An Extensible and Adaptive Framework for Load Balancing using Multicasting

Jens Vesti
Christian Theil Have
IT University, Copenhagen
{vesti,cth}@itu.dk

Supervisor: Kåre Jelling Kristoffersen

January 16, 2004

Abstract

Various load balancing solutions are available today. Most of these suffer from several issues such as single point of failure or merely the high prices that comes with specialised and proprietary solutions. We propose and implement a framework for load balancing solutions that eliminates the single point of failure. The framework is flexible and enables programmers to adapt the load balancing to the exact requirements a such may have. The framework enables the programmer to create single system image clusters, which to the outside world looks like a single host. We use multicasting in order to distribute the packets to all cluster hosts. These are hereafter filtered on each cluster such that only one host ends up receiving the packet. The framework is tested and demonstrated using two example modules. The tests performed proves the concept and shows a performance advantage of using the framework.

Contents

1 Introduction

1.1 Motivation

E-systems and web-services in general have become popular over the last decade. For some companies and organisations it has proven to be a major problem keeping up with the demand - Cardellini states in [CCCY02] that the number of online users is increasing with 90 percent per annum. If a system proves to be popular the number may even increase more than 90 percent. It is therefore essential to keep up with the demand in order to continuously provide quality service.

As the demand for web-services increase, so does the requirements to the server-side technology responsible for running and delivering these web-services. Furthermore, the computational implications of running such services has increased. Content are dynamically created by programs, often communicating with other systems. Large scale web-services does not run on a single server anymore, they run expensive platforms that include web servers, database servers and application servers etc.

There is a variety of techniques for scaling web-services to meet the demands. Our focus is on the techniques that distribute requests to several servers. Common for the solutions in this area is that they are not that flexible when it comes to handling different kinds of requests or adapt the requirements of the organisations. Some organisations require a fault-tolerant web service while other kinds of web requests require a fast, but not necessarily a fault-tolerant, service.

1.2 Aims

The intention of this report is to present the work done in creating a flexible and extensible framework for implementing load balancing algorithms. To justify the need of such a framework we try to give the reader a brief understanding of the problems in the field of scaling and load balancing. We introduce the existing solutions and current research in this area along with the issues these may have.

1.3 Methods

The work in this report will be carried out with respect to the principles of experimental computer science and in particular the proof-of-concept and to some extend the proof-of-performance [SBC+94]. Using the proof-of-concept we can demonstrate a concept by assembling a number of well known techniques and tools in a new form in order to fulfill a new purpose or set of objectives. The proof-of-performance can be used to evaluate a given phenomena, such as any of those used to assemble the concept we wish to prove.

1.4 Overview

The approach in this report is first to describe the basic terminology used in this field of study and research. We do this in order to create a common understanding and foundation for the rest of the report.

Secondly we briefly review the related load balancing techniques and proprietary and non-proprietary solutions available today. This is done as we wish to get an understanding of the issues associated with these.

Derived from the brief review we hereafter present the problem statement and the architecture of our solution, and finally the implementation and evaluation.

2 Basic Terminology

There is a number of basic concepts and terms that need to be understood in order to comprehend the issues and solutions given in this report. We will in the following describe these basic concepts and terms briefly, and where relevant, the issues related to these.

2.1 Scaling

Scaling means in general to adapt something to a given scale. In the case of scaling web servers we adapt the size of the servers to the load expected. Scaling a web service can be done in various ways. In this section we will in general terms present the techniques available, how they work, what the benefits and limitations are. We do this in order to get an understanding of where our solution fits into the greater picture. In Figure 1 is depicted an overview of the different disciplines in scaling Web server systems. The figure which was originally presented in [CCCY02] shows a number of boxes each representing a scaling technique or a superset of scaling techniques. Each scaling technique is described in details in the following sections.

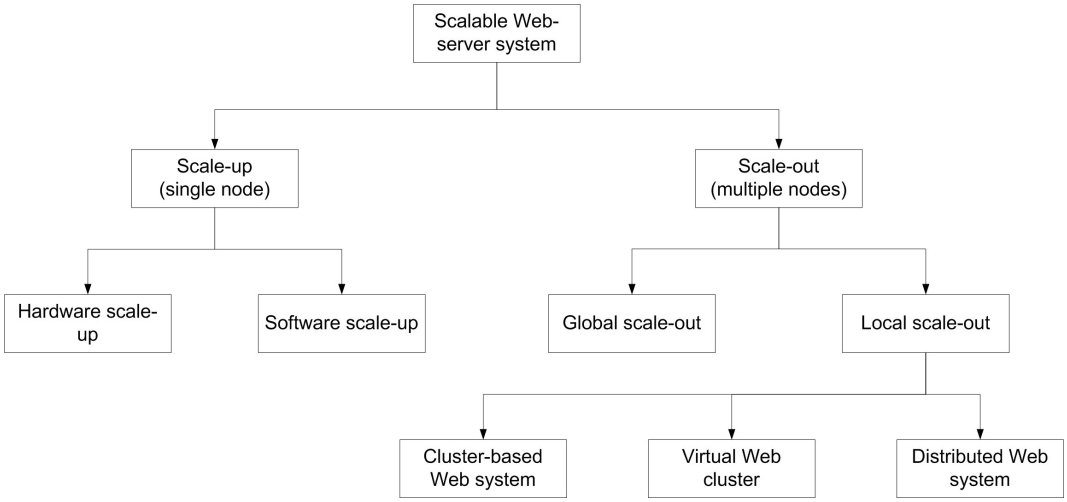

Figure 1: Scaling techniques in Web-systems [CCCY02]

2.1.1 Hardware scale-up

Changing and upgrading the hardware is referred to as a *hardware scale-up* [DGLS99, CCCY02]. This approach is often one of the first techniques to apply when scaling the system. Obviously there

is a limit to how often this approach can be taken as the increase in computational power of a single server is low, compared to the number of online users which in average is increasing with 90% per annum [CCCY02]. Given these figures, hardware must be changed on a regular basis when this approach is being used. Some providers may lease the hardware in which case a hardware scale-up is relatively cheap. But in the case of a company owning the hardware the replaced hardware is made redundant unless there is another place for it in the company or a *scale-out* approach is being taken as described in Section 2.1.3.

2.1.2 Software scale-up

Upgrading or changing the software in order to achieve improved performance is often called a *software scale-up* [CCCY02]. The term software covers in this case operating systems, Web servers (eg. Apache or Microsoft IIS), databases, and services developed in-house etc. Just as with hardware scale-ups there is a limit to how often this can be done. This because of the limited possibilities of changing software and the costs of doing this. It is however a better solution over time, compared to a hardware scale-up, as the changes may follow whatever other scaling techniques being used later on. It does however often only scale on a smaller basis, depending on the service being scaled. If well designed and optimised services, such as operating systems and Web servers, are being upgraded the system cannot be expected to scale-up well. On the other hand if poorly developed services are upgraded it may very well scale very well.

