A Content-Based Load-Balancing Router System [*]

Michael Batchelder and Kacper Wysocki

Abstract

This paper outlines the implementation of a content-based load-balancing service-attached router. The content that special routing is performed on is HTTP traffic. A unique routing protocol is implemented on top of IP which allows for querying and sharing of content-based routing information among a network of such routers. Each router that receives a new HTTP request lookups the web servers available for the given URL embedded in the HTTP request and load-balances this traffic by choosing the web server with the least number of active sessions. In addition, each router supports a bandwidth query service. This service allows a network administrator to request the bandwidth usage (both incoming and outgoing) for a given router.

1 Introduction

As the Internet matures and more services are required and expected of computer networks new technologies must be explored in order to find the best solutions possible. Three such technologies are exposed here in chronological significance.

1.1 Load-balancing

One technology, load-balancing routers, can offer improved efficiency, security, and scalability. In a typical setup, this router is the only machine exposed on the public Internet and all *real* web servers are behind the router in a private network. When requests for service arrive from the Internet they must be routed through the load-balancer; the router can then choose which server the request will be forwarded to. The algorithm used to select which server is chosen may be as simple as picking the server with the lowest load or it may be more complex. In some cases the load-balancer must inspect the packets in order to know what kind of service is being requested before it can select an appropriate server - this is called packet splicing and leads us to our next technology, content-based routing.

[*] McGill University COMP-535 Computer Networks

1.2 Content-based Routing

Content-based routing is another technology that can be used to enhance a network's features. In the above example, the load-balancing router simply distributed network traffic evenly across a list of servers. In the case of content-based routing, the server selection is reliant upon what content the client is requesting. In the implementation of the router explained in this paper, the content in question is HTTP traffic. When a client contacts the router to request a web page the router splices open the request and chooses a web server which has the specific HTTP content it is looking for. The router then forwards the request to the chosen web server and acts as a middleman thereafter, passing packets from the client to the server and from the server to the client in a *forwarding* mode.

1.3 Router-attached Services

Finally, router-attached services can add extra features and tools to a network. A service is *router-attached* if the router is responsible for providing the service. Historically, routers simply did what their name implies: routed traffic. Now, however, there is a view that routers could do more. For example they could report traffic demographics or collect usage statistics. In the router explained in this paper a service is available which allows an administrator of the router to view the bandwidth usage through the router. This information could be very useful in diagnosing a flaky network or for considering where new upgrades or expansions are needed.

2 Related Work

2.1 Load-Balancing

Many large content providers, such as `www.cnn.com`, `www.ebay.com`, and `www.nytimes.com`, must handle thousands of requests or more a second. Since this sort of load could quickly cripple a single server the solution is to balance the load across multiple servers.

One common approach to this end is through manipulation of DNS tables. By listing a single web address that maps to multiple server IPs, one can automatically load-balance in a primitive way. The DNS server simply cycles through the list of IPs such that *request1* would receive a *IP1*, *request2* would receive *IP2*, and so on. When the list of IPs is exhausted, the DNS server simply starts from *IP1* at the top of the list.

An alternative DNS load-balancing scheme is described in RFC 1749 [McCloghrie et al. 1994] but it relies on modifications to the original design of the DNS system and is not particularly efficient nor easy to set up or maintain.

Another approach to load-balancing is to place the burden of balancing on the routers themselves. This is a logical approach since the routers are already performing some direction on the traffic. By adding a table to the router which holds the available server IPs, a router can make decisions in a more efficient way than the simple DNS load-balancing approach - the router can keep track of how many requests have been sent to a given server and thereby route the latest request to the server with the least requests thus far. If the router is capable of maintaining state awareness of TCP and other stateful protocols, a session table can be kept which records which server has the least *open* connections - arguably a better choice to route the next request to.

A number of papers have been written to this end, both in the hardware and software realms. Cardellini et al [Cardellini et al. 1999] discuss a number of different load-balancing schemes. They outline several categories of load-balancing based on where the balancing is performed. Solutions such as client-side proxy routing and web-browser routing are performed at the client side. Another category is dispatcher-based approaches, in which a centralized *dispatcher* machine acts as the scheduler and balancing is achieved through HTTP redirection (see [Berners-Lee et al. 1996]) or packet rewriting.

A later paper by Cardellini, et al [Cardellini et al. 2000] proposes a geographic-based load-balancing system in which content-caching clusters are strategically placed throughout the Internets so that users can connect to servers that are very close by.

Finally, hardware-level solutions have been explored. Apostolopoulos, et al [Apostolopoulos et al. 2000] propose a hardware-based content router implementation which provides far better performance than software solutions.

