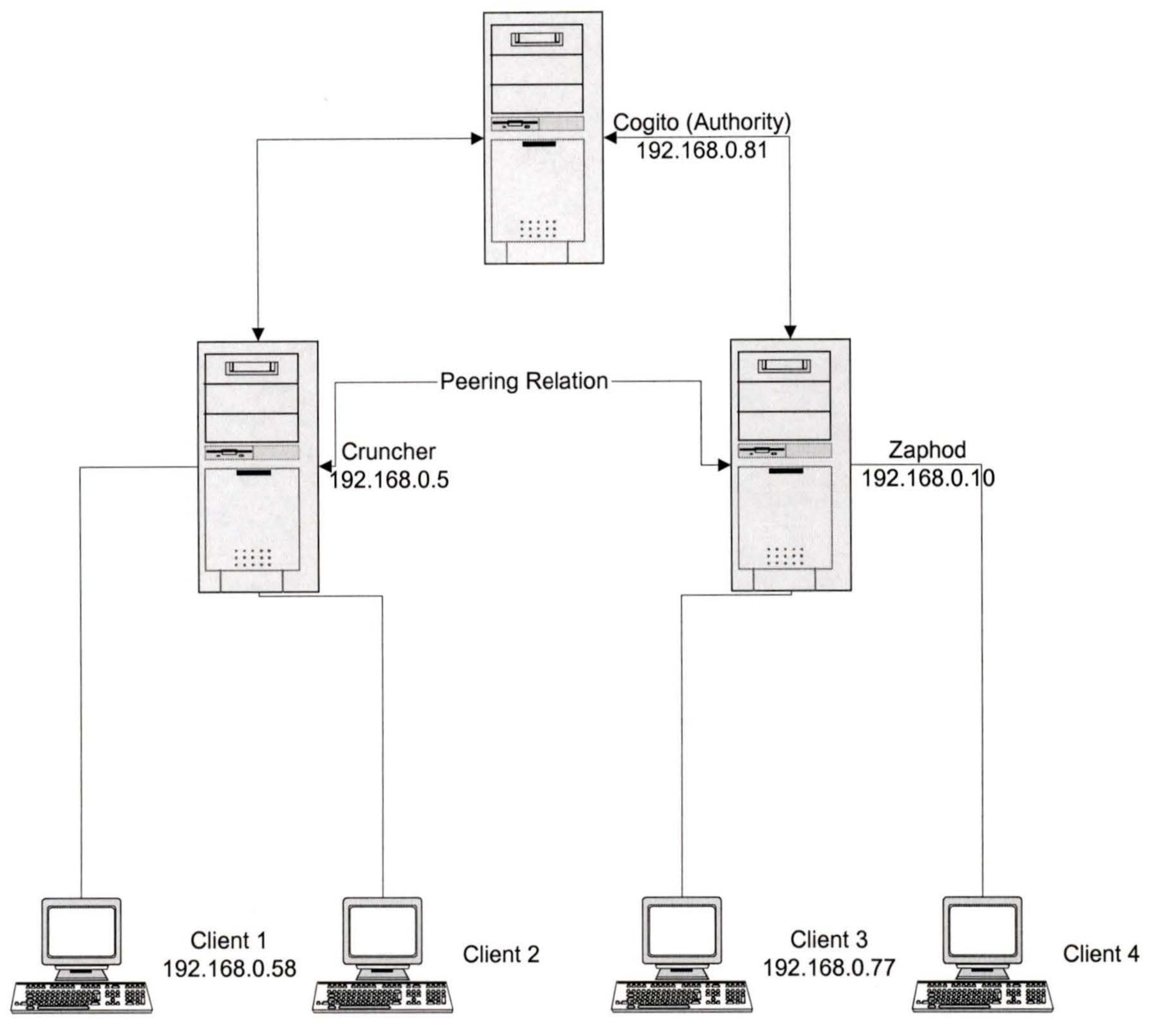


Figure 7.8) Architecture of URL Rewriting Testing network

Clients 3 and 4 were served images from Zaphod; clients 1 and 2 were served images from Cruncher. The clients can access any one of the three servers, and are then redirected to the appropriate server, see Figure 7.8. The redirection mechanism was not complex enough to support recursive redirection, however the functionality exists, as demonstrated in Figures 7.9, 7.10, 7.11 and 7.12. Figure 7.9 lists the JSP code given for the redirection, where Figure 7.10 is the interpretation of that code for Clients 1 and 2 of Figure 7.8. The

views of Client 1 is given in Figure 7.11, to compare with the view of the same JSP code for client 3 in Figure 7.12.

```
<html>
<body bgcolor="white">
<%-- include java files for requesting redirection address --%>
<%@ page import="java.io.*,java.net.Socket,java.net.SocketException"%>
<%-- Declare port number  --%>
<%! private int portNo = 5005; %>

<br>

<%-- Now we make a connection to the localhost with a well defined
        port number. Write the request.getRemoteAddr() to the port
        via a standard ObjectWriter, and read the response to the out --%>

<%!
private String redirectServer(String user) {
        try {
                //Socket bscGuy = new Socket(java.net.InetAddress.getLocalHost(), portNo);
                Socket  bscGuy  =  new  Socket(java.net.InetAddress.getByName("127.0.0.1"),
portNo);
                ObjectOutputStream                    output                =                   new
ObjectOutputStream(bscGuy.getOutputStream());
                ObjectInputStream                    input                =                   new
ObjectInputStream(bscGuy.getInputStream());
                output.writeObject(user);
                String response = (String)input.readObject();
                return response;
        } catch(OptionalDataException ode) {
                return "192.168.0.5"; // representing default choice
        } catch(StreamCorruptedException sce) {
                return "192.168.0.5";
        } catch(Exception e) {
                return "192.168.0.5";
        }
}
%>
<br><br>
<%
        String user = request.getRemoteAddr();
        String returnedString = redirectServer(user);
%>
<p>

Welcome!    You've    made    it    to    <img    src="http://<%=    returnedString
%>:8080/cdn/media/header.jpg"> where you can get the best service.
```

```
<br><br>
<br><br>
<center><h2>Image of the day!</h2>
<br>
<img src="http://<%= returnedString %>:8080/cdn/media/daily.jpg">
<br></center>
Thank you, come again
<br><br>
<p>
<br><br>
</font>
</body>
</html>
```

Figure 7.9) JSP Code used to redirect clients to different servers for embedded objects.

```
<html>
<body bgcolor="white">
<br>
<br><br>
<p>
Welcome! You've made it to <img src="http://192.168.0.5:8080/cdn/media/header.jpg">
where you can get the best service.

<br><br>
<br><br>
<center><h2>Image of the day!</h2>
<br>
<img src="http://192.168.0.5:8080/cdn/media/daily.jpg">
<br></center>
Thank you, come again
<br><br>
<p>
<br><br>
</font>
</body>
</html>
```

Figure 7.10) Redirection page HTML, the response to Figure 7.9 where the BSC algorithm has returned 192.168.0.5

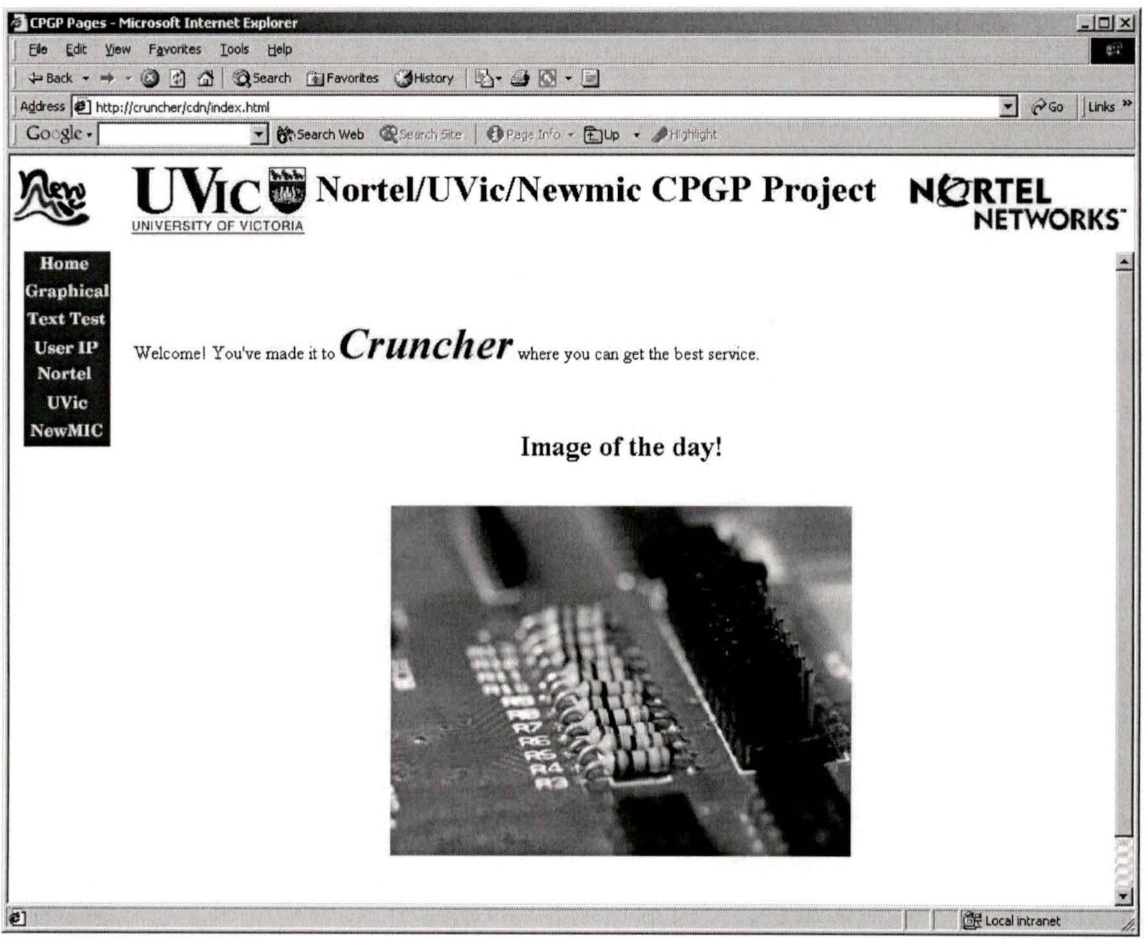


Figure 7.11) The rendered HTML from Figure 7.10 for clients 1 and 2 from Figure 7.8