2.1.3 Scale-out

Clustering is an interesting area that has much research focus. The technique is also called a *scale-out* and means in general expansion of capabilities by adding nodes [DGLS99, CCCY02]. This implies that the system is distributed or replicated over several hosts in order to provide high availability and performance. The term scale-out can furthermore be split up into a local and global scale-out.

Local scale-out is when the hosts are kept within the same network on the same geographical location. Cardellini et al identifies three types of Web cluster systems in [CCCY02]. With cluster we mean a group of servers that cooperate to act in a transparent way as a single identity to a client.

1. The *cluster based Web system* has a routing or load balancing device in front of the cluster to distribute to traffic to each cluster host. The cluster hosts are transparent to the client as they cannot see the IP address of each host but only the front-end devices.

2. The *virtual Web cluster*, first given in [CCCY02], covers clusters where the *virtual IP address* is the only visible address to the clients. This approach is particularly interesting because it has the ability to remove the single point of failure, that is common for most scale-out techniques, by moving the dispatching to the cluster hosts themselves instead of using a dedicated load balancing switch. It is furthermore interesting because the group communication

used is suitable for fault tolerance in terms of availability and to some extend reliability. A virtual Web cluster is a stand-alone cluster that works independently and with no front-end node to act as a dispatcher, that is the node who makes decisions whom to hand out the packets. This approach is a subset of *layer four switching with layer two packet forwarding* described in [SSB00]. The research in this area is sparse and limited to a handful of articles and technical reports [VC01, nlb00, Gre00, wlb99, CCCY02, SSB00, DCH+97]. We will in Section 3 review the research done in particular this area.

3. The *distributed Web systems* are some of the oldest cluster based Web systems. The dispatching is done outside the cluster which makes each host within in the cluster visible to the client. An example is the DNS Round Robin (DNSRR) solution described in Section 3.1. The dispatching granularity is very coarse and various issues in this approach has been identified [STA01].

Global scale-out is when cluster hosts are spread out over multiple geographically distant locations. This concept can be used whenever local scale-out is not longer sufficient because of a single location becoming a bottleneck [CCC01]. Global distributed clusters usually dispatches the load on different levels. Most commonly is coarse grained dispatching on DNS level using eg DNSRR described in Section 3.1 and fine grained on the Web cluster level using same methods as in local scale-out solutions [CCC01].

Other solutions may be used. One of the interesting methods is using the group addressing paradigm *anycasting* presented in Section 2.3.4 which makes the use of global scale-out techniques transparent.

2.2 Fault Tolerance

One of the issues today with Web services are the numerous failures a client may experience. The reason for these failures are many, ranging from software failures to hardware failures and again to a simple lack of performance. What is needed to avoid these failures, and what is being provided, to some extend, using various solutions, as those discussed in Section 3, is *fault tolerance*.

Fault tolerance is simply a way to incorporate redundancy to mask failures in the system [Jal98]. This can be done in various ways and on different levels. We will in the following describe the central terminology that relates to our focus in this report.

2.2.1 Reliability and Availability

Reliability and availability is often mistaken or used interchangeable, but do in fact have different meanings. *Reliability* can be expressed as the probability of the system still working at a given time in the future. Often reliability is presented as *mean time to failure*. *Availability* can be expressed as how often the system is available. This can be presented in percentage for example as *up-time* or *down-time*. It should be mentioned here that availability is different from reliability in the sense that a system can be highly available but erroneous, that is, not reliable. One way to go for higher

availability and reliability is replication. By replicating the services, one service may take over when another one fails.

2.2.2 Failures, Errors and Faults

There are several reasons why the system may be unreliable or unavailable. In order to fully understand how we can improve reliability and availability we first need to identify and get an understanding of what causes these conditions.

The terms failure, error and faults are often mistaken for one another but they do have different meanings. *Failures* occurs when the service does not behave as specified [Cri91b]. Failures can be said to be the outcome of errors and errors occur because of faults.

Failures in particular can be classified into following groups [Cri91b].

Omission failures happen when a service omits to respond to a specific input.

Timing failures occur when the right response is received but with the incorrect timing. This can be either as an *early* or *late* timing failure, that is the response can arrive earlier than expected or later.

Response failures are responses that do not match the expected result. This can either be in terms of *value failures* or in *state transition failures*.

Crash failures occur when the service does not respond to the requests until it has been restarted. The first failure occurring would be seen as an omission failure as it is a single failure, whereas the succeeding failures are crash failures. Depending on the state the service is in when it restarts we can further classify the crash failure into *amnesia-crash* where the service returns to a predefined state, *partial-amnesia-crash* where parts of the state before the crash is kept, *pause-crash* where the service restarts with the state the service had before the crash, and *halting-crash* where the service never restarts.

2.3 Group Communication

Because group communication is one of the central topics here it is ideal to shortly introduce the basic concepts.

2.3.1 Group Structure

There are two categories of systems supporting group communication, the *open groups* and the *closed groups* [Tan95]. *Open groups* permit any process to send to the group, including processes that are not members of the group. With *closed groups* only processes which are members of the group can send to the group, and non-members cannot. Whether groups should be closed or open depends on the purpose of the system.

2.3.2 Group Membership

There are two types of group membership, the static and the dynamic membership [Tan95, KT92]. This is relevant in relation to what we are trying to accomplish here because membership handling can be complex to handle. It must be done *on the fly*, that is it must be transparent to the client, he cannot notice delays or denial of service while the membership state is changing.

Static membership is the simplest form of membership to manage of the two types. By static we mean that it is not possible to change the state of the group as long as the group exists. Changing the state means to let processes *join* or *leave* the group. Most systems, however, do need to change their state over time, although most slowly and rarely [Bir96].

In contrast to a static membership we have the *dynamic membership* where processes may join and leave the group whenever they choose. This gives us a group that also shrinks with failures as well as grows with joins [Cri91a]. The complexity in handling this kind of group is high as a failed process per definition has left the group, but cannot signal this information because it has already failed. There are three ways of handling this complexity and keep group membership information updated in the group as described in [Cri91a], which is presented in a simplified way below.

Periodic broadcast membership. All processes broadcast when they are willing to join the group, just as they keep broadcasting messages telling the group that the member is still present. We will refer to these as *heartbeat* messages [ACT97]. In the absence of one or several heartbeat messages the group can presume that the member has left the group. The problem with this approach is the number of broadcasts being sent with a fixed time interval may flood the network. Another issue is whether the broadcast can be considered a reliable form of communication. If some broadcasts do not arrive in time at destination, or at all, it may be difficult to tell whether the process has left the group or not.

Attendance list membership. A process may join the group in the same way as with periodic broadcast membership, but after joining the group state is maintained passing an *attendance list* around a *virtual ring*. If a process fails it will be noticed when relaying the list among the members.