2.2 Content-based Routing

Historically the Internet has been a generic medium in which very little high-level application-specific routing is performed. Most common algorithms and protocols in place at the routing level are designed on the lower layers of the OSI Network Model - that is, from the transport layer down. These protocols meet a wide range of requirements and are not designed with any one service or application in mind. Nevertheless, times they are a changin'. As technology changes and the Internet becomes more advanced, more and more research is being done in the realm of content-based routing and there is good argue Ment to suggest that these approaches could be useful in overcoming some of the inherent limitations of the "best-effort" inter-networks we deal with today.

Shin, et al [Shin et al.] explain a system of *differentiated services* in which varying levels of quality of service can be maintained. Their choice example is that of streaming video content. Carzaniga, et al [Carzaniga et al. 2003] present a combined broadcast and content-based routing scheme (CBCB). In this scheme the content-based routing layer uses a "push-pull" method for propagating routing information - the "push" part comes when routers periodically advertise themselves and the "pull" part comes when a router sends requests to it's neighbours in order to update it's routing table. Though their approach is much more complex then ours, there are many similarities between our approaches.

2.3 Higher-level Routing protocols

Without a distributed content-based routing algorithm, the flexibility of the system is somewhat limited. Therefore many content-based routing solutions come hand in hand with some sort of higher-level routing protocol. These protocols may or may not have some resemblance to lower level routing protocols such as RIP (Routing Information Protocol) distance vector [Hedrick 1988] and OSPF (Open Shortest Path First)

[Moy 1989]. A commonly shared feature is the initial setup of routing information which pertains to the building of a "map" (at each router) of all the routers which implement the protocol and which, therefore, can answer content-based routing requests. The differences arise when the actual content is considered for final routing. Depending on the particular application, the routing algorithm could be very complex. It might even map single clients to multiple servers or vice-versa. These sorts of features lead to routing that is highly specialized but potentially more efficient.

Our particular content-based routing protocol, HRT (HTTP Request Traffic), is fairly similar to BGP (Border Gateway Protocol) [Lougheed and Rekhter 1990], in fact. BGP uses *OPEN* messages to create connections between peers. This connection is held open for a set amount of time and it is expected that *KEEPALIVE* messages will be generated before this timeout. Furthermore, BGP makes use of a marker field to authenticate incoming messages. Our HRT protocol broadcasts messages to create and tear down peer connections in a similar fashion to the BGP connections and also incorporates timeout and keep-alive mechanisms. HRT also makes use of a unique message number sent in every HRT packet to distinguish unique directives and avoid cycles.

3 Implementation

The router explained in this paper has been built as an extension to the GINI router [Maheswaran]. GINI (*GINI Is Not the Internet*) is an experimental toolkit for constructing user-level micro Internet simulations. GINI provides tools to simulate arbitrary network topologies complete with switches, routers and wireless components. It is the ideal tool for testing network code quickly under many different circumstances before deployment. The ability to rapidly test code without having to deal with hardware and physical network infrastructure has opened up a whole avenue of possibilities, and was key in making this work possible.

3.1 Load-Balancing

The load-balancing [1] portion of the router has been implemented in tandem with the content-based routing mechanism. We feel that our approach is more flexible than the initially suggested design of simply load balancing across several servers. Instead, all incoming sessions are matched against regular expressions from a user-created table mapping regexes to IP addresses. If a match is found, the incoming TCP session is spliced to establish a connection between the requesting client and the matching server. The regular expressions provide more flexibility in the deployment of the load balancer, without sacrificing too much performance. One can still do straight load-balancing by using:

```
.*
```

(i.e. match everything) as the regular expression mapping to a list of IP addresses.

Each HTTP session that is initiated through the router is recorded in a hash table with it's hash key being formed from it's unique tuple $\langle$IP source, TCP source port, IP destination, TCP destination port$\rangle$. The IP destination in this tuple is the web server that the router chooses to forward the HTTP request to. An additional table is kept in the router which lists all web servers seen so far and the number of currently active HTTP sessions the router is forwarding to each web server. The load-balancer simply spins through this table to find which web server (of those allowed by the content-based routing mechanism) has the lowest number of active sessions and then assigns the new HTTP request to this web server, incrementing the active session count for that server by one. The active session count for a server is decremented when either an HTTP session is explicitly torn down by the client or server, or if a session is timed out by the router. The state that is kept for each spliced session is cleaned up only when the session times out or when the session hash table is full. Since our data structures are constant-sized hashes and tables, this "lazy" cleanup technique suits us well.

TCP timeout is normally two times the maximum segment life (MSL). RFC 753[Postel 1979] defines the MSL to be 120 seconds, however this makes for a lengthy timeout of four minutes. RFC 753 was written in 1979 and networks now behave very differently. A design decision was therefore made to adopt the default MSL of the FreeBSD operating system, which is 30 seconds. We felt this was a more appropriate for our router.