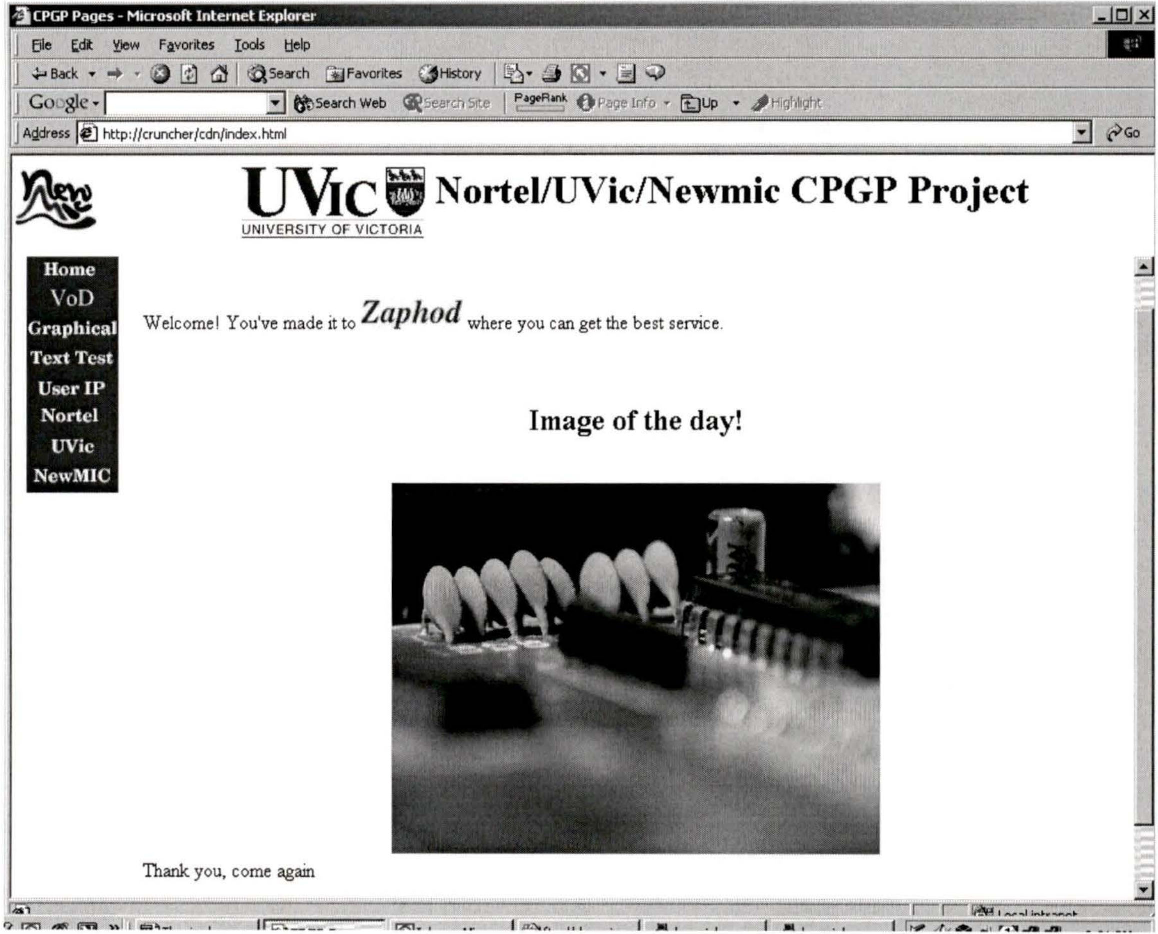


Figure 7.12) The rendered HTML generated from the Java Server Page outlined in Figure 7.9 for clients 3 and 4 from Figure 7.8

7.2.2 Setting up the Live Video On Demand Demo

We also applied the same principles used in the live redirection to constructing a live, streaming video on demand system. This system provided a potent example of the difference in quality achieved when the redirection system was utilized to route users to more local copies of video streams being requested. We began with the same architecture as used for the graphical redirection, as outlined in the previous example. The major difference was the inclusion of a video stream in the Java Server page. The video was encoded in MPEG-4, and formatted to be delivered by Darwin Streaming Media Server

©. Darwin is an Apple Public License program developed to deliver Quicktime© media streams live over the Internet. [51]

The same client/server architecture was employed to demonstrate the video streams, where one CPG represented and redirected to a Video server located across network boundaries at the University of Victoria, while the other redirected to a Video server located within the New Media Innovation Centre network in Vancouver. The video delivered from outside of the network was very low quality, with large amounts of jitter and delay. The same video delivered from within the network was smooth, and noticeably better quality.

7.2.3 Live Test Results

Having connected from multiple clients to multiple servers it is shown that the results can indeed be obtained from the CPGP system. On Demand URL Rewriting is a common method of request redirection, as well as being a visual display of the capabilities of the CDN suite.

As they are common technologies there are ample resources for developing and debugging java server pages, and writing html documents. The use of JSPs allows us to make use of these technologies, integrating servlet functionality (the logic) in with already developed HTML pages (the aesthetic). These pages may be developed separately, by professional web developers and utilize any other practices along side of the request direction.

This approach was further applied to a streaming video-on-demand system, where JSP pages were again used to embed video objects in HTML pages. The resulting video-on-demand system better showed the difference between streaming from more local caches than more remote caches.

Although it has been shown how the basic functionality can be applied, it is also useful to show that the inclusion of an options field, that could be used to store additional path information, perhaps to fill out the URL of an inline media inclusion, would be invaluable, as it would alleviate the need for each of the surrogates to have the same hierarchical structure for storing the content on their disks.

7.2.4 Live Test Conclusions

The live test has shown us not only how a Request Routing System may be constructed from 3 separate simulated content networks running CPGP, but also how that system may be used to meet client's needs. Making use of freely available and common services can assure us that the means to connect independent content networks is not merely a desirable option, but a realizable facility. The means to connect these networks can easily be based upon the CPGP and will support not only the requirements of hiding as much information as possible, but provides an extremely robust, scalable and functional environment. The visual nature of the live test is appropriate in its pedagogical aspects, and is sufficient to show how the CPGP service can be integrated with other redirection mechanisms, like DNS.

8. Conclusions

8.1 Major Contributions

A satisfactory solution to connecting Content Distribution Networks is a significant step towards scalable content delivery. This work has addressed how to direct requests to the best server for a unit of content, a major concern of the problem of content delivery. CPGP successfully coordinates the delivery of large amounts of content, in a manner that is scalable, reliable, and robust. The mechanisms developed are all meant to contribute to Content Network Interoperability. The development of a means of contention detection, for use in peer-to-peer communication establishment and persistence constitutes the most interesting aspects of this thesis.

The way in which the protocol identifies scope of interdependencies, and communicates this to its participants allows for a high degree of distribution of workload and communication burden. These connections allow each network to have a complete view of its own world, but filter out any unnecessary data. This allows each Content Network to be authoritative in its own domain, and will lead to a more reliable, and scalable networking world for all of them. It allows for deterministic distributed decision-making, an important and novel contribution of this thesis. The distributed nature of the information allows for the Best Server Choice Algorithm to be deterministic across multiple servers on multiple networks.

8.1.1 The CPGP Test Suite

The CPGP Test Suite was developed as a visual, and effective means of explaining and examining the CPGP protocol. It creates a visual environment in which a user can explore the interdependencies of Content Networks, and examine how they may communicate

and interoperate. The environment is cross-platform, and well supported, making the technology open. This increases the possible reach of CPGP, and applies the concepts evolving from the scope of CDNs to other areas.

8.1.2 The Live Test: On Demand URL Rewriting

The On-Demand URL Rewriting live test showed the validity of our approach to contention detection. Users were successfully redirected to more appropriate servers depending on their location. This resulted in faster and better service for those customers. This example also showed the versatility of application layer redirection, and it's application to content distribution.

8.2 Future Work

As with any research, the questions raised are often as interesting as the question being studied. There is much work to be done in Content Distribution Systems, including performance evaluations, methods of collecting network metrics, privacy, data integrity, accounting, digital rights management (DRM) issues, and automated distribution methods. Trials that support these issues could include the use of CPGP as their communication mechanism, which could be extended and applied to other knowledge sharing distributed systems.

9. References

[1] B. Cain, O. Spatscheck, M. May, A. Barbir, "Request-Routing Requirements for Content Internetworking" IETF Internet Draft draft-ietf-cain-request-routing-req-03.txt, http://www.ietf.org/internet-drafts/draft-cain-request-routing-req-03.txt

[2] M. Green, B.Cain, "CDN Peering Architectural Overview" IETF draft draft-green-cdnp-gen-arch-01.txt, http://www.content-peering.org/draft-green-cdnp-gen-arch-01.html;

[3] A. Barbir, B. Cain, et al "Known CN Request-Routing Mechanisms" IETF draft draft-cain-cdnp-known-reqeust-routing-04.txt, http://www.ietf.org/internet-drafts/draft-cain-cdnp-known-request-routing-04.txt

[4] M. Day, B. Cain, et al "A Model for Content Internetworking", IETF draft draft-day-cdnp-model-09.txt;. http://www.ietf.org/internet-drafts/draft-day-cdnp-model-09.txt

[5] G. J. Holzmann, Design and Validation of Computer Protocols, Prentice Hall Software Series; 1991

[6] F. Kuo et al., Protocols and Techniques for Data Communication Networks, Prentice Hall series in Computer Applications in Electrical Engineering, 1981

[7] C. Scheideler, Universal Routing Strategies for Interconnection Networks, Springer, 1998

[8] R. Sharp, Principles of Protocol Design, Prentice Hall, 1994

[9] IETF Secretariat, "IETF Overview" http://www.ietf.org/overview.html, September 2001

[10] L. Fan, P Cao, J, Almeida, A, Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", IEEE Transactions On Networking Vol. 8. No 3., June 2000, pp 281-293

[11] Lampson, Butler, Srinivasan, Venkatachary, Varghese, "IP Lookups Using Multiway and Multicolumn Search", IEEE Transactions On Networking Vol. 7. No 3, June 1999, pp 324-334

[12] B. Krupczak, K. Calvert, M. Ammar, "Increasing the Portability and Re-Usability of Protocol Code", IEEE Transactions On Networking Vol. 5. No 4, August 1999, pp 445-459

[13] "Routing On Longest Matching Prefixes", IEEE Transactions On Networking, Vol. 4. No 1, February 1996, pp 86-97

[14] P. Deolasee et Al., "Adaptive Push-Pull: Disseminating Dynamic Web Data", 10th Annual WWW Conference Proceedings, May 2001, pp 265-274

[15] S. Gruber et Al., "Protocol Considerations for a prefix-caching proxy for multimedia streams", Computer Networks 33, 2000, pp 657-668

[16] P. Rodriguez, "SPREAD: scalable platform for reliable and efficient automated distribution", Computer Networks 33, 2000, pp 33-49

[17] PC Webopaedia, "Domain Name Service", http://www.scit.wlv.ac.uk/~jphb/comms/dns.html, November 2001

[18] D. Vrsalovic, "Intelligent, Stupid, and Really Smart Networking", ACM Mixed Media, May 1998, pp 44-47

[19] J. Dilly, M. Arlitt, "Improving Proxy Cache Performance: Analysis of Three Replacement Policies", IEEE Internet Computing, December 1999, pp 44-50

[20] Nortel Networks Product Brief, Nortel Networks: Alteon Web Switching Module, 2001

[21] C. Kalmanek, S. Srinivasan, W. Marshall et Al.; "Xunet 2: Lessons from an Early Wide-Area ATM Testbed", IEEE/ACM Transactions on Networking, Vol. 5, No. 1, Feb 1997, pp 40-55

[22] Booch, Rumbaugh, Jacobson, The Unified Modeling Language User Guide, Addison-Wesley; 1999

[23] Yrekhter, T.J. Watson, T. Li, et al.; "Network RFC 1771: a border Gateway Protocol 4 (BGP-4)", IETF RFC 1771, March 1995

[24] Cisco Systems white paper, "Border Gateway Protocol", Internetworking Technology Overview, June 1999

[25] P. Mockapetris, "Domain Names – Concepts and Facilities", IETF RFC 1034, November 1987

[26] BBC News article, "Web Attacks on the Rise", BBC News http://news.bbc.co.uk/hi/english/sci/tech/newsid_1602000/1602493.stm, 16 October 2001;

[27] M. Hall, Core Servlets and Java Server Pages, Sun Microsystems Inc., August 2000

[28] R. Moss, et Al., "Information, Communications & Media (ICM) Panel: ITEC Technologies Document", Office of Science and Technology, Department of Trade and Industry UK, September 2000

[29] M. Raynal, T.Tronel, "Group Membership Failure Detection", British Computer Society, September 1999, pp 95-102

[30] X. Chen, L.E. Moser, P.M.. Melliar-Smith, "Totally Ordered Gigabit Multicasting", Distributed Systems Engineering 4, 1997

[31] Ahamad, M., Raynal, M., and Thia-Kime, G., "An Adaptive Protocol for Implementing Causally Consistent Distributed Services". Proc. of the 18th IEEE Int'l Conf. on Distributed Computing Systems (ICDCS-18), 1998, pp.86-93.