Neighbor surveillance. This approach works in the same way as the above described approaches, except the way failures are detected. A neighbor is being monitored by its successor, and in the case of failure a *regroup message* is being broadcast.

For all the approaches described above applies the *regroup message* meaning that in the case of a process leaving the group, eg because of a failure, the group must reorganise.

2.3.3 Message Delivery and Response Semantics

There are four *message delivery semantics* we need to address [KT92]:

1. *Single delivery semantics* is used when only one of the group members are required to receive the message.
2. *k-delivery* is used when at least k members of the group must receive the message.
3. *Quorum delivery* is when the majority of the group receives the message.
4. *Atomic delivery* is the most difficult to manage as *all* hosts within the group requires to receive the message.

There are five categories of response semantics [KT92]:

1. *No response* is when no response is sent back to a message request. This implies an unreliable communication.
2. *Single response* is when exactly one host within the group responds to a request.
3. *k-responses* is when k responses are sent back from the group.
4. *Majority response* is when a majority of the group hosts responds to a request.
5. *Total response* is when all hosts in the group must respond to the message received.

When we use the term group we mean the current group.

2.3.4 Group Addressing

The different group addressing paradigms used here are relevant in order to understand how fault tolerance is provided in Web cluster systems. We will in this section describe the terms uni-, broad-, multi- and anycasting in relation to Web system clustering and fault tolerance.

Unicasting is a *one-to-one* communication paradigm which in our case relates poorly to our project in terms of reliability as it is costly to address multiple servers at once in order to provide replication.

Broadcasting is a way of addressing all hosts on the network, it is a so called *one-to-all* or *all-to-all* message approach. It is in general unreliable because of the possibilities of receive and omission failures but is often sufficient [Bir96]. The problem with broadcasting is that not all hosts on the network may be interested in the message sent, but they receive everything anyway. This is

an overhead that can be avoided using the multicast approach described next.

Multicasting is a way of addressing, and delivering datagram by best effort to several hosts on the network, it is therefore a *one-to-many* approach. The way it works is that a multicast address is specified and messages sent to this address is being received by those subscribing or listening to it. As with broadcasting, unreliable delivery is often sufficient. There is however heavy research done in the area of reliable multicasting.

Anycasting has the purpose of delivering, by best effort, an anycast datagram to at least one host, and preferable no more than one host within the group as originally described in [PMM93]. So it is fair to say that anycasting is a *one-to-any* communication paradigm. The way anycasting is intended to work on the Internet is that hosts who wishes to join an anycast group registers themselves as being members of this group. Then whenever a client sends out a datagram destined for an anycast group a decision on to which group member the datagram should be sent to is taken.

This approach is interesting in terms of a webserver global scale-out. Depending on which algorithm is used to find the group member and forward the datagram to, it is useful to locate resources, and as every packet forwarding need a lookup in the routing table the problems experienced with DNSRR, as described in Section 3.1 are not present here.

In contrast to multicast addresses in IP version 4 and 6, the anycast addresses are indistinguishable from unicast addresses. It is transparent to the client, and the communication can be done just as with unicasting.

The downside to anycasting is that it is stateless, and therefore connectionless. The original specification [PMM93] and the IPv6 specified in [DH95] describes a method to create a stateful connection using TCP. Basically a client tries to establish a connection to an anycast server which changes the address from an anycast to the unicast address of the server.

Other limitations that need addressing would be the high communication costs between members within the anycast group when distributed over at slow WAN in comparison to a fast LAN. This would effectively mean less communication across the WAN and make the group inflexible. This issue naturally depends of the nature of the service provided. If the need of up to date redundancy is not high the communication between the anycast group members therefore can be proportional lower.

Most research in anycasting has been focused on the network layer, which does not apply for anycasting on a LAN. A proposal in [BEH$^+$97] describes a method of addressing a single host in the LAN. It basically sends out an ARP request to retrieve the physical address of one of the anycast group members. The first to reply is the host to forward the datagram to. This technique is insufficient as all subsequent datagram all are sent to that member and none to the other members.

3 Related Work

There is limited research done relating to what is the intention of this project. In this section we will survey some of the research projects that have been carried out. The survey will reflect the topics presented in Section 2.3 and 2.2. The main focus is on virtual Web cluster systems as these have a potential to be more reliable and available that other scaling techniques as presented in Section 2.1.3.

3.1 DNS Round Robin

DNSRR is presented in RFC 1794 [Bri95] in 1995 and was one of the first load balancing mechanisms used. It expands the original design of the DNS which is to look up and translate domain names into IP addresses. The DNSRR load balances by holding a list of IP addresses for the same domain name and resolve the domain names in a round robin fashion, that is, the IP address resolved is not the same for every lookup [KMR95].

There are several issues in using DNSRR. One of the most critical is the caching of addresses. Caching occurs in the routers and the clients using the domain name. This implies that when a member of the group leaves, for example in the case of a crash failure, changes in the group state do not necessarily propagate throughout the network immediately. Furthermore, in such a situation there is not mechanism to mask the failure. The level of reliability and availability is therefore low.

Another issue is that DNSRR does not provide true load balancing, that is, the mechanism does not take into consideration the load and the resources available of a server. The load is, when optimal balanced, equally balanced between all hosts regardless resources.

3.2 ONE-IP

One of the first virtual Web clusters was the ONE-IP originally developed at Bell Laboratories [DCH+97, WDC+97]. The purpose of this research project was to be able to address a cluster with a single IP address image. The project uses two methods of solving the problem, the *dispatching method* and the *broadcast method*. We will in the following describe the two methods.

The dispatching method is a *hierarchical, open client-server group* as any client can send to the group through the dedicated dispatcher which is controlling the group members load by dispatching the connections to the cluster hosts. A cluster of hosts has a common IP address, what we call a cluster address[1]. This approach is depicted in Figure 2.

The broadcast method is a *peer and open client-server group* as there is no dedicated dispatcher to control the load of the group members. Just as with the dispatching method a cluster address is identifying the cluster. The router is receiving the packet, determines whom to forward the packet, and broadcasts it on the network using Ethernet broadcasting. Each cluster host has a filter that

[1]For consistency purposes we chose to keep to the same terms in this report rather than adopting terminology from each technology as we go along