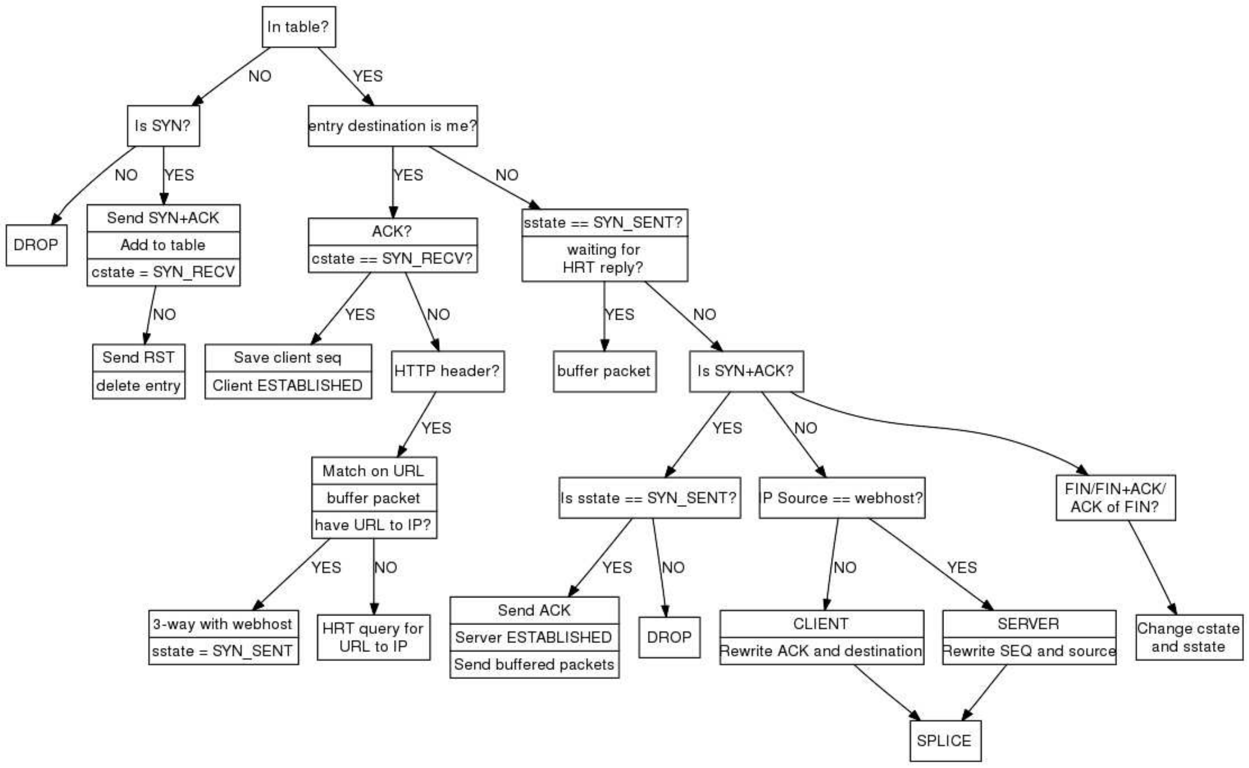


Figure 1: *Load balancer state diagram showing the typical life of a packet traveling through the load balancer.*

It should be noted that the load-balancing scheme that has been implemented is effective even in the presence of multiple load-balancing routers. In the worst case scenario, two routers both initiate their web server tables to have zero active sessions for all servers. If identical requests come to both routers at the same instant, they will both route the requests to the same (first-matching) server. This is not so bad, though, when the session counts reach into the hundreds and possibly thousands - that is, the first-matching server will only ever have, at most, $n-1$ extra active sessions across all routers than any other server, where n is the (relatively small) number of load-balancing routers in the network.

3.2 Content-Based Routing

Content-based routing is performed on HTTP traffic in this router through the use of TCP splicing and forwarding as described in Spatscheck, et al [Spatscheck et al. 2000]. To accomplish this, all TCP packets that are sent directly to the router are examined. The router then responds to clients who are trying to initiate a new session using the the normal TCP three-way handshake as described in RFC 753[Postel 1979].