[32] P. Krishnan, D. Raz, "The Cache Locations Problem", IEEE Transactions on Networking Vol. 8, No 5., October 2000, pp 568-582

[33] M. van Sinderen, L. Pires, C. Vissers, "Protocol Design and Implementation Using Formal Methods", The Computer Journal 35, October 1992, pp 478-491

[34] Y. De Serres, L. Hegarty, "Value-Added Services in the Converged Network", IEEE Communications Magazine, September 2001, pp 146-154

[35] Q. Cai, "An Object-Oriented Model for Intelligent Networks", ACM, 1992

[36] R.Fielding, "Hypertext Transfer Protocol" RFC 2616, IETF Network Working Group RFC, 1999

[37] D. Clark, "The Design Philosophy of the DARPA Internet Protocols", ACM Symposium proceedings on Communications architectures and Protocols, 1998, pp 106-114

[38] C. Labovitz, G. Malan, J. Farnam, "Internet Routing Instability", Proceedings of ACM SIGCOMM '97, 1997, pp 115-126

[39] R. Caceres, "Characteristics of Wide-Area TCP/IP Conversations", ACM, 1991, pp101-112

[40] C. M. Kehoe, J. Pitkow, "Surveying the Territory: GVU's Five WWW User Surveys", The World Wide Web Journal, Vol. 1, no. 3, 1996, pp 77-84

[41] C. M. Kehoe, J. Pitkow, "Emerging Trends in the WWW User Population", Communications of the ACM, Vol. 39, no. 6, 1996

[42] R.Wooster, M.Abrams, "Proxy Caching that Estimates Page Load Delays", http://vtopus.cs.vt.edu/~chitra/docs/www6r/, 1996

[43] G. Apostolopolous, "Design Implementation and Performance of a Content Based Switch", Technical Report, IBM T. J. Watson Research Centre; 1999

[44] Nortel Whitepaper, "Optimizing ISP Networks and Services with DNS Redirection", Nortel Networks Alteon Web Systems; August 1999

[45] T. Berners-Lee, L. Masinter, M. McCahill, Uniform Resource Locator (URL), IETF RFC 1738, May 1994

[46] M. Hayden, "The Ensemble System", Cornell University Technical Report, TR98-1662, January 1998

[47] R. van Renesse, K. Birman, M. Hayden, A. Vaysurd, D. Karr, "Building Adaptive Systems Using Ensemble", Cornell University Technical Report, TR97-1638, July 1997

[48] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", IETF RFC 1889, IETF, January 1996

[49] K. Hubbard, M. Kosters, D. Conrad, D. Karrenberg, J. Postel, "Internet Registry IP Allocation Guidelines", IETF RFC 2050, IETF, November 1996

[50] T. Devlin, BC Net Representative, Personal Communication, January 2002

[51] Apple Developer Site "Apple – Open Source - Darwin", http://www.opensource.apple.com/projects/darwin/, April 2002

10. Appendices

10.1 Glossary of Terms

BGP p.31– Border Gateway Protocol, used to communicate routes for packets to traverse across multiple networks

BSC p.72– the algorithm run to determine the best server choice for a given client

CDI p.13 – Content Distribution Internetworking: the industry acronym for Interconnection of Content Networks

CDN p.11 – Content Distribution Network: also known as Content Network

Child p.25 – a CPG representative of a specific CDN that has connected to the authoritative server

CPG p.20 – Content Peering Gateway: The entry-point to a CDN from the outside, also representative CPG

CPGP p.20 – The protocol used by CPG's to communicate

DNS p.13 – Domain Name Service: The way in which a readable URL is translated to an IP address

Footprint p.21 – The IP prefix representing a specific service area

HTML p.6 – Hypertext Markup Language: Syntax used to format web pages

HTTP p.6 – Hypertext Transfer Protocol: Protocol used to retrieve web pages

IETF p.13 – Internet Engineering Task Force: Volunteer group that regulates Internet protocols

IP v4 / IP v6 p.36 – Internet Protocol, version 4 or 6: Protocols used to address computers on the Internet

Peer p.26– two CPG's that service at least one common IP Range are peers for those common IP Ranges

Root p.23 – The Authoritative Server

Surrogate p.13 – a server hosting content on behalf of a content network

URL p.13 – Uniform Resource Locator: A string representing the protocol, server, and location on the server, of a specific unit of content

TTL p.15, 39 – Time To Live: used to designate how long a specific piece of information is assumed to be correct

10.2 CPG Code

10.2.1 bscListener.java

```
import java.net.*;
import java.util.*;
import java.io.*;

public class bscListener extends Thread {

    private cpg parent;
    private ServerSocket theSocket;
    private boolean root;
    private ObjectInputStream StringInput = null;
    private ObjectOutputStream StringOutput = null;

    public bscListener(cpg parent, int port) {
        this.parent = parent;
        try {
            //create the socket
            theSocket = new ServerSocket(port, 5);
        } catch (IOException e) {
            System.err.println("Error creating bscListner");
            System.exit(1);
        }
    }

/** public void run()
  * accepts sockets, if this is the root then the newConnection function
  * should simply add children
  * If they are CPG's then these are all peers and should be added as such
  **/
    public void run() {
        Socket newSocket = null;
        String ipToResolve = null;
        String ipOut;

        while(true) {
            // handle one client at a time. This belies multithreading
            // however as the critical sections are all protected
            // this behaviour allows us to queue requests here
            try {

                newSocket = theSocket.accept();
```

```
            // we now have a socket, accept the IP as a string
            StringInput =
               new ObjectInputStream(new
BufferedInputStream(newSocket.getInputStream()));
            StringOutput =
               new ObjectOutputStream(newSocket.getOutputStream());

            // find out what we need to resolve
            ipToResolve = (String)StringInput.readObject();

            ipOut = parent.BSC(new ip(ipToResolve));

            // We now have our answer, return it to the requesting agent
            StringOutput.writeObject(ipOut);
            StringOutput.flush();

            // string has been returned, close the socket, null it

            StringInput.close();
            StringOutput.close();
            newSocket.close();
            newSocket = null;

         } catch(ClassNotFoundException cnfe) {
            System.err.println("error trying to read string");
         } catch(IOException e) {
            // as there are a lot of things that could go wrong here
            // it is a very good spot to do some error handling,
            // we just don't
            System.err.println("error with accepting socket");
         }
      } // while
   } // run

} //class socketaccepter
```