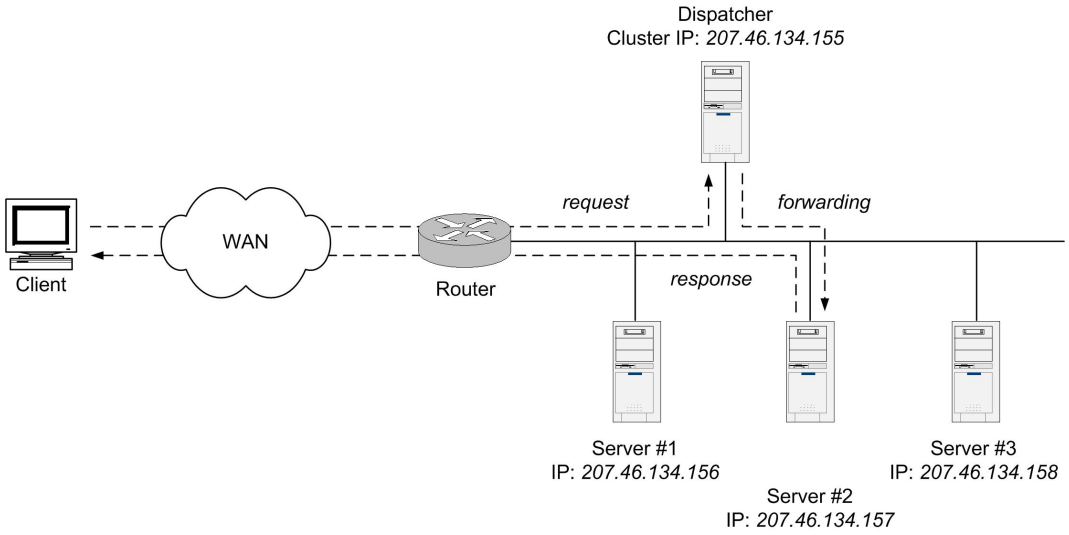

Figure 2: The dispatching approach in ONE-IP

filters packets meant for other hosts. The filtering rule is created using a unique number that each cluster host is assigned. This implies that only one group member receives the packet. As a host sending out the message we do not care who receives the message. It is therefore fair to say that this is an anycast simulated using broadcasting and filtering techniques. This approach is depicted in Figure 3.

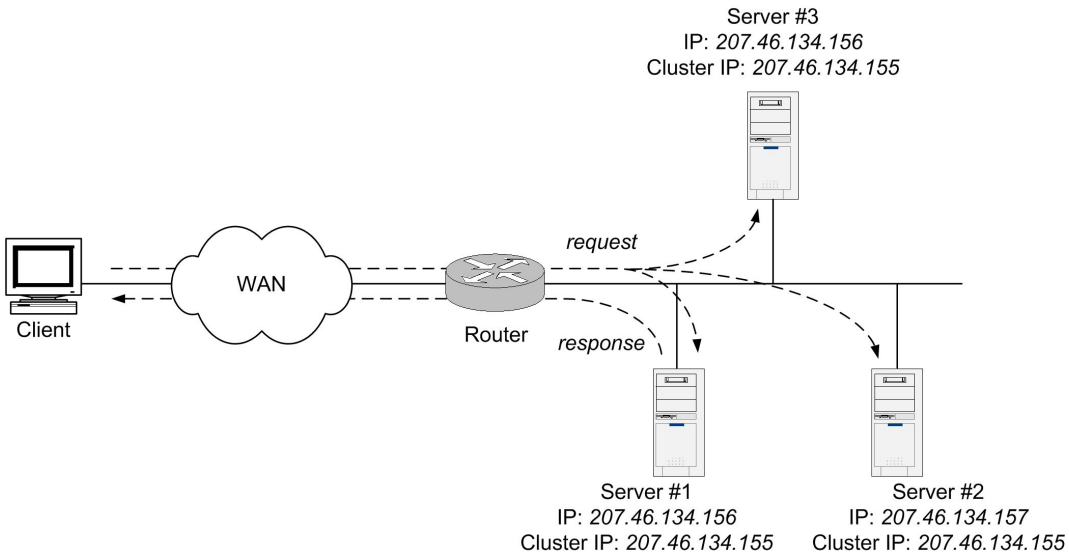

Figure 3: The broadcasting approach in ONE-IP

The membership is static for both methods as the unique number is statically assigned and cannot be changed once the cluster is running. This makes the solution less useful in the case of a crash failure where no other members are able to take over for the failed group member. There

are in general no techniques used to mask failures what so ever.

This architecture is good in the way that packets are filtered at the lower layers based on the filtering rule described. The packets are sent directly back to the client and not through the router which reduces the overhead of forwarding packets. However, by introducing a dedicated server to distribute the load, a single point of failure is introduced.

Using broadcasting all hosts within the network will receive the packets. This is an overhead worth noticing because all members as well as non-members on the same Ethernet segment, except the one which the packets are intended, will spend resources filtering the packets. This may however not be a problem as the cluster should run on a dedicated network in order to avoid unnecessary communication to take up bandwidth.

3.3 Clone Cluster

In the Cluster Clone approach Vaidya and Christensen extends the ONE-IP approach described in Section 3.2 by eliminating the dedicated server distributing the packets [VC01]. This is achieved using Ethernet multicast to deliver the packets to each cloned cluster host (hence the "cluster clone"). A cloned cluster host is basically a host that is an exact copy of another host. This can be achieved using replication. The packets are being filtered except on the hosts on which a connection has been established. What Vaidya and Christensen have done is a good proof of concept but needs more work to be a solid and applicable solution.

The filtering rule is static (modulus on the least significant byte in the IP address with the server number), that is, the same machines respond to the same IP addresses all the time. This is a not sufficient for a reliable Web cluster for the following reasons:

- The dispatching granularity, that is how the packets are being distributed eg all packets in connection, or all packets from one client (in this case IP address), will be handled by the same cluster host. In this case the granularity is IP oriented. This implies that there may be a risk of requests of significant sizes coming from one IP address, for example if the client requesting is using Network Address Translation (NAT), then the actual number of clients behind may be unprecedented. An example could be an Internet Service Provider (ISP) who externally only has one IP address, but may hold thousands of clients behind it, which the cluster is not aware of.
- If a cluster host leaves the group there is no fail-over mechanism and the service to the client is lost. All packets from specified IP addresses are still forwarded to the cluster host that has failed.

- The scheduling is unfair if the system is heterogeneous as one server may have more resources available than another.

Another issue that is not being addressed is the scalability of the cluster. As all packets blindly

must be sent out to all group members there is an overhead at each member in terms of processing the packets not intended for the host.

The implementation that was provided discards the gratuitous ARP packets being sent out. This makes it more problematic if other nodes on the network, except from the cluster nodes, are being configured with the same IP address. It is possible to set up a host on the same network with the same IP address as the cluster, in which case no warnings are being issued, on the cluster anyway. This is a minor issue as 1) the cluster often would be on a dedicated network so it would not be affected by unnecessary network traffic and 2) the host not belonging to the cluster group will most likely issue a warning.

As the Ethernet multicast address is applied directly it is not possible to do unicasting at the same time because it is only possible to hold one Media Access Control (MAC) address at the time. This is needed if we want to address a specific cluster host as in the case of administration etc. To do this another network card to connect eg to a separate back-end LAN is needed for unicast communication. This increases the complexity and cost of running such a network, but at the same time may increase the performance as communication is removed from the LAN with the multicasting.