Once a session is properly established, the router waits for a client HTTP request packet as described in RFC 1945[Berners-Lee et al. 1996]. Specifically, the first line of data in an HTTP request packet is a request line. The request line starts with a method token (e.g. GET or POST) followed by a uniform resource identifier (URI) and ends with HTTP version. The line is terminated with CRLF. All elements mentioned are separated by space characters (CR and LF not included). This request is matched by using a regular expression of the form:

```
\s*[A-Z]+\s+(.*)\s+HTTP/\d+\.\d+\s*
```

The content that is the basis for routing is the URI. This URI may be something along the lines of `http://www.host.com/documentation/` or `http://www.host.com/cgi-bin/`. Suppose a situation arises in which the network administrator serves all documentation from *Server1* and all cgi scripts from *Server2*. In this case, the administrator would add two content-based routing entries at the router. In this router, content-based routes are specified with regular expressions. This allows for a very powerful and flexible set of routing rules. In this particular example, the two routes could possibly look like this:

- *Server1*

  `http://www\.host\.com/documentation.*`

- *Server2*

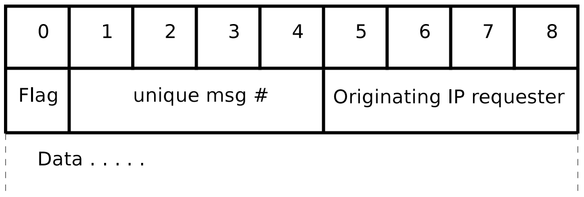
  `http://www\.host\.com/cgi-bin.*`

Each entry that the URI matches in the content-based routing table specifies a list of valid server IPs for that content. Note that, in the example above, each entry specifies only one server but the implementation allows for many.

Once a server has been chosen from the content-based routing table the router initiates a three-way TCP handshake with this server as if the router itself were the client and all packets received on the existing TCP stream from the client to the router are buffered at the router. When the router-server TCP session is established the router switches into forwarding mode and sends all buffered client packets to the server. In forwarding mode, the router simply patches client packets with the server IP and port as the new destination and adjusts it's sequence number to match that of the server's (always re-computing the checksum as well). Server packets are patched to be forwarded to the client in a similar way.

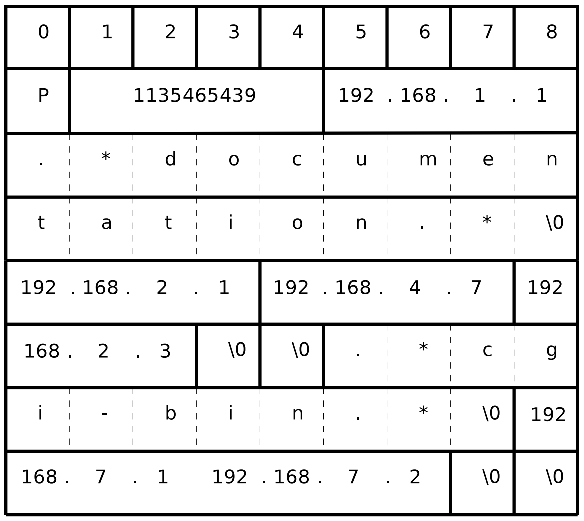
3.3 Distributed Routing Tables

In addition to simple content-based routing, the router implements a unique protocol dubbed HRT (HTTP Routing Table). This protocol, built on top of IP, is used to distribute content-based routing information to routers that may not know what the routing rules are themselves. Unaware routers are called *Edge* routers and those that do have content-based routing knowledge are called *Core* routers (See Figure [2]). When an *Edge* router receives a client HTTP request that it cannot route itself, it sends a numbered HRT request to all of it's direct neighbours and buffers the packet. Note that this buffering is different from the previously mentioned HTTP packet buffering - packets in this buffer can only be cleared when an HRT response packet is received from another router which delivers new routing information that resolves the HTTP packet's URI. In the case where a direct neighbour receives a HRT request which it cannot resolve it simply forwards the request on to it's own direct neighbours (except the original sender of the request). In this way HRT requests will eventually reach all parts of the router network, ensuring that if a content-based routing rule exists which matches the request, it will eventually be reported. If a router sees a HRT request or response packet which it has processed recently (based on the HRT message number) it does not re-process it. This ensures no infinite cycle issues will arise. Manually written entries at *Core* routers are never timed out and never superseded by new HRT information.



0	1	2	3	4	5	6	7	8
Flag	unique msg #				Originating IP requester			
Data								

Figure 3: *HRT Packet Header.*



0	1	2	3	4	5	6	7	8
P	1135465439				192 . 168 . 1 . 1			
.	*	d	o	c	u	m	e	n
t	a	t	i	o	n	.	*	\0
192 . 168 . 2 . 1				192 . 168 . 4 . 7				192
168 . 2 . 3			\0	\0	.	*	c	g
i	-	b	i	n	.	*	\0	192
168 . 7 . 1 192 . 168 . 7 . 2							\0	\0

Figure 4: *Example HRT Response (POST) packet with data.*

The HRT protocol is very similar to the ARP protocol as described in RFC 826[Plummer 1982] except that it is meant to resolve URIs to IPs instead of IPs to MAC Addresses. The main difference is that HRT requests are sent directly to other routers instead of through a broadcast mechanism. This is accomplished by maintaining a table of HRT neighbours at each router. This table is built by the HRT protocol itself. Routers which implement the HRT protocol explicitly announce themselves with a *Hello* packet broadcast on each network interface as it is created. These broadcasts are only