10.2.2 cpg.java

```
import java.lang.Thread;
import java.lang.Long;
import java.lang.Math;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import java.net.InetAddress;
//import java.util.Vector

/** CPG.JAVA is the main for a CPG. It collects the matrix, supports the
```

```
 * surrogate data collection as well as the protocol implementation
 * the protocol is directly implemented by the CPG, which owns the
 * metric matrix.
 *
 * Written: Douglas Johnson
 *      May 19 2001
 **/


public class cpg extends Thread {

private cpggui mygui = null;
public cpgexchange exchanger = null;
protected matrix metricmatrix = null;
private Object protocolHandler = null;
private boolean parent = false;
private int port;
private InetAddress reportTo;
private short[] myIP = null;
private bscListener myBSC = null;

   public cpg(String name, int port, boolean parent){
      // IMPORTANT metricmatrix must be instantiated first.
      metricmatrix = new matrix();
      try {
      this.port = port;
      exchanger = new cpgexchange(port);
      exchanger.start();
      myBSC = new bscListener(this, 5005);
      myBSC.start();

      } catch (NullPointerException e) {
         System.err.println("error making exchanger. port in use");
         System.exit(1);
      }
      this.parent = parent;
      mygui = new cpggui(this, name);

      String ipString = JOptionPane.showInputDialog(new JPanel(), "Please enter this
CPG's IP address", "IP Input", JOptionPane.QUESTION_MESSAGE);
      myIP = (ipString.length() > 0) ? ip.parseIP(ipString) : new short[] { 0, 0, 0, 0};
   } // cpg constructor


   public static void main(String args[]) {
      cpg me = null;
```

```
        if (args.length == 3) {
            me = new cpg(args[0], Integer.parseInt(args[2]), ((args[1].charAt(0) == 'y') ?
true : false));
            //me.parent = (args[1].charAt(0) == 'y') ? true : false;
        } else {
            me = new cpg("CPG", Integer.parseInt("1500"), false);
        }

            me.run();
    }

    public String BSC(ip clientIP) {
        datamatrix matching;
        long mindelay = Long.MAX_VALUE;
        int minrow = 0;
        // this will get the processing data, then find the rows.
        matching = (metricmatrix.getp()).findMatches(clientIP, true);
        System.out.println("client ip is: " + clientIP);
        System.out.println("Got the matching matrix: " + matching.size());
        for(int i = 0; i < matching.size(); i++) {
            if(((row)matching.elementAt(i)).delay < mindelay) {
                mindelay = ((row)matching.elementAt(i)).delay;
                minrow = i;
            }
            System.out.println("entry: " + ((row)matching.elementAt(i)).footprint);
        }

        if(mindelay == Long.MAX_VALUE) {
            System.out.println("default choice: 127.0.0.2");
            return "127.0.0.2";
        }

        // the minimum row is the best match we have

        System.out.println("minimum delay: " + mindelay);
        System.out.println("destination ip: " +
((row)matching.elementAt(minrow)).destinationIP);

        return (((row)matching.elementAt(minrow)).destinationIP).toString();
    }

    public void run() {
        mygui.pack();
        mygui.setVisible(true);
        while(true) {
                while(exchanger.messageAvailable()) {
```

```
                //Handle incoming message
                recieveMessage(exchanger.getMessage());
            }
            //exchanger.processLoop();
            yield();
        }
    }

public datamatrix findContenders(row newRow, datamatrix inputmatrix) {
/* it would be more efficient to store this information
   sorted by who this is to be advertised to, however
   as we are concerned with the protocol, rather than
   the efficiency of the test suite, this is left
   for further revisions

   Currently this takes in the row that will be added to
   the advertised matrix, and checks the iData for
   all representative rows (with little effieciency)
   as this will probably not occur too frequently.

*/
   int i;
   row aRow;
   datamatrix matching;
   datamatrix returnmatrix = new datamatrix();

   // first search for rows in iData that represent this
   // row, OR that this row represents
   // we do this by appending as many .255.255's to the
   // ip prefix to make it complete, and extracting the iData rows
   // then we search the matching rows to see which ones
   // need to advertised


   System.out.println("looking for ip's which match: " + newRow.footprint);

   // this will get the idata, then find the rows.
   matching = inputmatrix.findMatches(newRow.footprint, true);

   // now remove the useless rows

   for(int j = 0; j < matching.size(); j++) {
      // do not report contending elements from this cpg
      aRow = (row)((row)matching.elementAt(j)).clone();
      if((aRow.reportTo == newRow.reportTo)
         && (aRow.port == newRow.port)) continue;
```

```
            returnmatrix.add(aRow);
/**
            //if(aRow.representative(newRow)) {
               // for each of these rows we must add it to the
               // matrix, with the ID of the recipient in the
               // reporting field.
               // update aRow to have the new id in it
               returnmatrix.add(aRow);
            //} // representative of the row
**/

         }

         for(i = 0; i < returnmatrix.size(); i++)
            System.out.println("Found: " + (row)matching.elementAt(i));

         return returnmatrix;

      } // find contending rows from an input matrix



      public void removeRowInternal(row oldRow) {
         cpgMessage newMessage;
         row aRow;
         metricmatrix.removei(oldRow);
         mygui.updatelist('I');

         // This is also an appropriate place to remove rows from
         // the advertisedMatrix, and the processing matrix.

         if(!parent) {
            aRow = (row)oldRow.clone();
            aRow.destinationIP = new ip(myIP);
            //removeRowAd(aRow);

            // also create a new message to send to parent
            newMessage = new cpgMessage(cpgMessage.REMOVE);
            newMessage.addData(aRow);
            exchanger.sendToParent(newMessage);
         }

         removeRowProcessing(oldRow);
```

```
    return;
}

public void addRowInternal(row newRow) {
    cpgMessage newMessage;
    row pRow, aRow;

    try {
        newRow.reportTo = InetAddress.getLocalHost();
    } catch (java.net.UnknownHostException uhe) {
        System.err.println("Error with Network connectivitiy");
    }

    newRow.port = this.port;
        //addRowAd(aRow); // no longer add Rows to advertisement generally.


    metricmatrix.addi(newRow);
    mygui.updatelist('I');
    if(!parent) {
        aRow = (row)newRow.clone();
        aRow.reported = true;
        aRow.destinationIP = new ip(myIP);

        newMessage = new cpgMessage(cpgMessage.NEW);
        newMessage.addData(aRow);
        exchanger.sendToParent(newMessage);
    }

    addRowProcessing(newRow);

    return;
}


public void addRowAd(row newRow, int recievingIndex) {

datamatrix contending = null;
row aRow;

// find matching

contending = findContenders(newRow, metricmatrix.geti());

// now remove the useless rows
```

```
for(int j = 0; j < contending.size(); j++) {
   // for each of these rows we must add it to the
   // matrix, with the ID of the recipient in the
   // reporting field.
   aRow = (row)((row)contending.elementAt(j)).clone();
   aRow.destinationIP = new ip(myIP);
   aRow.reported = false;
   // update aRow to have the new id in it
   aRow.reportIndex = recievingIndex;
   metricmatrix.adda(aRow);
} // representative of the row

mygui.updatelist('A');


} //addRowAd


public void removeRowAd(int sendingIndex) {
   // this will remove all the rows from
   // the advertise matrix that send to this index

   for(int i = 0; i < metricmatrix.geta().size(); i++) {
      // MODIFY HERE!! REMOVE ROWS FROM A
   }

   return;
}

public void addRowExternal(row newRow) {
   row eRow = (row)newRow.clone();
   addRowProcessing(newRow);
   metricmatrix.adde(eRow);
   mygui.updatelist('E');
}

public void removeRowExternal(row oldRow) {
   removeRowProcessing(oldRow);
   metricmatrix.removee(oldRow);
   mygui.updatelist('E');
   if(!parent) {
      //removeRowAd(oldRow);
   }
}

public void addRowProcessing(row newRow) {
```

```
    row pRow = (row)newRow.clone();
    metricmatrix.addp(pRow);
    mygui.updatelist('P');
}

public void removeRowProcessing(row oldRow) {
    metricmatrix.removep(oldRow);
    mygui.updatelist('P');
}

public void removeRowsFromCPG() {


} // remove rows from cpg

public void removeRowsToCPG() {


} // removeRowsToCPG

public void newRoot(String childIP, int port) {
    int rootID = exchanger.connectToParent(childIP, port);
    cpgMessage hi = new cpgMessage(cpgMessage.HELLO);
    // we now have the new child, send it a hello packet
    if(rootID != -1) {
        exchanger.sendToParent(hi);
    } else {
        System.err.println("invalid child");
    }
}

public void recieveMessage(cpgMessage m) {
    System.out.println("recieving Message:" + m);
    switch (m.Type) {
        case cpgMessage.HELLO:  HELLO(m);
                              return;
        case cpgMessage.RESET:  RESET(m);
                              return;
        case cpgMessage.NEW:     NEW(m);
                              return;
        case cpgMessage.UPDATE: UPDATE(m);
                              return;
        case cpgMessage.CONTEND: CONTEND(m);
                              return;
        case cpgMessage.REMOVE: REMOVE(m);
                              return;
```

```
        case cpgMessage.TERM:   TERM(m);
                                return;
        case cpgMessage.KA:     return;
        case cpgMessage.NULL:   return;
        default:
    }//switch
    return;

} // recieveMessage

private void HELLO(cpgMessage m) {

    // reset messages are also used as a confirmation of an initial connection
    // the connection itself is made independently and does not necessarily entail
    // any exchange of information.
    // reset the connection
    if(m.recievingID != -1)
    exchanger.sendToPeer(new cpgMessage(cpgMessage.RESET), m.recievingID);

} //HELLO

/** private void NEW handles all new incoming rows.
It is assumed that the exterior CPG indicates
that they are providing us with all NEW information
in THIS format.
**/
private void NEW (cpgMessage m) {

    row workingRow = null;
    datamatrix contending;
    cpgMessage contention = new cpgMessage(cpgMessage.CONTEND);
    cpgMessage reportNew;

    try {
        if(parent) {
            datamatrix inData = (datamatrix)m.getData();
            // first get contenders
            for(int i = 0; i < inData.size(); i++) {
                workingRow = (row)inData.get(i);

                // check for contending row

                contending = findContenders(workingRow, metricmatrix.gete());

                if(contending.size() == 0) continue;
                for(int j = 0; j < contending.size(); j++) {
```

```
                    System.out.println("Checking for contender: " +
(row)contending.elementAt(j));
                    contention.addData((row)contending.elementAt(j));
                }

            } // put all rows in our matrix

            // now add the rows (if we don't do it this way
            // we get the rows we just put in
            for(int i = 0; i < inData.size(); i++) {

                workingRow = (row)inData.get(i);
                addRowExternal(workingRow);

            }

            if(((datamatrix)contention.getData()).size() > 0)
                exchanger.sendToPeer(contention, m.recievingID);

        } // if parent

        else { // we're getting a new row from a peer
            contending = new datamatrix();
            datamatrix inData = (datamatrix)m.getData();
            for(int i = 0; i < inData.size(); i++) {
                workingRow = (row)inData.get(i);
                workingRow.reportIndex = m.recievingID;
                //workingRow.reported = true;
                addRowExternal(workingRow);
                addRowAd(workingRow, m.recievingID);
            } // for


            // now our aData is uptodate and eDatnewRow.footprint
            // find the rows in aData that need to be
            // reported, and report them.
            // we narrow this down to be simply those
            // to this host

            datamatrix workinglist = metricmatrix.geta();
            workinglist = workinglist.findMatches(m.recievingID);
            for(int i = 0; i < workinglist.size(); i++) {
                workingRow = (row)workinglist.get(i);
                if(workingRow.reported == false) {
                    workingRow.reported = true;
                    contending.add(workingRow);
```

```
                }
            }

            //now contending has a list of the
            // new rows to be reported to the other
            // cpg if there are rows, we send them
            if(contending.size() > 0) {
                reportNew = new cpgMessage(cpgMessage.NEW);
                reportNew.addData(contending);
                exchanger.sendToPeer(reportNew, m.recievingID);
            }

        } // else
    } catch (Exception e) {
        System.err.println("data exception");
        e.printStackTrace();
    }


} // NEW

/** private void UPDATE handles updates to existing
    rows that are stored in our eData matrix.
    They are simple look-ups/replacements
    **/
    private void UPDATE(cpgMessage m) {
        row workingRow = null;
        try {
                datamatrix inData = (datamatrix)m.getData();
                for(int i = 0; i < inData.size(); i++) {
                    workingRow = (row)inData.get(i);
                    workingRow.reported = true;
                    addRowExternal(workingRow);
                } // put all rows in our matrix
        } catch (Exception e) {
            System.err.println("data exception");
            e.printStackTrace();
        }


    } //UPDATE

    private void CONTEND (cpgMessage m) {

        int index;
        int hostIndex;
```

```
datamatrix inData, outData, cData, tData;
InetAddress remoteHost;
int remotePort;
row workingRow = null;

try {
   inData = (datamatrix)m.getData();
   while(inData.size() > 0) {
   /**
   For each REMOTEHOST
     get the INDEX of the remote host
     foreach ROW
        add that row to the eData
        search for contending rows in the iData
           foreach contending row found add to aData
            w/ reference to the remote host (INDEX)

     send HELLO message to the remote host
     this will preciptate a RESET message, where we then send a complete
     list of rows found in our aData for that index.
   **/
      outData = new datamatrix();
      tData = new datamatrix();
      workingRow = (row)inData.remove(0);
      // extract each row by remote host name
      // presumably the rows have the ip of the remote host

      index = 0;
      remoteHost = workingRow.reportTo;
      remotePort = workingRow.port;
      hostIndex = exchanger.newPeer(remoteHost, remotePort);
      outData.add(workingRow);
      // we extract all the rows in contention for the first host
      while(inData.size() > 0 && index < inData.size()) {
         if(((row)inData.get(index)).reportTo == remoteHost) {
            outData.add(inData.remove(index));
         } else {
            index++;
         }
      } // inner while loop
      // the outData matrix now has the rows for a given peer

      // now each of these rows should be checked for local contenders
      // for each contender found, add it to another datamatrix
      // then for each row in that datamatrix, set the reporting index
      // and add it to the aData
```

```
/**
            for(int i = 0; i < outData.size(); i++) {
                cData = findContenders(outData.elementAt(i), matrix.geti());
                // this seemingly uneccessary step is to keep the size
                // of data being actively added to aData minimized.
                // Removing the redundancy here will save processing later
                for (int j = 0; j < cData.size(); j++) {
                    tData.add(cData.elementAt(j));
                } // for all cData
            } // for all outData

            for (int k = 0; k < tData.size; k++) {
                (row)tData.elementAt(k).reportIndex = hostIndex;
                (row)tData.elementAt(k).reported = false;
                aData.add(tData.elementAt(k));
            } // adding to the aData
**/

            // Add them all to the addMatrix
            // Add them all to the eData matrix
            for(int i = 0; i < outData.size(); i++) {
                workingRow = (row)outData.elementAt(i);
                System.out.println("adding to A data: " + workingRow);
                workingRow.reportIndex = hostIndex;
                workingRow.reported = false;
                addRowAd(workingRow, hostIndex);
                addRowExternal(workingRow);
            } // for

            cpgMessage outgoing = new cpgMessage(cpgMessage.HELLO);

            // FINALLY send the HELLO packet
            exchanger.sendToPeer(outgoing, hostIndex);


        } // outer while

    } catch (Exception e) {
        System.err.println("data exception");
        e.printStackTrace();
    } // TryCatch


} // CONTEND

private void REMOVE(cpgMessage m) {
```

```
    row workingRow = null;
    try {
        datamatrix inData = (datamatrix)m.getData();
        for(int i = 0; i < inData.size(); i++) {
            workingRow = (row)inData.get(i);
            removeRowExternal(workingRow);
        } // put all rows in our matrix

    } catch (Exception e) {
        System.err.println("data exception");
        e.printStackTrace();
    }

    // relay this information to our parent
    //exchanger.sendToParent(m);
    // NO LONGER relay as we no longer have "transitivity"

} //UPDATE

private void RESET(cpgMessage m) {


    datamatrix tData = new datamatrix();
    datamatrix nData = new datamatrix();
    datamatrix wData = null;
    int rowIndex = m.recievingID;
    row aRow;

    if(m.recievingID == cpgexchange.maxPeers) {

        cpgMessage updateMessage = new cpgMessage(cpgMessage.NEW);
        // behavior changes if it is from our root
        // if so then we must send it all of our
        // internal rows, slightly modified for
        // black-box mentality
        wData = metricmatrix.geti();

        for(int i = 0; i < wData.size(); i++) {
            aRow = (row)((row)wData.get(i)).clone();
            aRow.destinationIP = new ip(myIP);
            tData.add(aRow);
        }

        updateMessage.addData(tData);

        exchanger.sendToPeer(updateMessage, m.recievingID);
```

```
    } // if it was our parent

    //THIS IS WHAT HAPPENS WHEN A PEER SENDS RESET
    else {
        // The message was obviously from A peer, so we will send all
        // rows in aData that go to this peer
        // get all messages for this InetAddress/port


        wData = metricmatrix.geta();
        for (int j = 0; j < wData.size(); j++) {
            if(((row)wData.elementAt(j)).reportIndex == rowIndex) {

                aRow = (row)((row)wData.get(j)).clone();
                aRow.destinationIP = new ip(myIP);
                if(aRow.reported == false) {
                    ((row)wData.elementAt(j)).reported = true;
                    nData.add(aRow);
                } else {
                    tData.add(aRow);
                }
            } // if
        } // finding the right ones...

        if(nData.size() > 0) {
            // we're makinga new new message
            cpgMessage newMessage = new cpgMessage(cpgMessage.NEW);
            newMessage.addData(nData);
            exchanger.sendToPeer(newMessage, rowIndex);
        } // nData

        if(tData.size() > 0) {
            cpgMessage updateMessage = new cpgMessage(cpgMessage.UPDATE);
            updateMessage.addData(tData);
            exchanger.sendToPeer(updateMessage, rowIndex);
        }

    }


    return;


} //reset message
```

```
    private void TERM(cpgMessage m) {
        // first remove all the rows FROM this CPG
        removeRowsFromCPG(); // send data on which CPG

        // next remove any rows going TO this CPG
        removeRowsToCPG(); // send data on which CPG

        // finally release the resources on the exchanger side
        exchanger.removePeer(m.recievingID);

        return;
    } // TERM message


}
```