3.4 Microsoft's NLB and WLBS

Today there is only one virtual Web cluster technology available for use in production which is the *Microsoft Network Load Balancing (NLB)* server [nlb00, Gre00] and the *Windows NT Load Balancing Service (WLBS)* [wlb99]. Both services are compatible with each other, so it is not necessary to upgrade all hosts when nodes with newer operating systems are introduced. The approach taken in NLB and WLBS is *k-delivery* where $k > 0$, and a *single response*. It both acts as a client-server group and a peer group.

All cluster hosts are being assigned a shared *cluster IP address* and a *dedicated IP address* that is unique to each cluster host. Traffic from all IP addresses, except from the dedicated IP addresses, is being load balanced.

Each host is being assigned a *priority* number ranging from 1 to 32 (maximum number of hosts), where number 1 has the highest priority and is the *default host*, 2 the second highest and so forth. If the default host fails the host with the next highest number takes over and becomes default host. This makes the system more difficult to configure and in the case of adding a cluster host to the system that is intended to be the new default host would require changes in all other hosts. But it is a general problem because there is no centralised configuration across hosts.

Port rules can be set up so it can be specified which hosts to handle the incoming connections. This is useful when services are only running on some of the hosts (due to restriction on licenses, issues with replicated data etc.). There are two policies that can be used here; the *multi-host* and the *single-host* policy. The *multi-host* policy distributes request to several hosts (specified via a *handling priority*) and the *single-host* policy directs all connections to a specific port to a single host. This can be used to block transport as well.

The *multi-host* policy adapts three load balancing approaches called *client affinity* which each has a different *dispatching granularity*:

1. *No client affinity* assures that load is being distributed among hosts with no regards to sessions. That is, traffic from one IP address can end up on different hosts, but traffic within one connection will still end up on the same cluster host.

2. *Single-client affinity* ensures that all traffic from one IP address all ends up on the same cluster host. In most cases this ensures the sessions being run.

3. *Class C affinity* ensures that traffic from the same class C address space ends up on the same cluster host. This means that even if a client uses several proxy servers the traffic still ends up on the same cluster host. This assumes that all proxies are within the same class C address space.

Parameterising the client affinity can become a powerful tool when administering the cluster. There may however still be some problems with session handling across multiple connections and the granularity of the dispatching, depending on the affinity chosen.

The percentage of load that each host are allowed is being specified. This allows an administrator to differentiate between hosts as to what they can be expected to process.

There are several ways NLB and WLBS can be used. The most interesting way, in this context, is the use of the anycasting approach implemented using Ethernet multicast distribution. The technique is similar to Vaidya's and Christensen's solution described in Section 3.3. It works by spoofing the MAC address so that the host in front of the cluster (eg a router) sends all packets destined to a single IP address to all cluster hosts. Only one of the servers respond to the packets, hence the anycasting. All hosts receive the packets which are being filtered if they are not intended for that specific cluster host. This may introduce a slight CPU overhead as packets are being filtered at a higher level that normal.

The dispatching of the connections between the cluster hosts is determined using an algorithm based on statistics on connections rather than on load of each cluster host (CPU and memory use).

The cluster is an open group model as each member may join or leave the cluster at any time. The distribution of the connections are done whenever a change in the cluster occurs, that is whenever a cluster host joins or leaves. A host may leave the cluster if a number of heart beat messages are not received by the other hosts. The process of defining the cluster in terms of who handles which IP addresses is called the *convergence process*. This implies that true load balancing is not accomplished. A server may be slower than usually for various reasons (memory leaks, service updates being installed, disk defragmentation etc.), or a service may crash in which case the clients requests will not be handled.

3.5 Distributed Packet Rewriting

The Distributed Packet Rewriting (DPR) approach is used in the project named *Commonwealth* [BCLM98, AB00] and in [CL02][2]. It works in the way that connections a group member does not have the capacity to handle is being forwarded to a member that can. The packets are not sent back through the *routing host* but are sent directly to the client (masking or spoofing the IP address so that the client thinks the packet is sent from the cluster host initially connected to). All packets in the connection are being forwarded to the new group member until the connection is seize to exist. The DPR method is depicted in Figure 4.

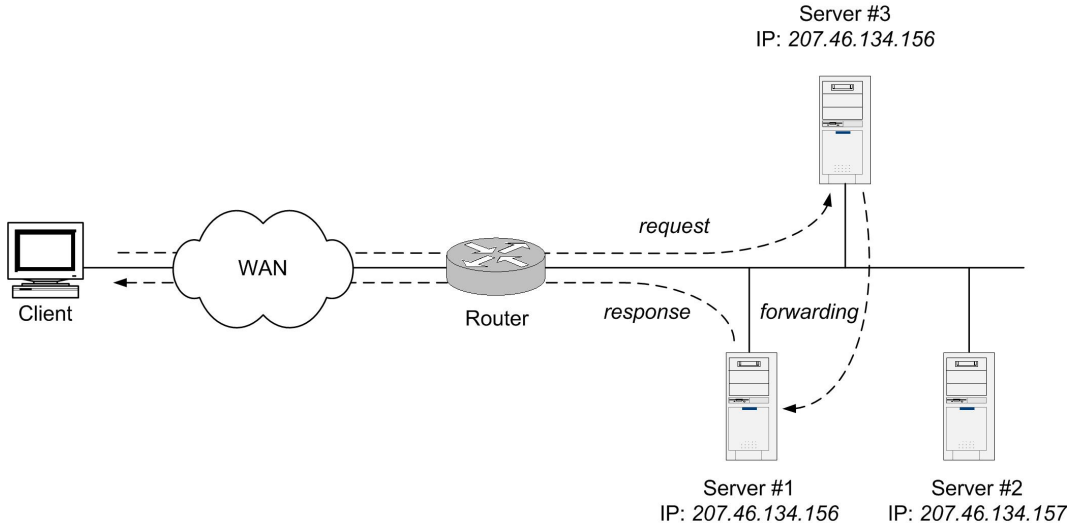

Figure 4: Distributed Packet Rewriting

DPR refines the control of load distribution by refining the granularity of the lightweight dispatching done by a host external to the cluster group. This external lightweight dispatcher could be a DNSRR algorithm or a dedicated load balancing switch. Using a non-adaptive load balancing algorithm we can be sure that the overhead in letting a cluster member route the packets to another member will persist which is not optimal. This overhead is not only in forwarding the packets but also in recalculating the IP and TCP checksums. The recalculation is need as the IP address of the destination is withheld in the packet, but the destination is not the same after rewriting the packet.

Using an external dispatcher introduces some, if not all, of the issues from the dispatcher to the cluster group. An example is the DNSRR which caches the IP addresses and therefore are not aware of the state of the group members being dispatched to. Chances are therefore that client requests are being routed to group members that are no longer functioning. Another example is the issue of single point of failure when using a dedicated load balancing switch.