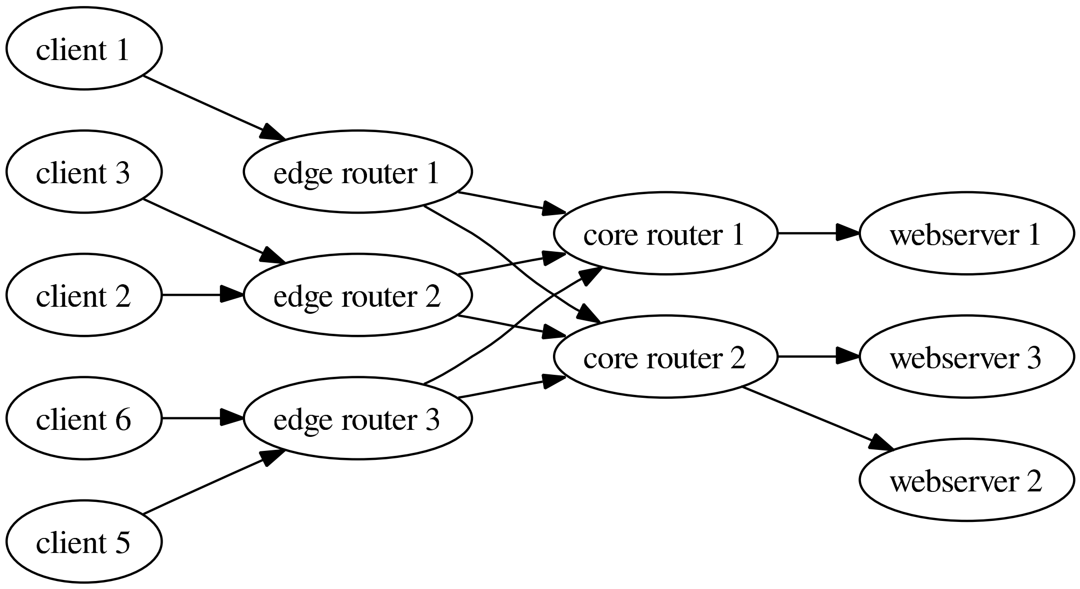


Figure 2: *Typical load balancing topography.*

for that direct physical connection and are not propagated to other networks. All routers which hear a *Hello* packet respond directly to the sender with an *Acknowledge* packet and then add the sender to their tables as a direct neighbour. When the original router receives *Acknowledge* packets it adds the senders of these packets to it's own direct neighbour table. Whenever a network interface is brought down on a router, the router first broadcasts a *Goodbye* packet which tells it's neighbours to remove it from their tables. Since the GINI framework does not yet provide an IP broadcast mechanism we had to implement a primitive mechanism ourselves. This primitive broadcast only works on a single network and therefore multi-network broadcasts are not properly propagated at this time.

Figure [3] shows the basic structure of an HRT packet. Note that *Hello*, *Flush*, and *Goodbye* packets are sent with a single flag byte only. Request ("R") and Response ("P") packets contain a message number unique within the given router and the original requesting IP. Duplicate messages can be detected using their $\{UniqueMessageNumber, OriginatingIP\}$ tuple. Figure [4] shows an example Response packet. Note the flag "P" and the use of terminating characters (\0) to end each regular expression entry. The packet is responding to the original requesting router (192.168.1.1) with the following HTTP routing table information:

- `.*documentation.*`

  ⇒ 192.168.2.1, 192.168.4.7, 192.168.2.3

- `.*cgi-bin.*`

  ⇒ 192.168.7.1, 192.168.7.2

As an upkeep routine, direct neighbours are sent periodic *Hello* packets (non-broadcast) and are expected to respond with *Acknowledge* packets. If a direct neighbour is not heard from within a predefined time span (15 minutes) then that neighbour is removed from the table. Any HRT packet received from a direct neighbour whatsoever, whether it be a request, a response, or a flush command, is sufficient to reset the timer for that neighbour (and therefore act as an unrequested *Acknowledge*). This limits the number of keep-alive message passing that is required within the network.

As an extension of the HRT protocol itself, information in HRT response packets is cached in the same way that ARP routing information is cached. However, this information times out after fifteen minutes to ensure that route information is always as up to date as possible. In the case where a network administrator explicitly changes routing information at a *Core* router, they can then ensure that this new information is quickly adopted by issuing an HRT Flush broadcast which tells all routers in the network to clear cached content-based routing information.

Note that routers can be partly an *Edge* router and partly a *Core* router at the same time. That is, a router may have knowledge about how to route certain content but not all content.