10.2.3 cpgexchanger.java

```
import java.net.*;
import java.io.*;
import java.util.Vector;

/** Class cpgexchange
  * cpgex is the class used to exchange messages between CPGs
  * it supports the simple functions of send and recieve
  * it provides a buffer to hold incoming messages, and
  * reports those messages.
  * Each CPG can have at MOST 1 authority, but multiple peers
  * the cpgex for each cpg is a seperate thread in the program
  **/

public class cpgexchange extends Thread {

    public static final int maxPeers = 255;
    private int authority = maxPeers;
    private boolean root = false;

    socketHandler[] connections = null;
    Vector incomingMessages = null;
    socketaccepter myaccepter = null;
    //keepalive mykeepalive = null;


/** cpgex is the constructor, as much info as can be passed is passed
  * here we must also create a connection to our authority (if we're not one
  **/
    public cpgexchange(int myPort) {
```

```
        connections = new socketHandler[maxPeers + 1];
        this.root = root;
        incomingMessages = new Vector();

        try {
            myaccepter = new socketaccepter(new ServerSocket(myPort, 3), this, root);
            // a buffer of 3 extra sockets can be held while processing
        } catch (IOException sio) {System.out.println("couldn't bind to port " + myPort);}

        myaccepter.start();
    }

/** keepSocketsAlive ensures that we do not timeout with our peers.
  * by sending a keepalive packet every 29 seconds
  **/

    public synchronized void keepSocketsAlive() {
        cpgMessage m = new cpgMessage(cpgMessage.KA);
        for(int i = 0; i < maxPeers; i++) {
            if(connections[i] != null)
                (connections[i]).send(m);
        } //for
    }

/** newConnection(Socket inSocket)
  * newConnections accepts incoming socket requests from
  * other CPG's (Peers) for communication.
  * If we wish to make a connection (push) then we must
  * use the "newPeer" function.
  **/
    public synchronized int newConnection(Socket inSocket) {
        int pos = 0;
        while(pos < maxPeers && connections[pos] != null) pos++;
        if (pos == maxPeers) return -1;
        try {
            connections[pos] = new socketHandler(inSocket, pos, this);
        } catch (IOException e) {}
        connections[pos].start();
        return pos;
    }

/** newPeer(String Child, int port)
  * when we get CONTENTION for rows we are bound to make the connection
  * to the CONTENDERS for the advertised footprint.
  * Thus the latecomer is the one that initiates contact.
```

```
 **/
  public synchronized int newPeer(String child, int port) {

      InetAddress childInet = null;
      // FIRST see if we already have a connection to
      // this CPG
      try {
         childInet = InetAddress.getByName(child);
      } catch (UnknownHostException ihe) {
         System.err.println("Cannot connect to peer: " + child);
         return -1;
      }
      return newPeer(childInet, port);
  }


/** newPeer(InetAddress Child, int port)
 * when we get CONTENTION for rows we are bound to make the connection
 * to the CONTENDERS for the advertised footprint.
 * Thus the latecomer is the one that initiates contact.
 **/
  public synchronized int newPeer(InetAddress childInet, int port) {
  System.out.println("child inet " + childInet);
  System.out.println("port " + port);

      for(int i = 0; i < maxPeers; i++) {
         if(connections[i] == null) continue;
         if(connections[i].compareTo(childInet, port)) return i;
      }

      //first find the position to insert this child at
      int pos = 0;
      while(pos < maxPeers && connections[pos] != null) pos++;
      if (pos == maxPeers) return -1;

      try {
      //  childSocket = new Socket(child, port);
         connections[pos] = new socketHandler(childInet, port, pos, this);
         connections[pos].start();
         return pos;
      } catch (IOException e) {
         System.out.println("error making socket");
         e.printStackTrace();
         return -1;
      }
```

```
    }
/** sendToParent(cpgMessage)
  * sends to the root level
  **/

    public synchronized boolean sendToParent(cpgMessage m) {

        if(connections[maxPeers] == null) return false;
        connections[maxPeers].send(m);
        return true;

    }// sendToParent


/** public connectToParent(String rootName, int portNo)
  * this function connects to the root level
  **/
    public synchronized int connectToParent(String rootName, int port) {
        Socket authoritySocket;
        try {
            //authoritySocket = new Socket(rootName, port);

            connections[maxPeers] = new
socketHandler(InetAddress.getByName(rootName), port, maxPeers, this);
            connections[maxPeers].start();
        } catch (IOException e) {
            System.out.println("error making socket");
            e.printStackTrace();
        }
        return maxPeers;
    }  // connectToParent


    public synchronized boolean sendToPeer(cpgMessage m, int i) {
        try {
            System.out.println("writing to peer " + i + " a message: " + m);
            if(connections[i] == null) return false;
            connections[i].send(m);
        } catch (ArrayIndexOutOfBoundsException aioobe) {
            System.out.println("out of bounds sendingToPeer");
            return false;
        }
        return true;
    } // sendToPeer
```

```
    public synchronized boolean sendToPeer(cpgMessage m, InetAddress recipient, int
port) {
        // used for sending to an unknown index, if there is no such index we
        // create a new connection, otherwise we send to the existing addy
        int index = newPeer(recipient, port);

        return sendToPeer(m, index);


    }

    public synchronized void removePeer(int i) {
        connections[i] = null;
    }

    public synchronized void addMessage(cpgMessage m) {
        incomingMessages.add(m);
    }

    public synchronized boolean messageAvailable() {
        return (incomingMessages.size() > 0);
    }

    public synchronized cpgMessage getMessage() {
        try {
            return (cpgMessage)incomingMessages.remove(0);
        } catch (Exception E) { System.err.println("invalid call to getMessage"); }
        return null;
    }

    public void run() {
        while(true) {
            try {
            // sleep until keepalive
                sleep(29000);
                keepSocketsAlive();
            } catch (InterruptedException ie) { System.out.println("interrupted");}
        }

    }

}
```

10.2.4 cpggui.java

```
import javax.swing.*;
import java.awt.BorderLayout;
```

```
import java.awt.event.*;
import javax.swing.tree.*;
import javax.swing.event.*;
import java.util.*;
import java.awt.FlowLayout;
import java.awt.GridLayout;

/** cpggui is the gui handler for a CPG
  * Written by: Douglas Johnson
  * May 21st 2001
  **/

public class cpggui extends JFrame implements ActionListener {

    private cpg parent = null;
    private JPanel eastPanel, centerPanel, bottomPanel;
    private JButton newSurrogate, addParent, requestButton;
    private Vector surrogateVector;
    private matrixlist ilist, elist, plist, alist;

    public cpggui(cpg parent, String name) {

        setTitle(name);
        surrogateVector = new Vector();

        getContentPane().setLayout(new BorderLayout());
        //I am representative of a specific CPG
        this.parent = parent;
        getContentPane().add(new JLabel(name), BorderLayout.NORTH);

        bottomPanel = new JPanel(new FlowLayout());
        addParent = new JButton("Connect");
        addParent.addActionListener(this);
        bottomPanel.add(addParent);
        requestButton = new JButton("Request");
        requestButton.addActionListener(this);
        bottomPanel.add(requestButton);
        getContentPane().add(bottomPanel, BorderLayout.SOUTH);


        eastPanel = new JPanel(new GridLayout(1,1));
        newSurrogate = new JButton("Add Surrogate");
        newSurrogate.addActionListener(this);
        eastPanel.add(newSurrogate);
        getContentPane().add(eastPanel, BorderLayout.EAST);
```

```
    //create a window adapter that shuts things down when x is selected
    addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
          System.exit(0);
       }
    });

    doMenu();
    doMatrix();

    getContentPane().add(centerPanel, BorderLayout.CENTER);
  }


/** public void doMenu()
  * This is really just to encapsulate some of the functions from
  * the constructor in another method so that it is cleaner code
  * This handles all the menu operations
  **/
  public void doMenu() {

     //set up a file menu with one submenu and an exit menuItem
     JMenuBar menuBar = new JMenuBar();
     setJMenuBar(menuBar);

     JMenu menu = new JMenu("System");
     menu.setMnemonic(KeyEvent.VK_S);
     menu.setToolTipText("Hot tip: System menu operations");

     JMenuItem printMI = new JMenuItem("Print");
     printMI.addActionListener(this);
     printMI.setMnemonic(KeyEvent.VK_P);
     menu.add(printMI);

     // exit
     JMenuItem exitMI = new JMenuItem("Exit");
     exitMI.addActionListener(this);
     exitMI.setMnemonic(KeyEvent.VK_X);
     menu.add(exitMI);


     menuBar.add(menu);

  }// doMenu

  private void doMatrix() {
```

```
        centerPanel = new JPanel(new GridLayout(4,1));

        centerPanel.add(new JLabel("IData"));
        ilist = new matrixlist(parent.metricmatrix.geti());
        centerPanel.add(ilist);
        //centerPanel.add(new matrixlist(parent.getMatrix('I')));
        centerPanel.add(new JLabel("EData"));
        elist = new matrixlist(parent.metricmatrix.gete());
        centerPanel.add(elist);
        //centerPanel.add(parent.metricmatrix.gete());
        centerPanel.add(new JLabel("AData"));
        alist = new matrixlist(parent.metricmatrix.geta());
        centerPanel.add(alist);
        //centerPanel.add(parent.metricmatrix.geta());
        centerPanel.add(new JLabel("PData"));
        plist = new matrixlist(parent.metricmatrix.getp());
        centerPanel.add(plist);
        //centerPanel.add(parent.metricmatrix.getp());

    }// private void doMatrix

    private void addParent() {
        // we first get the info as to where the child is
        String parentIP = JOptionPane.showInputDialog(this, "Please enter the Root IP
address", "Child IP", JOptionPane.QUESTION_MESSAGE);
        String parentPort = JOptionPane.showInputDialog(this, "Please enter the Childs
port number", "Child Port", JOptionPane.QUESTION_MESSAGE);

        int port = Integer.parseInt(parentPort);
        // now we create a connection to that child
        parent.newRoot(parentIP, Integer.parseInt(parentPort));
        addParent.disable();
    }

    private void addSurrogate() {
        GridLayout gl;
        surrogate newS = new surrogate(parent);
        JButton newSBut = newS.getButton();
        gl = (GridLayout)eastPanel.getLayout();
        gl.setRows(gl.getRows() + 1);
        eastPanel.add(newSBut);
        surrogateVector.add(newS);
        setSize(getPreferredSize());

        validate();
```

```
}

/** testRequest()
 * test request is the request analyzing function. It prompts for
 * a client IP address, and given a VALID ip will deliver the best server
 * choice. It bases it's information on the processing data, so given
 * a request at any given level it will only choose from its own level, or
 * those levels that it is responsible for.
**/
private void testRequest() {

    String clientIPString = JOptionPane.showInputDialog(this, "Please enter the
client's IP address", "Client IP", JOptionPane.QUESTION_MESSAGE);
    // if the IP was invalid return
    if(clientIPString.length() == 0) return;
    ip clientIP = new ip(clientIPString);
    if (clientIP.length == 4 || clientIP.length == 16)
    parent.BSC(clientIP);

}

/** updatelist()
  * this functions force the gui to update the current view of the
  * represented data. This is the internal view and will generally have
  * more farreaching effects, for instance updating i means re-advertising
  * information, which means updating aData (or pData, and in some cases both)
  * Though the actual updating of the other matrices is not done here, this
  * is strictly gui handling.
  **/
public void updatelist(char letter){
    switch (letter) {
        case 'I':
                ilist.setData(parent.metricmatrix.geti());
                return;
        case 'E':
                elist.setData(parent.metricmatrix.gete());
                return;
        case 'P':
                plist.setData(parent.metricmatrix.getp());
                return;
        case 'A':
                alist.setData(parent.metricmatrix.geta());
                return;
    } // switch

//Vector updated = parent.metricmatrix.getI();
```

```
   }

/** actionPerformed
  * This function is the handler for all actions within the scope of the
  * cpg. It works on all the menu options and buttons we create
  **/

   public void actionPerformed(ActionEvent e){
       if(e.getActionCommand().equals("Print"))
           System.out.println("Action Print");
      if(e.getActionCommand().equals("Add Surrogate"))
           addSurrogate();
      if(e.getActionCommand().equals("Connect"))
           addParent();
      if(e.getActionCommand().equals("Request"))
           testRequest();
      if(e.getActionCommand().equals("Exit"))
         System.exit(0);
   }


} // cpggui
```