Transparency is provided in the sense that the client cannot see which cluster member is actually handling the requests. But it is not fully transparent because the address of the host being connected

[2] *Socket Cloning for Cluster-Based Web Servers* is basically a system build upon DPR

to is still visible to the client.

DPR scales well, which is not that surprising due to the use of an external dispatcher which saves the cluster group resources. The limitations of scalability will have to be found in the uneven dispatching from the external dispatcher. This increases the risk of a group member using all its resources on rewriting packets and forwarding them to other members.

One of the major advantages of DPR is that the cluster hosts are not limited to a single collision domain nor to a single network. In the extreme the cluster members may be distributed over geographically different and distant areas. This is only limited by the costs of communicating over large distances.

In terms of availability DPR is actually worse than using a single host. The reason is that the group member routing and forwarding the packets needs to remain in the connection until terminated [VC01] which increases the probability of a failure. There is no fail-over mechanisms in DPR.

DPR does not work at all without an external dispatcher and is therefore not a true load balancing technique by itself. It is merely refining the granularity of the dispatching done.

The DPR approach is not a true Virtual Web Cluster as each cluster host is not entirely transparent to the client, depending on the way dispatching is implemented that is. If DNSRR is used the IP address of a single cluster host is revealed and not the cluster as a whole. We include it here anyway as it is highly relevant to our work and we can see the approach as *a cluster with many entries* and using one entry instead of another does not change the way the cluster works.

3.6 Common Address Redundancy Protocol

OpenBSD's Common Address Redundancy Protocol (CARP) is still under development and detailed documentation and information is limited to the man pages and source code [exs03, man03]. allows multiple hosts on the same network to share a set of IP addresses and enables a high availability.

An open group build upon CARP uses a *periodic broadcast membership* approach, here called advertisements, to maintain its *dynamic membership* structure.

For any of the virtual IP addresses a master will be chosen and the other hosts will become backups. Whenever the master leaves the group a backup host will take over as master. The time between the masters presumed *crash failure*, that is when it stops advertising its membership, to the backup takes over is 3 times the advertisement interval.

CARP determines the master from the frequency of advertisements, that is the more frequent a host sends out its group membership advertisements the more likely it is to become a member. It is possible to force a host to send out more frequently than others and therefore increase the chances of becoming the master.

It uses the multicast technique, as in 3.4 and 3.3, to send out the packets, but only *single delivery* and *single response* is required, that is, to the master. This gives a *unicasting* approach even though using multicasting to deliver the packets.

CARP is designed to be protocol independent and it therefore supports both IPv4 and IPv6.

The intention of CARP is not to provide a load balancing mechanism but rather to provide a backup mechanism in the case of a *crash failure*. An application of this protocol could be to support a backup router and therefore be an inexpensive alternative to for example Ciscos Virtual Redundant Router. Nevertheless CARP do provide a course grain load balancing. By specifying multiple MAC addresses for the same IP address the MAC addresses stored in the cache on other hosts will end up more or less randomly. There are however several drawbacks using this solution. First of all, the load balancing will not work on most web server clusters as these serve clients on the other side of a router where the MAC address always will be the same. Another drawback is that the granularity is very course and is unaffected by the load of each host.

CARP is newly developed and has not been released yet (at the time of the writing) but is planned to be released in the upcoming version 3.5 of OpenBSD. Because CARP is still under development it suffers from several issues and bugs, but from what we have seen it promises to deliver high availability. CARP does however not provide a flexible and reliable solution. The time it takes for the backup to mask the failure may prove critical.

3.7 Summary

We have in this section reviewed some of the closely related solutions and various conclusions can be drawn from this. First of all, the Web clustering techniques presented here in general provides high availability but low reliability. None of the solutions presented provided masking of any type of failure. Some solutions provides regrouping in the case of a crash failure, but this can take up to several seconds to make the system stable [nlb00].

Another issue is the often unfair dispatching approaches that are taken. If we wish to provide a highly available service we must ensure that the load is evenly balanced so that a single server is not overburdened and therefore becomes partial unavailable.

Granularity of the dispatching is a large area that should be researched. We have in this report described some of the approaches taken in various solutions. This is an issue not easily solved when making reliable clustering. The problem is the web services that keep the state information in the application layer. If these services experience an error resulting in a crash failure, the states are likely to be lost.

What all of the solutions suffer from is the inflexible nature of the load balancing. One has to realise that not one load balancing algorithm is superior to any other. The effectiveness of a load balancing algorithm depends on its use. It cannot be both fault-tolerant and fast at the same time, it can only be more or less faster than other algorithms providing the same level of fault-tolerance.

4 Problem Statement

As outlined and described in the previous section existing solutions do not provide the adequate flexibility for adapting to the demanding load balancing problems faced today. There is no perfect

solution for all these problems, but the described solutions do not easily allow the customisation or configuration needed to solve the real-world, often domain specific, problems.

It is not the intention of this project to solve all these problems, but it *is* the intention to create a flexible and extensible framework that allows people to solve, at least some of, these problems.

We will in the following describe the requirements for such a framework.

4.1 Approach

The technological approach must be fully and truly distributed. It should be possible to eliminate the single point of failure. Also, it should provide transparency to users and applications.

4.2 Flexibility

The solution has to be able to support implementation and installation of multiple algorithms. There must be a static interface between the implementation of the algorithm and the user (eg. the OS kernel).

Another way to make the solution flexible is to focus on its usability. The reader should, however, keep in mind that as important as usability may be, just as difficult it is to measure. We will therefore let the usability requirements be guidelines and limit ourselves from measuring these. The ultimate test with regards to these will be the test of time, and the measurement of how well adopted the framework will become.

The requirements for usability will be:

- Ease the future implementation of modules by providing as much transparency as possible.
- Follow the rules given in [Ray03], especially the rules of extensibility and modularity are important.
- Conforming to the unwritten standards of the open source community.
- Not restricting or enforcing policies about the use of the program.

4.3 Platform

The operating system platform on which we implement the solution has to be open source as we need to alter the way it handles network traffic. The operating system also has to support the TCP/IP protocol suite and Ethernet since these are the wide-spread network protocols in use in most networks today.

4.4 Performance and scalability

Our solution must not be the bottleneck when scaling the system, that is it must be scalable and limitations of this scalability must be from the implementations of the interface that we provide.

4.5 Limitations

As mentioned in the introduction the intention with this report is not to provide a load balancing solution but merely to create the framework for one. This implies that we do not focus on a specific algorithm and therefore do not implement one as part of our solution (although we do provide one as an example. See Appendix A.2). Furthermore we do not provide an extensive test as we only wish to prove the concept not to deliver a working solution. As for the usability requirements we cannot test these with the limited resources and time restrictions given and will therefore not be tested as well.

5 Load Balancing Framework Architecture

Given the problem statement, we have derived an architecture based on the Linux operating system, the Netfilter firewall and the packet mangling framework that it provides.