3.4 A Router-attached Service

The implemented router benefits from one additional feature which a network administrator might possibly find useful. Each router, whether *Core* or *Edge*, provides a command to report it's bandwidth usage to it's console. These statistics take the form of a visual graph, built with ASCII [Gorn et al. 1963] characters, which reports the per second bandwidth average for each minute over the last hour. Incoming and outgoing traffic are plotted separately, but on the same graph (see Figure [5]). This service proved indispensable when testing the performance of the router.

```
********** BANDWIDTH USAGE ********
***** MAX (in/out): 32.13 b/s *****
    in = ,  out = .  in+out = ;
***********************************
     Now                30 min ago*
   32   ,,
   30   ,,
   28   ,,
   27   ,,
   25   ,,
   24   ,,
   22   ,,
   20   ,,
   19   ,,
   17   ,.
   16   ,;         ,
   14   .;         ,
   12   ;;,        ;
   11   ;;;        ;
    9   ;;;     ,  ;
    8  ;;;;,    ,  ;         ,
    6  ;;;;;    ,. ;.        .
    4, ;;;;;, . ., ;;        ;
    3,,;;;;;; ,.;;;;;     .  ; ;
    1.,;;;;;; ;,;;;;;     ;  ; ;,,.
```

Figure 5: Bandwidth statistics output by the router in a visual manner - the graph has been vertically truncated to fi t

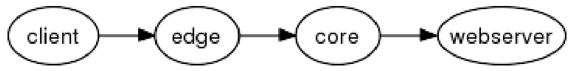


Figure 6: Test topology designed to stress the HRT load-balancing content-based routing protocol. HRT entries were set to time out quickly after being updated so that the *Edge* router would continuously send HRT requests for the content to web server mapping.

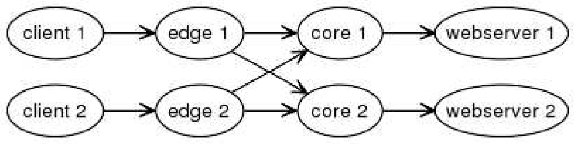


Figure 7: Test topology used to determine throughput. Here the two clients were set up to continuously request different content from their respective *Edge* routers which would map, through the two *Core* routers, to the different web servers.

4 Performance

Routers are often small and low-powered devices. For this reason, it is desired to have very efficient algorithms implemented on the router. We were especially cognizant of this fact throughout the development of the router. For example, very efficient data structures were used for storing information at the router. The session table, for example, is a hash table of hash tables where the keys are the client's TCP port and the server's TCP port. This allows for very quick session lookup. Wherever possible, preference has been given to hash-based, constant-memory data structures over dynamic, sorted data structures, conserving precious router memory and increasing throughput. Additionally, the router snoops HRT responses passing through it destined for other routers. This allows the router to keep fresh content-based routing information and, possibly, to limit the amount of HRT requests it will itself be required to send in the future. Finally, network byte order is used whenever possible when storing packet header information. For example, all IPs stored in data structures on the router are stored in network byte order. This limits the amount of byte flipping that has to be done by the router.

Unfortunately, extensive testing of the router was not possible within the confines of the GINI framework due to the use of blocking message queue calls. Specifically, the GINI router passes messages between it's layers using inter-process communication (POSIX IPC) message queues using blocking sends and receives. When there is heavy message traffic between the router layers the router will eventually deadlock with some threads stuck in the *msgrcv* function and some threads stuck in the *msgsnd* function. Detailed analysis using *gdb* [The GNU Project Debugger] and further research into the problem suggests a number of possible problems not least of which may be the fact that the message queues are possibly running out of space due to an extremely high number of "in-flight" messages as the router load increases. Because our work was done in a shared lab environment we were without root level access on the machines we were using. Setting the maximum message queue size is possible through the *msgctl* function but this requires root access and therefore we were unable to test message queues of larger sizes than the default. Various attempts were made to make no-blocking *msgsnd* and *msgrcv* calls but to no avail. Finally, as the project deadline loomed near we were forced to abandon any hope of producing stress-testing data on our newly created content-based load-balancing service-attached router.

Nevertheless, various network topologies were set up (see Figures [6] [7]) and tested. In our testing, one or more user-mode linux (UML) machines were initialized as web servers by running the command "/etc/init.d/httpd start". A fairly large HTML file was placed in /var/www/html to act as content. The Linux command line utility NetCat was used on other UML machines to request content from the routers themselves. The routers were set up with various content-based routing tables.

All portions of the router were successful. The routers built up and tore down HTTP sessions in a similar manner to a normal TCP session - sessions were properly timed out after the allotted 2*MSL time period after final RST packets were sent to both client and server. The splicing of HTTP data requests generated content-based routing lookups which, in turn, triggered HRT request packets to be flooded to the direct neighbours of the requesting router. The HRT content-based routing protocol performed well and as expected, passing routing information back to those routers making requests. Packets were successfully buffered for open sessions while waiting for an HRT response and these packets were properly forwarded once a webs ever was chosen and it's TCP session with the router was initialized.