10.2.5 cpgMessage.java

```
import java.io.Serializable;
import java.util.Vector;
import java.net.*;
public class cpgMessage implements Serializable {

   public static final byte NULL = 0;
   public static final byte HELLO = 1;
   public static final byte UPDATE = 2;
   public static final byte REMOVE = 3;
   public static final byte RESET = 4;
   public static final byte TERM = 5;
   public static final byte KA = 6;
   public static final byte CONTEND = 7;
   public static final byte NEW = 8;


   public int recievingID = -1;
   //public byte[] SenderID = new byte[4];
   public InetAddress sender = null;
   public byte[] DataLength = new byte[4];
   public datamatrix Data = null;
```

```
    public byte[] Options = null;
    public boolean newRow = false;

    public byte Type = NULL;

    public cpgMessage() {
    }

    public cpgMessage(byte type) {
        this.Type = type;
        if ((Type == NEW) || (Type == UPDATE) || (Type == REMOVE) || (Type ==
CONTEND)) Data = new datamatrix();

    }

    public cpgMessage(byte type, int myid) {
        this.Type = type;
        if ((Type == NEW) || (Type == UPDATE) || (Type == REMOVE) || (Type ==
CONTEND)) Data = new datamatrix();
        //this.senderID =
    }

    public void setrecievingID(int i) {
        this.recievingID = i;
    }

    public void addData(row inRow) {
        if (Data == null) Data = new datamatrix();
        Data.add(inRow);
    }

    public void addData(datamatrix inVector) {
        Data = inVector;
    }

    public Vector getData() {
        if (Data == null) return new datamatrix();
        return Data;
    }

    // This sets options for messages, for instance to set the
    // parent option to PEER simply cpgMessageInstance.setOption(PEER);
    // options must be set in the order determined by the message type
    // before being sent.
    public void setOption(byte[] option) {
        byte[] tempOptions = new byte[Options.length + option.length];
```

```
        for(int i = 0; i < Options.length; i++)
            tempOptions[i] = Options[i];
        for(int j = 0; j < option.length; j++)
            tempOptions[Options.length + j] = option[j];
        Options = tempOptions;
        return;
    }

    public String toString() {

        switch (Type) {
            case 0: return "NULL";
            case 1: return "HELLO";
            case 2: return "UDPATE";
            case 3: return "REMOVE";
            case 4: return "RESET";
            case 5: return "TERM";
            case 6: return "KEEP ALIVE";
            case 7: return "CONTENTION";
            case 8: return "NEW";
            default: return "no type specified";
        }
    }
}
```

10.2.6 datamatrix.java

```
import java.util.*;
import java.io.Serializable;
/** datamatrix.java
  * This is a sorted list of rows. It is used in our CPG system
  * in order to store the various parts of the metric matrix
  * mutually exclusive blocks of code in synchronized methods
  * Written: Douglas Johnson
  *         May 21st 2001
  **/

public class datamatrix extends Vector implements Serializable {

    /** datamatrix()
      * default constructor, empty set, no elements.
      **/

    public datamatrix() {
        super();
    } //dataMatrix
```

```
    //override the default add function
    public boolean add(Object o) {
        row myrow;
        try {
            myrow = (row)o;
        } catch(ClassCastException e) {
            System.err.println("invalid object, matrix only holds rows");
            return false;
        }
        addRow(myrow);
        return true;
    }

    /** findMatches
      * find matches will search through the given data matrix
      * and return a new datamatrix with all matching IP addresses
      * this takes into account subset matches (prefixes)
      * as this is syncrhonized, it cannot be recursive
      **/
    synchronized public datamatrix findMatches(ip baseIP, boolean either) {

        // first find a matching first digit
        // this could be done via an nlogn algorithm for better efficiency
        datamatrix returnMatrix = new datamatrix();
        int left = 0;
        int right = size();

        if(either) {
            /*
            // we'll try to match up to the baseIP length
            // and we ensure that our search range >= 0
            // since this is matching either way, we temporarily look for a smaller IP
            short[] temparray = new short[1];
            temparray[0] = baseIP.Prefix[0];
            ip tempIP = new ip(temparray);

            while(left < right && (((ip)((row)elementAt(left)).footprint).compareTo(tempIP))
>= 0) {
                // find matching ip left as first index
                left++;
            }


            // if left >= right, then we are out of elements, no matches
            if(left == right) return returnMatrix;
```

```
            //now make right the last element
            right--;

            // find the right index

            short[] farRightByteArray = new short[4];
            for(int i = 0; i < 4; i++) {
                farRightByteArray[i] = (i < baseIP.length) ? baseIP.Prefix[i] : 255;
            }
            ip farRight = new ip(farRightByteArray);
            System.out.println(farRight);
            while(right >= left && farRight.compareTo((ip)((row)elementAt(right)).footprint)
<= 0) {
                right--;
            }

            // from left to right, see if the footprint represents this ip
            // add it to the list if it does
            row baseRow = new row();
            baseRow.footprint = baseIP;
            for(int i = left; i <= right; i++) {
                if(((row)elementAt(i)).represents(baseIP) ||
                    baseRow.represents(((row)elementAt(i)).footprint))

                    returnMatrix.add(elementAt(i));
            }
            */

// Changed to linear search, a really slow way to go, but this
// is just to prove it can work. Implementations could
// use and nLogN search algorithm to find the left and right
// indices, and then do a linear search on a much smaller list

            while (left < right) {
                row baseRow = new row();
                baseRow.footprint = baseIP;

                if(((row)elementAt(left)).represents(baseIP) ||
baseRow.represents(((row)elementAt(left)).footprint))
                    returnMatrix.add(elementAt(left));
                left++;
            }


        } else {
```

```
        // we'll try to match up to the baseIP length
        // and we ensure that our search range >= 0
        //while(left <= right && baseIP.compareTo((ip)((row)elementAt(left)).footprint) > 0)
{
        while(left < right && (((ip)((row)elementAt(left)).footprint).compareTo(baseIP)) > 0)
{
            // find matching ip left as first index
            left++;
        }

        // if left >= right, then we are out of elements, no matches
        if(left == right) return returnMatrix;

        //now make right the last element
        right--;


            // now find right index
            // while our right is right of left, and is bigger than our ip
            while(right >= left && baseIP.compareTo((ip)((row)elementAt(right)).footprint) <
0) {
                right--;
            } // while

            // from left to right, see if the footprint represents this ip
            // add it to the list if it does

            for(int i = left; i <= right; i++) {
                if(((row)elementAt(i)).represents(baseIP)) {
                    returnMatrix.add(elementAt(i));
                }
            }
        } // if else


        return returnMatrix;


    }

    /** findMatches
      * find matches will search through the given data matrix
      * and return a new datamatrix references to the rows by a given reporter
      * as this is syncrhonized, it cannot be recursive
      **/
```

```
synchronized public datamatrix findMatches(int reportingCPG) {

    // first find a matching first digit
    // this could be done via an nlogn algorithm for better efficiency
    datamatrix returnMatrix = new datamatrix();
    int left = 0;
    int right = size();


    while(left < right) {
        if(((row)elementAt(left)).reportIndex == reportingCPG) {
            ((row)elementAt(left)).reported = true;
            returnMatrix.add(elementAt(left));
        }
        left++;
    }

    return returnMatrix;


}

/** addRow(row newRow)
  * addRow puts a row into the matrix. It sorts them according to
  * the row comparable operation
**/
synchronized public void addRow(row newRow) {
    int i = 0;
    while(i < size() && newRow.compareTo((row)this.elementAt(i)) > 0) {
        i++;
    } // while, we have now found where to put in the row
    // we do not insert duplicate rows AND we keep the old row if possible
    if (i < size() && newRow.compareTo((row)this.elementAt(i)) == 0) return;
    this.insertElementAt(newRow, i);
    return;

} // addRow

/** removeRow(short[] )
  * removeRow, not sure exactly how I'm going to implement this yet
  * what remove operations will be useful. All about a CPG perhaps?
**/
synchronized public void removeRow(row oldRow) {
    removeElementAt(indexOf(oldRow));
} // removeRow
```

```
    //** replaceRow(row newRow) {
    synchronized public void replaceRow(row newRow) {
        int index = indexOf(newRow);
        if(index == -1) { System.err.println("possible place to raise an exception" +
            " as the row being replaced doesn't exist. meaning that the UPDATE" +
            " was of a non-existant row");
            return;
        }
        setElementAt(newRow, index);
        return;

    }

    public String toString() {
        String myString = "dataMatrix";

        // for each of our rows, calls its toString method and add a newline

        return myString;
    }

    private void print(String s) {
        System.out.println("datamatrix: " + s);
    }

} //data matrix
```