The concept is similar to the ones described in Section 3.3, 3.2 and 3.4. It is based on the multicast approach where the cluster hosts share an IP address and every host receives all packets (as depicted in Figure 3). The decision whether to accept the packet or discard it is done by each host. This is where our solution differs as the decision is handed over to an external module which can be implemented relatively easy by any programmer without any knowledge of kernel programming. All kernel specific implementations are hidden from the user of our framework.

The conceptual packet flow in the solution is shown in Figure 5 and described below.

1. A client wants to send a request to the cluster IP. We assume that the client is on the same network (Ethernet) as the cluster. The client could in this case be the gateway router. The client sends ARP request to discover the clusters MAC address.
2. The kernel replies to this request, but the reply is intercepted by the lb kernel module.
3. The lb kernel module calls the registered load balancers ARP handlers to get a decision on the ARP reply.
4. A load balancer module could reject, but claims responsibility in this case.
5. The lb module modifies the ARP reply by setting a multicast MAC address, and hands it back to the kernel network stack.
6. The kernel sends the ARP reply to the client.
7. The client stores the clusters MAC address and sends it's request to this multicast address.
8. The request packet is intercepted in the kernel by the lb module.
9. The lb module queries the load balancer module for a decision on the received packet.
10. A load balancer module claims responsibility, just as well as it could have rejected it.

11. The lb module gives the request back to the kernel, which passes it up the network stack.

12. The request is answered normally.

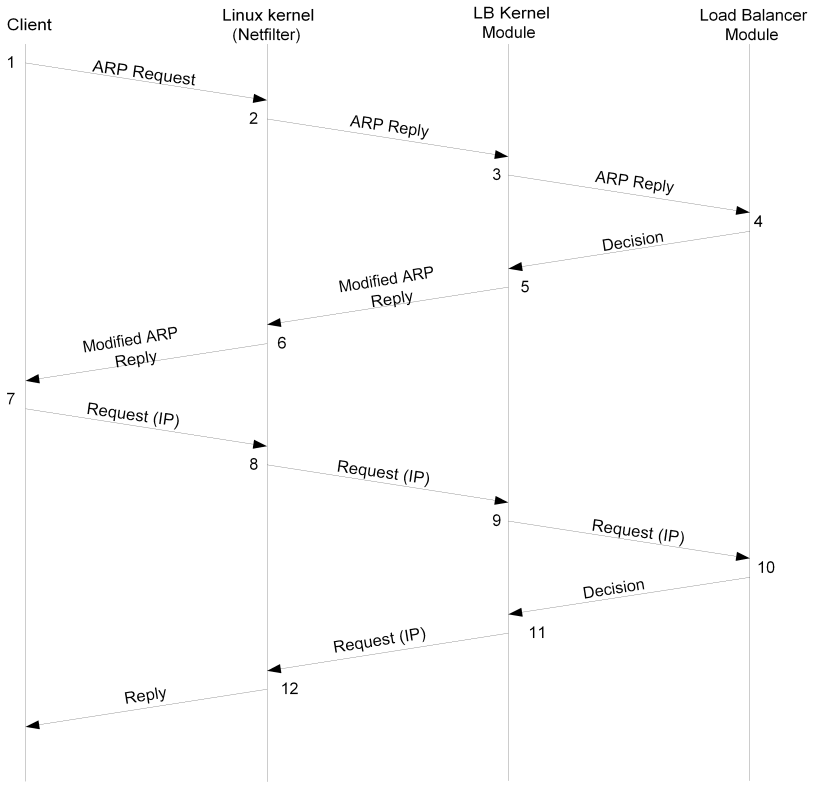

Figure 5: Packet flow in the network and in a cluster host kernel

5.1 Netfilter

Netfilter (www.netfilter.org) is the firewall subsystem in the Linux kernel, and was introduced with the 2.4.x kernel series (at the time of writing, the latest stable release of Linux is 2.4.23.

Netfilter provides a number of hooks which the traversal of a packet can be controlled with. A hook is a defined point in the traversal of a packet, when hit, jumps to the netfilter. Here the decision of what happens to the packet is made.

Netfilter allows to implement advanced firewall features in the Linux network stack, without bloating the network code. As a consequence, Linux now provides very advanced firewall features.

5.2 Conceptual model

In Figure 6 is depicted a conceptual architectural model of how the load balancing module interfaces the Netfilter and the TCP/IP stack.

Netfilter is a good interface to use when implementing the load balancing model interface as netfilter provides a clean interface, which makes it possible to alter the packet flow without altering

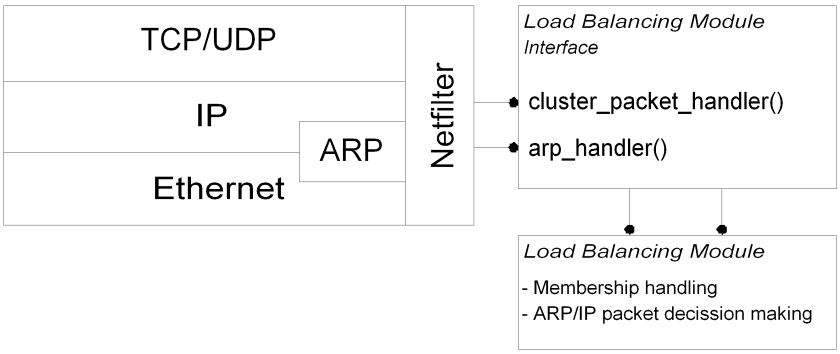

Figure 6: Conceptual architecture

the kernel. This ensures that the solution is made flexible and transparent. It is not needed to look all over the source code in order to find the related code. Doing it this way makes it easier to implement and maintain the solution with the benefits of higher portability, security and stability. Using netfilter we ensure the required flexibility as stated in 4.2.

The downside of using Netfilter is that it is only implemented on Linux at the moment which makes the solution less portable to other operating systems. But it is our belief that the benefits of using Netfilter outweighs fully the disadvantages.

5.3 Kernel modules

In order to make the system more flexible it was decided to implement our solution as loadable kernel modules. While Linux is a traditional monolithic kernel, it supports modification/insertion of kernel code at runtime through a feature known as *dynamic loadable kernel modules* (LKMS). Large parts of the Linux kernel have the option of being compiled and loaded as LKMS.

The benefit of having the solution as a loadable module, is that it can be deployed with existing, even running, Linux kernels, without the need for rebooting or recompiling the entire kernel.

However, there is no hindrance that our load balancing mechanisms can not be integrated by recompiling the entire kernel and adding the corresponding entries to the sys call table.

5.4 Extensibility of our framework

Eric Raymond outlines in [Ray03] the "Rule of Separation: Separate policy from mechanism", which is an essential part our design philosophy, in regards to our load balancing framework.