5 Restrictions

5.0.1 Packet flow

It should be noted that in order for the load-balancing algorithm to work properly the client machine must be on one side of the edge router and the server on the other with an assurance that packets cannot be routed around the router. If this were not the case, the server could potentially deliver packets directly to the client or through a path not including the edge router as an intermediate step without giving the router a chance to rewrite the packets, which would be delivered with incorrect IP headers. These packets would leave the server and client TCP stacks significantly confused, and would cause incorrect state to be kept at the router. This state would eventually time out. It is therefore vital that the edge router remain as forwarding agent throughout the life of a content-base routed TCP session.

5.0.2 HRT packet loss

The HRT protocol is built on top of IP datagrams and is therefore susceptible to packet loss. There is no mechanism in place to guarantee that HRT packets are delivered uncorrupted, or indeed delivered at all. Data corruption in Request and Response packets could lead to content being matched with the incorrect server. The effects of packet loss in Hello packets can be neglected since Hello packets are retransmitted periodically. Packet loss in Request or Response packets could lead to a request for a URL to IP matching never being filled even though there exists an edge router on the network that can fill the request; this would lead to the eventual timing out of the initiating session.

6 Future Work

Although all the design goals of the content-based load-balancing service-attached inter-networking router were implemented in full, there are many related ideas that warrant investigation. Certainly, one potential avenue for future work is achieving actual performance data. The obstacle here is the message queue deadlock which must be resolved before the router can handle any real throughput. One potential solution would be to increase the kernel-set maximum message buffer size beyond the size possibly needed to store all messages passed by the router at any one given time. This parameter would of course vary on machines and architectures with different specifications. Another possibility would be to perform a detailed analysis of the contents of the message queue at the point of deadlock. We have ascertained that deadlock occurs when all threads are either stuck in a layer send due to the message queue filling up, or in a receive where no messages of the required type are available in the queue. It remains to see what type of messages do fill the message queue. With high probability there is a module that is stuck in a send, or in a receive for messages of a certain type while the queue is filled with messages destined for the same module, but of a different type. Another potential avenue would be to discard IPC for shared memory or sockets for communicating between the router threads, although this is susceptible to similar deadlock vulnerabilities.

One interesting idea that we had while working on this project is a UML-to-Internet gateway. This would create the possibility of designing a complete network simulation, and connecting it to the Internet. Possible applications would include honey pots, hosting services and public “network sandboxes” where anything goes. The implementation of such a scheme is possible because each router and each UML is running as one or several client processes on the host machine. Therefore, these processes should be able to negotiate connections with the Internet in the same fashion as any other user process on the host machine. Routing GINI packets out onto the Internet would be restricted to certain protocols: root privileges on the host machine are required to modify raw Ethernet frames. Therefore, an unprivileged router can only act as a gateway for packets which any unprivileged process can create: UDP, TCP and possibly ICMP packets.

A GINI-to-Internet router would have a route entry for all destinations that are “on the outside”. Incoming packets matching that entry would be examined, and unsupported protocols would be rejected (or dropped). Supported packet types would have to be sanitized and rewritten for transmission. In the simpler UDP (datagram) case, the router would open the socket and simply transmit a UDP packet of its own which has the same flags and payload. Figuring out which UDP port the client expects a reply on would be trickier, and would probably require user intervention. TCP sessions could be handled with a splicing method similar to the approach described for load-balancing. The difference here would be that the outgoing/incoming end on the Internet “interface” of the gateway could rely on operating system facilities to simplify opening and closing connections and transmitting data. We are aware that the UML includes several different UML interfaces that offer functionality similar to what we have described. Efforts could be made to extend and simplify the interfaces already offered by UML.

Another avenue for future work would be to make the HRT protocol withstand packet loss and corruption. This could be accomplished by storing HRT packet hashes/checksums and sending messages acknowledging the receipt of HRT requests and replies.

7 Conclusions

In this paper we have presented a content-based load-balancing service-attached inter-networking router within the GINI [Maheswaran] framework. TCP splicing and forwarding was implemented in roughly the same manner as in [Spatscheck et al. 2000] which allows the router to perform content-based routing. A novel and powerful regular expression approach to content request matching was described which allows for a very flexible routing scheme. A unique protocol, HRT, was built on top of IP which allows for passing of content-based routing information among routers in a similar manner to MAC address routing with the ARP protocol. Router-based load-balancing over a number of servers is explained as a more secure and efficient approach than other schemes such as DNS looping. Finally, a useful feature was added to the router which allows for the reporting of bandwidth usage statistics.