10.2.7 ip.java

```
import java.lang.Comparable;
import java.io.Serializable;
/** class ip is wrapper for my ip short arrays
  * Written: Douglas Johnson
  * May 23 2001
  **/

public class ip implements Comparable, Serializable {

    public short[] Prefix;
    public int length;
    public int port;

    public ip() {
        Prefix = new short[] {0,0,0,0};
        length = 4;
    }
```

```
    public ip(short[] input) {
        Prefix = input;
        length = input.length;
    }

    public ip(String input) {
        Prefix = ip.parseIP(input);
        length = Prefix.length;
    }

    public String toString() {
        String myString = null;
        if (Prefix.length >= 0) {
            myString = "" + Prefix[0];
        } else {
            return "null";
        }
        for(int i = 1; i < Prefix.length; i++) {
            myString = myString + "." + Prefix[i];
        }
        return myString;

    } // toString

    /** matchByte(short tomatch, int position)
      * matchByte takes a short to match and a position to match it
      * at and returns 0 if they match negative if the tomatch is larger
      * and positive if it is smaller
      **/
    public int matchByte(short tomatch, int position) throws
ArrayIndexOutOfBoundsException {
        return (Prefix[position] - tomatch);
    } //matchByte


    /** parseIP takes in a string and returns the short
      * array that we recognize from that string.
      * it only works with delminated string
      **/
    public static short[] parseIP(String inString) {
        String digit = null;
        short[] ipBuffer = new short[16];
        short[] ipPref = null;
        int j = 0;
```

```
    try {

        // Parse the IP field
        for(int i=0; i<inString.length();) {
            digit = "";
            while(i < inString.length() && Character.isDigit(inString.charAt(i))) {
                digit = digit + inString.charAt(i);
                i++;
            }// while

            //we now have a digit
            try {
                ipBuffer[j] = Short.parseShort(digit);
                if(ipBuffer[j] > 255 || ipBuffer[j] < 0) throw new Exception();
            } catch(Exception digiterror) { System.out.println("invalid digit"); }

            j++;
            i++;

        } // for parsing ip

        ipPref = new short[j];
        for(int k = 0; k < j; k++) {
            ipPref[k] = ipBuffer[k];
        }
    } catch (NumberFormatException nfe) { System.err.println("no number"); }
    return ipPref;
} //parseIP

/** accordingto strict definition of comparable
  * if the ip is smaller numberically, it is
  * less than
  * else it is the same
  * returns negative IFF the object is less than
  * returns 0 IFF the object is equal to
  * returns positive IFF the object is greater than
  * note that shorter but otherwise equal ips will
  * be considered numerically smaller
  **/

public int compareTo(Object ip1) {
    boolean done = false;
    int i = 0;
    ip inIP = (ip)ip1;

    while(i < length && i < inIP.length) {
```

```
            if(Prefix[i] < inIP.Prefix[i]) return -1;
            if(Prefix[i] > inIP.Prefix[i]) return 1;
            i++;
        }
        // will only exit from while loop if one of the prefixes
        // is shorter than the other, but otherwise the same
        // so we check to see which was shorter, to be the smaller
        // ip value


        // this is in case there are port number associated with
        if ((length == inIP.length) && (port == inIP.port)) return 0;

        if (i == length && i == inIP.length) return 0;
        if(i == length) return -1;
        if(i == inIP.length) return 1;



        // if we make it this far, something is wrong
        return 0;
    }

}// class ip
```

10.2.8 matrix.java

```
public class matrix {

    private datamatrix iData = new datamatrix();
    private datamatrix eData = new datamatrix(); // external data
    private datamatrix pData = new datamatrix(); // process data
    private datamatrix aData = new datamatrix(); // advertised data


/** The matrix is made of three of these sets of data
  * used for a surrogate to add a row to the iData matrix
  * used for a surrogate to add a row to the iData matrix
  * each serving the purpose outlined in the design specifications
  * The pData and aData are mutually exclusive, however in some cases
  * may
  **/
    public matrix() {

        iData = new datamatrix();
```

```
        eData = new datamatrix();
        pData = new datamatrix();
        aData = new datamatrix();

    }

    public datamatrix geti(){

        return iData;
    }// datamatrix geti

    public datamatrix gete(){

        return eData;

    }// datamatrix geti

    public datamatrix getp(){

        return pData;

    }// datamatrix geti

    public datamatrix geta(){

        return aData;

    }// datamatrix geti


/** public void addi()
  * addi adds rows to the iData, it is called directly by the cpg
  **/
    public void addi(row newRow) {
        iData.addRow(newRow);
    } //iAddRow


    public void updatei(row newRow) {
        iData.replaceRow(newRow);
    }

/** public void removei(row oldrow)
  * removes this row from the internal matrix
  **/
    public void removei(row oldrow) {
```

```
        iData.removeRow(oldrow);
    }

/** public void eAddRow()
  **/
    public void adde(row newRow) {
        eData.addRow(newRow);
    } //eAddRow


    public void updatee(row newRow) {
        eData.replaceRow(newRow);
    }


    public void removee(row oldRow) {
        eData.removeRow(oldRow);
    }

/** public void pAddRow()
  **/
    public void addp(row newRow) {
        pData.addRow(newRow);
    } //addp

    public void updatep(row newRow) {
        pData.replaceRow(newRow);
    }


    public void removep(row oldRow) {
        pData.removeRow(oldRow);
    }


/** public void aAddRow()
  **/
    public void adda(row newRow) {
        aData.addRow(newRow);
    } //aAddRow

    public void updatea(row newRow) {
        aData.replaceRow(newRow);
    }

    public void removea(row oldRow) {
```

```
        aData.removeRow(oldRow);
    }

/** public void resetA()
  * this removes everything from A, then re-adds those rows that are internal
  **/
    public void resetA() {
        aData = new datamatrix();
    }



} // class matrix
```

10.2.9 matrixlist.java

```
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import javax.swing.ListSelectionModel;
import javax.swing.*;
import javax.swing.event.*;

import javax.swing.DefaultCellEditor;

import javax.swing.JScrollPane;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import java.awt.*;
import java.awt.event.*;

import java.util.*;


/** surrogatelist
  * This is a modified version of code written by Robert Watson
  * Written by Douglas Johnson
  * It has been changed drastically
  **/

public class matrixlist extends JPanel {

    private boolean DEBUG = false;
    //private Vector IPPrefixes = null;
    //private Vector IPLag = null;
    private datamatrix myRows = null;
    private JTable table;
```

```
//public matrixlist(Vector IPPrefixes, Vector IPLag) {
public matrixlist(datamatrix myRowsIn) {
    super();

    this.myRows = myRowsIn;
   matrixTableModel theModel = new matrixTableModel();
   table = new JTable(theModel);
   table.setPreferredScrollableViewportSize(new Dimension(200, 70));

   JScrollPane scrollPane = new JScrollPane(table);
   this.add(scrollPane, BorderLayout.CENTER);

   theModel.setData(myRows);
}

public void setData(datamatrix myRows) {
    this.myRows = myRows;
    matrixTableModel theModel = (matrixTableModel)table.getModel();
    theModel.setData(myRows);
    theModel.fireTableDataChanged();
}

/** return a selected row number
  * used for removing rows
  **/
public int selectedRow() {
    return table.getSelectedRow();
}

class matrixTableModel extends AbstractTableModel {
    final String[] columnNames = {"IP Prefix", "Delay", "Destination"};

    private Vector myRows = null;

    public void setData(Vector myRows) {
        this.myRows = myRows;
    }

    public int getColumnCount() {
        return columnNames.length;
    }

   public int getRowCount() {
    if (myRows != null) {
        return myRows.size();
```

```
        } else {
          return 0;
        }
      }

      public String getColumnName(int col) {
        return columnNames[col];
      }

      public Object getValueAt(int listrow, int col) {

       Object theObject = null;

          String myString = null;

          row theRow = (row)myRows.elementAt(listrow);

          // for the matrix we'll have to make some more
          if (theRow != null) {
              if (col == 0) {
                  theObject = theRow.footprint;
          } else if (col == 1) {
                  theObject = new Long(theRow.delay);
          } else if (col == 2) {
                  theObject = theRow.destinationIP;
              }


       } else {
          theObject = new String("null");
       }
          return theObject;
       }

       public Class getColumnClass(int c) {
          return getValueAt(0,c).getClass();
       }

       /** nothing is editable right now
       **/
       public boolean isCellEditable(int row, int col) {
          return false;
       }
    }
}
```

10.2.10 row.java

```
import java.lang.Comparable;
import java.io.Serializable;
import java.net.InetAddress;
/** Class row is the datastructure that holds information pertaining to a
  * surrogate. It holds the footprint, delay to footprint. There is a
  * seperate row for each footprint that a surrogate advertises a delay for
  * as such there are SUM(n * m) rows, where n is the number of surrogates
  * and m is the number of ip-prefixes that that surrogate advertises a
  * delay for
  *
  * I have chosen to keep the variables public, so that the owner program may
  * operate on the variables directly. I do not suspect this will cause any
  * race conditions because the rows themselves will only have one owner at
  * a time
  *
  * Written: Douglas Johnson
  *     May 19 2001
  **/

public class row implements Serializable, Comparable, Cloneable {

    public static byte ipv4 = 0;
    public static byte ipv6 = 1;

//  public byte[] IPPrefix = null;
//  public byte IPPrefixVersion = ipv4;
//  public byte[] destinationIP = null;
//  public byte destinationIPVersion = ipv4;
//  public byte[] reportingIP = null;
//  public byte reportingIPVersion = ipv4;

    public ip footprint = null;
    public byte IPPrefixVersion = ipv4;
    public ip destinationIP = null;
    public byte destinationIPVersion = ipv4;
    public ip reportingIP = null;
    public byte reportingIPVersion = ipv4;

    public int reportIndex = -1;
    public InetAddress reportTo = null;
    public int port = -1;
    public boolean reported = true;

    public long delay = 1000; // delay to footprint in mSec
```

```
    public byte sla = 0; // by default the SLA info for this row

    public boolean dirty = false;

    public row(short[] IPPrefix, short[] destination, short[] reportingIP, long delay, byte
sla) {
        this.footprint = new ip(IPPrefix);
        this.destinationIP = new ip(destination);
        this.reportingIP = new ip(reportingIP);

        this.IPPrefixVersion = (IPPrefix.length > 4) ? ipv4 : ipv6;
        this.destinationIPVersion = (destination.length > 4) ? ipv4 : ipv6;
        this.reportingIPVersion = (reportingIP.length > 4) ? ipv4 : ipv6;

        this.delay = delay;
        this.sla = sla;

    } // constructor row

    public row(short[] IPPrefix, long delay){
        this.footprint = new ip(IPPrefix);
        this.delay = delay;
        this.destinationIP = new ip();
        this.reportingIP = new ip();
    }

    public row() {
        this.footprint = new ip();
        this.destinationIP = new ip();
        this.reportingIP = new ip();
    }

    public row(short[] IPPrefix, long delay, short[] destination){
        this.footprint = new ip(IPPrefix);
        this.delay = delay;
        this.destinationIP = new ip(destination);
    }



    public void setdestination(short[] destIP) {
        this.destinationIP = new ip(destIP);
    }

    /** public String toString()
      **/
```

```
    public String toString() {

        return "Prefix :" + footprint + " delay :" + delay + " reportTo:" + reportTo + " port: "
  + port;

    }

    /** public row clone()
      * returns an object that is identical to this row
      **/
    public Object clone() {
        row clone = new row();
        clone.footprint = this.footprint;
        clone.destinationIP = this.destinationIP;
        clone.reportingIP = this.reportingIP;
        clone.delay = this.delay; // delay to footprint in mSec
        clone.sla = this.sla; // by default the SLA info for this row
        clone.reportTo = this.reportTo;
        clone.port = this.port;
        clone.reported = this.reported;
        return clone;
    } //clone

    /** public boolean represents(row r1)
      * this is to check if the footprint of this row represents the IP passed
      * to it. (via the footprint) it checks to see if the footprint is
      * the same
      **/

    public boolean represents(ip i) {

        // we can only be representative of ip's that we are shorter than
        if (footprint.length > i.Prefix.length) return false;

        for(int j = 0; j < footprint.length; j++) {
            if(footprint.Prefix[j] != i.Prefix[j]) return false;
        }

        return true;

    }

    /** public boolean representative(row r1)
      * this is to check if the footprint of this row represents the IP passed
      * to it. (via the footprint) it checks to see if the footprint is
      * the same
```

```
 **/

public boolean representative(ip r1) {

    int smaller = (footprint.length < r1.length) ? footprint.length : r1.length;

    for(int j = 0; j < smaller; j++) {
        if(footprint.Prefix[j] != r1.Prefix[j]) return false;
    }

    return true;

}

/** Public int compareTo(Object o1)
 * this compares to rows to see which will come first in an ordered set
 * by our convention, this will mean that the row with the smallest
 * IP Prefix (numerically) will come first. In the case of a tie, the
 * row that then has the smallest delay will come first.
 * As a final tie breaker (ie, two surrogates report the same delay
 * to a common footprint) the arbitrator will be the routing destination
**/
public int compareTo(Object r1) {
    row row1 = (row)r1;

    try {
        // we have 2 rows, compare them.
        // start with their ip prefixes
        // if it's footprint is before ours, return 1
        if(footprint.compareTo(row1.footprint) > 0) return 1;
        // if it's footprint is after ours, return -1
        if(footprint.compareTo(row1.footprint) < 0) return -1;

        // so its footprint must be the same as ours
        // check the delay
        if(delay < row1.delay) return -1;
        if(delay > row1.delay) return 1;

        // footprint and delay are equal, check the destination
        if(destinationIP.compareTo(row1.destinationIP) > 0) return 1;
        if(destinationIP.compareTo(row1.destinationIP) < 0) return -1;

        // if the footprint, delay and destinationIP are all the same
        // we have defined these as equal rows...

        return 0;
```

```
        } catch(Exception e) {System.out.println("invalid row");}

        return 1;

    }

    public boolean equals(Object r2) {
        try {
            row row1 = (row)r2;

            if(footprint.compareTo(row1.footprint) != 0) return false;
            if(delay != row1.delay) return false;
            if(destinationIP.compareTo(row1.destinationIP) != 0) return false;

            // if the footprint, delay and destinationIP are all the same
            // we have defined these as equal rows...
            return true;

        } catch (Exception e1) { return false; }
    } //boolean equals

} // class row
```