8 Appendix: Running the Router

The router discussed in this paper extends the base source code of the GINI router [Maheswaran] and therefore the GINI documentation should suffice for information not related to the content-based routing, the load-balancing, or the router-attached bandwidth reporting service.

Load-Balancing

The load-balancing performed by the router requires absolutely no setup by the administrator. If multiple servers are available that serve a given content request, then traffic will be balanced in a session by session manner across these servers.

Setting up the Content-Based Route Table

Content-based routing is performed by matching content requests (URIs) to regular expressions. These expressions are mapped to one or many servers which offer the requested content. To add an entry in this content-based routing table issue the following command:

- htable **add** regex ip1 [ip2 ip3 ...]

It should be noted that the regular expression should have no spaces in it because the command line interface of the router will parse it as multiple parameters to the add command. Also, incomplete IPs are interpreted by filling in zeros where numbers are missing. Thus, “192.168.” will be interpreted as “192.168.0.0”. The following is a sample output of the “htable **show**” command:

```
=========================================
      H T A B L E
-----------------------------------------
  RegEx         IP Address   Timeout(secs)

  [0]  .*documentation.*         0
              192.168.1.222
              192.168.1.223
  [1]  .*cgi-bin.*               0
              192.168.1.30
-----------------------------------------
      2 entries found.
```

Timeouts of zero denote entries which have been explicitly added to the table by an administrator. Timeouts greater than zero denote cached routing information that will eventually be purged when it’s timer runs out. Also, note that there are index numbers listed for each regular expression. IPs can be added or removed from a given entry by referencing the entry’s index number when issuing the “htable **add**” or “htable **del**” command as follows:

- htable **add** -i ‘regex index number’ ip1 [ip2 ip3 ...]

If no IPs are listed with a “htable **del**” command then the entire entry is removed from the table. Two extra show commands, “htable **shows**” and “htable **shown**”, are made available for printing the number of active sessions for each server and for showing the HRT neighbours of the router, respectively. Finally, there are two commands made available which generate HRT traffic. The first, “htable **flush**”, causes the router to clear all cached routing information and to also send a HRT flush packet to each of it’s neighbours. The second command, “htable **send**”, is primarily for debugging and testing purposes and can be used to generate a HRT request packet as follows:

- htable **send** URI ip

Note that the IP given in this command is used as the originating requester IP (i.e. the router which will receive the HRT responses).

Requesting Bandwidth Statistics

Bandwidth statistics can be reported for the last hour by issuing the following command at the router console:

- htable **bandw**

References

APOSTOLOPOULOS, G., AUBESPIN, D., PERIS, V. G. J., PRADHAN, P., AND SAHA, D. 2000. Design, implementation and performance of a content-based switch. In *INFOCOM*, 1117–1126.

BERNERS-LEE, T., FIELDING, R., AND FRYSTYK, H., 1996. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May.

CARDELLINI, V., COLAJANNI, M., AND YU, P. S. 1999. Dynamic load balancing on web-server systems. *IEEE Internet Computing 3*, 3, 28–39.

CARDELLINI, V., COLAJANNI, M., AND YU, P. S. 2000. Geographic load balancing for scalable distributed web systems. In *MASCOTS*, 20–27.

CARZANIGA, A., RUTHERFORD, M., AND WOLF, A., 2003. A routing scheme for content-based networking.

GORN, S., BEMER, R. W., AND GREEN, J. 1963. American standard code for information interchange. *Commun. ACM 6*, 8, 422–426.

HEDRICK, C., 1988. Routing Information Protocol. RFC 1058 (Historic), June. Updated by RFCs 1388, 1723.

LOUGHEED, K., AND REKHTER, Y., 1990. Border Gateway Protocol (BGP). RFC 1163 (Historic), June. Obsoleted by RFC 1267.

MAHESWARAN, D. M. Gini is not the internet, a toolkit for constructing user-level micro internets.

MCCLOGHRIE, K., BAKER, F., AND DECKER, E., 1994. IEEE 802.5 Station Source Routing MIB using SMIv2. RFC 1749 (Proposed Standard), Dec.

MOY, J., 1989. OSPF specification. RFC 1131 (Proposed Standard), Oct. Obsoleted by RFC 1247.

PLUMMER, D., 1982. Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware. RFC 826 (Standard), Nov.

POSTEL, J., 1979. Internet Message Protocol. RFC 753, Mar.

SHIN, J., KIM, J., AND KUO, C. Content-based packet video forwarding mechanism in differentiated service networks.

SPATSCHECK, O., HANSEN, J. S., HARTMAN, J. H., AND PETERSON, L. L. 2000. Optimizing TCP forwarder performance. *IEEE/ACM Transactions on Networking 8*, 2, 146–157.

THE GNU PROJECT DEBUGGER, G. Gdb: The gnu project debugger.