10.2.11 socketHandler.java

```
import java.io.*;
import java.lang.Thread;
import java.net.*;

public class socketHandler extends Thread {

private Socket cSocket = null;
private ObjectOutputStream writer = null;
private ObjectInputStream reader = null;
private int myindex;
private int myport = 0;
private cpgexchange reportTo = null;
private InetAddress myInet = null;

public socketHandler(InetAddress childInet, int portno, int myindex, cpgexchange
reportTo) throws IOException {
    super();
    BufferedInputStream bis = null;
    this.myindex = myindex;
    this.reportTo = reportTo;
```

```
    this.myport = portno;
    this.myInet = childInet;

    try {
        this.cSocket = new Socket(myInet, portno);
        bis = new BufferedInputStream(cSocket.getInputStream());
        writer = new ObjectOutputStream(cSocket.getOutputStream());
        reader = new ObjectInputStream(bis);
        System.out.println("Successfully made socket Handler");
    } catch (IOException e) {
        System.out.println("error making socketHandler");
        close();
        throw e;
    }

} // socketHander constructor

public socketHandler(Socket inSocket,  int myindex, cpgexchange reportTo) throws
IOException {
    super();
    BufferedInputStream bis = null;
    this.myindex = myindex;
    this.reportTo = reportTo;
    this.myport = inSocket.getPort();
    this.myInet = inSocket.getInetAddress();

    try {
        this.cSocket = inSocket;
        bis = new BufferedInputStream(cSocket.getInputStream());
        writer = new ObjectOutputStream(cSocket.getOutputStream());
        reader = new ObjectInputStream(bis);
    } catch (IOException e) {
        System.out.println("error making socketHandler");
        close();
        throw e;
    }

} // socketHander constructor

public boolean compareTo(InetAddress ia, int p) {
    System.out.println("comparing: " + ia + " to " + myInet);
    return (ia.equals(myInet) && p == myport);
}

public synchronized void send(cpgMessage m) {
    try {
```

```
    writer.writeObject(m);
    writer.flush();
    } catch(IOException e) {
        System.out.println("error sending message");
        close();
    }
}

/** provided for synchronization **/
public synchronized void report(cpgMessage m) {

        reportTo.addMessage(m);

}

public void close() {
    try {

        writer.close();
        reader.close();
        cSocket.close();
    } catch (Exception e) {
    } finally {
        writer = null;
        reader = null;
        cSocket = null;
        reportTo.removePeer(myindex);
        destroy();
    }

}

public void run() {
    cpgMessage newMessage = null;

    // main infinite loop
    while(true) {

    try {
        newMessage = (cpgMessage)reader.readObject();
        newMessage.setrecievingID(myindex);
        newMessage.sender = cSocket.getInetAddress();
        report(newMessage);

    } catch (ClassNotFoundException ci) { System.err.println("class cast");
        ci.printStackTrace();
```

```
    } catch (OptionalDataException ode) { System.err.println("optional Data");
        ode.printStackTrace();
    } catch (NullPointerException npe) {
        close();
    } catch (IOException io) {
        System.err.println("Error reading object, closing child");
        close();
    }


    } // while
}

} // class
```

10.2.12 surrogate.java

```
import java.util.Vector;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

/** Class surrogate
  * This class implements the basic surrogate structure, allowing one to
  * bind a surrogate to a CPG, to which it then reports all updates to its
  * IP Footprint.
  * Written: Douglas Johnson
  * May 22nd 2001
  **/

public class surrogate extends JFrame implements ActionListener {
    public short[] myIP;
    private datamatrix myRows;
    public JButton myButton, addButton, delButton;
    private cpg parent;
    private Container myPanel, EastContainer;
    private addRow myAddRow = null;
    private matrixlist mylist = null;

    public surrogate(cpg mycpg) {

        parent = mycpg;
        myIP = new short[] { 0,0,0,0 };
        //myIP = new short[] {12, 24, 48};
        myButton = new JButton("Surrogate", new ImageIcon("./icons/surrogate.ico"));
        myButton.addActionListener(this);
```

```
 EastContainer = new Container();
 EastContainer.setLayout(new GridLayout(2,1));

 addButton = new JButton("Add Row");
 addButton.addActionListener(this);

 delButton = new JButton("Delete Row");
 delButton.addActionListener(this);

 EastContainer.add(addButton, BorderLayout.EAST);
 EastContainer.add(delButton, BorderLayout.EAST);


 myRows = new datamatrix();

 //Setup some default values

 myPanel = getContentPane();
 myPanel.setLayout(new BorderLayout(1,1));
 myPanel.add(new JLabel("Surrogate"), BorderLayout.NORTH);

 // Construct the surrogate lists
 mylist = new matrixlist(myRows);
 myPanel.add(mylist, BorderLayout.CENTER);
 myPanel.add(myButton, BorderLayout.SOUTH);

 myPanel.add(EastContainer, BorderLayout.EAST);



 //create a window adapter that shuts things down when x is selected
addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
       hide();
       //setVisible(false);
    //  System.exit(0);
   }
});

 // Messy Menu stuff in a lonely function below
 doMenu();

 // lastly, set the size up.. must have the menu first
```

```
        setSize(new Dimension((int)getPreferredSize().getWidth(),
(int)(getPreferredSize().getHeight() + 15)));
        setVisible(true);

        String ipString = JOptionPane.showInputDialog(this, "Please enter this surrogates
IP address", "IP Input", JOptionPane.QUESTION_MESSAGE);
        myIP = (ipString.length() > 0) ? ip.parseIP(ipString) : new short[] { 0, 0, 0, 0};
    } // constructor


    public JButton getButton() {
        return myButton;
    }


    public void doMenu() {

        //set up a file menu with one submenu and an exit menuItem
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        JMenu menu = new JMenu("System");
        menu.setMnemonic(KeyEvent.VK_S);
        menu.setToolTipText("Hot tip: System menu operations");

        JMenuItem printMI = new JMenuItem("Print");
        printMI.addActionListener(this);
        printMI.setMnemonic(KeyEvent.VK_P);
        menu.add(printMI);

         // exit
        JMenuItem exitMI = new JMenuItem("Exit");
        exitMI.addActionListener(this);
        exitMI.setMnemonic(KeyEvent.VK_X);
        menu.add(exitMI);


        menuBar.add(menu);

    }// doMenu

    /** seemingly simple newRow is all that remains to add a row
      * to a surrogates row list. This must then be updated to the
      * cpg, however that function will be performed in setData
      **/
    public void newRow(row inRow) {
```

```
        inRow.setdestination(myIP);
        myRows.add(inRow);
        parent.addRowInternal(inRow);
        mylist.setData(myRows);
    }

    /** seemingly simple removeRow is all that remains to remove a row
     * from a surrogates row list. This must then be updated to the
     * cpg, however that function will be performed in setData
     **/
    public void removeRow() {
        parent.removeRowInternal((row)myRows.get(mylist.selectedRow()));
        myRows.removeElementAt(mylist.selectedRow());
        mylist.setData(myRows);
    }

    public void actionPerformed(ActionEvent e){
        if(e.getActionCommand().equals("Print"))
      if(e.getActionCommand().equals("Exit"))
           hide();
           //setVisible(false);
           //System.exit(0);
      if(e.getActionCommand().equals("Surrogate")) {
           show();
           //setVisible(true);
           //get the focus
           pack();
           repaint();
      }
      if(e.getActionCommand().equals("Add Row")) {
           myAddRow = new addRow(this);
      }
      if(e.getActionCommand().equals("Delete Row"))
           removeRow();
    }

private class addRow extends JFrame implements ActionListener {

    private surrogate parent = null;

    private JTextField ipField = null, delayField = null;
    private Container myPane = null;
    private JButton okButton, cancelButton;

    public addRow (surrogate parent) {
```

```
        this.parent = parent;

        myPane = this.getContentPane();

        ipField = new JTextField();
        delayField = new JTextField();
        myPane.setLayout(new GridLayout(3,2));
        myPane.add(new JLabel("IP"));
        myPane.add(new JLabel("delay"));
        myPane.add(ipField);
        myPane.add(delayField);

        okButton = new JButton("OK");
        okButton.addActionListener(this);

        cancelButton = new JButton("Cancel");
        cancelButton.addActionListener(this);

        myPane.add(okButton);
        myPane.add(cancelButton);

        setSize(getPreferredSize());
        myPane.validate();
        this.setVisible(true);

    } // constructor

    public void actionPerformed(ActionEvent e) {
        String text = null;
        short ipPref[];
        if(e.getActionCommand().equals("OK")){

            text = ipField.getText();
            ipPref = ip.parseIP(text);
            parent.newRow(new row(ipPref, Long.parseLong(delayField.getText())));
            this.setVisible(false);
        } //ok button //ok button
        if(e.getActionCommand().equals("Cancel")) {
            this.setVisible(false);
        }

    } //ActionPerformed

} // AddRow class
```

```
} // surrogate class
```

Vita

Surname:	Johnson	Given Names:	Douglas Arthur
Place of Birth:	Prince George, British Columbia, Canada		

Educational Institutions Attended:

University of Victoria	1993 - 2001

Degrees Awarded:

B.Sc.	University of Victoria	2000

Honors and Awards:

NewMIC Scholarship	2000 - 2002

University of Victoria Partial Copyright License

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

A Protocol for the Interoperability of Content Distribution Networks

Author ______________________

Douglas Arthur Johnson

February 14, 2002