We decided to separate the core infrastructure needed to perform multicast based load balancing from the algorithms used for implementing the actual decisions on how the load should be distributed. These decisions, whether one host should handle a packet addressed to the cluster, are deferred to the extension modules.

Inspired by the extensibility of both Netfilter and LKMS, we decided on providing a similar extension mechanism for our framework. Extension modules registers themselves with the framework, and are then called upon when load balancing decisions need to be made. This approach is

much the same as the one used for implementing firewall filters with Netfilter. In this model the load balancing algorithms are implemented as intelligent filters.

5.5 Kernel vs. user space

The core mechanisms for managing transparent addressing of the cluster must be, and should be, implemented as part of the kernel. However, it makes sense to implement some load balancing extension modules in userspace. That is, running as normal processes without the privileges of being part of the kernel.

First of all there is a security concern. A process running as a restricted user can do less damage if compromised. Most application-level networking services take this approach for example the Apache webserver.

Secondly it is much easier to monitor a process, than monitoring your kernel. If it crashes or runs out of control, it can be restarted or killed. This is not possible with the kernel, and in the case of a kernel crash the entire host needs to rebooted.

Finally it also gives a higher flexibility for anybody wanting to write load balancing modules. They can use standard application libraries and even other languages than C which is the language of the majority of the Linux kernel. Userspace programs are usually written at a higher abstraction level than the kernel, which makes it conceptually easier for most people to comprehend. This makes it easier to prototype a new load balancing algorithm and bug tracking it in the case of an error.

There are, however, downsides of running a load balancing algorithm in userspace. The most notable is performance. Userspace programs can be swapped (paged in Linux, to be exact) to the disk. Also, on a loaded system a userspace process might wait a while to be scheduled if the system is highly loaded. The kernel space memory on the other hand, is never swapped out and the kernel operates at the highest priority.

5.6 Connection tracking

The Linux kernel supports what is known as connection tracking. The kernel keeps track of connections to the system, and their state. It is used by the NATing subsystem to keep track of connections to different hosts. Various kernel modules implement support for different protocols, some on the application level such as ftp. A conceptual connection can consist of several connections on a lower level.

In relation to our system, it is extremely useful. Typically, a conceptual connection should be handled by a single host. Connection tracking allows load balancing algorithms to operate a higher abstraction level. Instead of taking decisions about every low-level packet, by forcing it to implement it's own connection tracking, it can decide whether it should handle a connection or not.

This also limits the load on the load balancing algorithm, which in turn takes fewer decisions.

On the other hand, some load balancing algorithms might want to distribute connections to several cluster hosts. In that case, it would not be sensible to enable connection tracking.

6 Implementation

In this section we will describe the implementation of the solution presented in previous section. We will do this by first describing the dependencies and structure of the source code. Then the kernel module and userspace module are described and in the end how to create new modules and extend the functionality of the solution.

6.1 Browsing the source

When files are referred to, throughout this document, it will be relative to the top-level directory of the source tree. The top-level directory of the source code archive is organised in the following manner:

- *Makefile* The top-level make file. Type *make* in this directory to compile everything recursively.
- `include/` All the common include headers (except the C++ ones) are in this directory.

- `common/` Files, other than include files, that are shared between the kernel and the userspace API.

- `c++/` This directory contains the C++ API files.

- `doc/` Documentation. The `report/` directory contains sources for this report, and the `api/` directory generated API documentation.

- `ext/` Extension modules:

  - `ext/simple_hash/` A simple demonstration hash based load balancing algorithm. There is a kernel space and a userspace version of this. This module was used for the performance tests. The algorithm only works on a cluster with a fixed number of nodes, and does not provide any fail-over. This example is described further in Appendix A.1.
  - `ext/watchdog/` The watchdog is an algorithm that keeps track of available nodes in the cluster. It is written in C++ and is further described in Appendix A.2.

- `kernel/` In the kernel directory you will find the main kernel module (*lb_core.c / lb.o*), along with a few additional kernel modules. *acceptall.c* and *dropall.c* are kernel-level extension modules, mainly useful for testing purposes. Also, *droparp.c* drops all ARP replies from one hosts. This can be useful, if setting up a static cluster with a fixed number of nodes and you want all but one host to drop replies.

- `test-suite/` Miscellaneous test scripts.

6.2 The main kernel module

The main kernel module, is the module that provides the necessary infrastructure for the load balancing framework. It uses the netfilter framework to snoop incoming IP packets and outgoing ARP packets.

When the module is loaded (with `insmod`) it parses the configuration variables sent to it. It reads the cluster Ethernet address, the cluster IP address and the network device the cluster should use for communication. As other optional parameters it can be told if it should use connection tracking as described in Section 5.6 and whether it should expect userspace extension modules. This is handled in the *lb_core_init* function and subsequently the *init_config()* function. Utility functions for parsing Ethernet MAC addresses, IP addresses etc. are shared with the userspace library and found in the *lb_utils.c* file.

The module allows accessing configuration variables after it has been loaded, via the proc filesystem which hooks into the `/proc/net/lb` directory. The proc filesystem is a memory filesystem that gives userspace processes access to runtime information about the kernel. *lb_proc_create()* creates the file entries in `/proc` and associates them with the *lb_proc_read()* function. This function is called whenever a process reads one of the files. It determines which of the files that has been read, and sends the corresponding data to the process. The *lb_proc_cleanup()* function removes the entries from proc, should the module be unloaded.

The kernel module calls the *nf_register_hook()* function with two *nf_hook_ops* structures that define where and how we want to hook into the network stack. The *arp_ops* hooks into outgoing ARP packets, *cluster_ops_4* hooks into incoming IP packets. As part of the structures are function pointers to functions that should be called when a packet passes through the hook. Our ARP handling function is called *arp_handler()* and the function that handles IP packets are called *cluster_packet_handler()*.

After the module has been loaded, extension modules can hook up to the core module. They do this by registering callback functions with the exported *lb_register_arp_responsible()* and the *lb_register_packet_responsible()* functions. The prototypes for these functions (the API) can be found in *include/lb_core_api.h*.

6.2.1 ARP handling

Before an ARP packet is sent to a network interface driver, Netfilter will hand the packet to the *arp_handler()* function.

First we check whether the packet is an ARP reply. If it is not, it does not concern us, and we tell Netfilter to send it along. Also, as ARP is used by several network layer protocols we check if the packets protocol at the network layer is IP, and specifically ipv4. If it is not, we send it along.

Currently, our framework only handles Ethernet with 48 bit addressing, so any ARP packets that deal with an other link layer protocol are passed along.

We then check whether it is a reply from the cluster IP. All other replies should be allowed to pass without further questioning. Linux has the feature that it routes ARP packets. The

consequence is that with multiple interfaces on the same network, it duplicates ARP replies. We only want one reply so we check whether it is going out on the interface the cluster is configured to use. If it is not, we quietly drop it.

Finally, we call an extension module, if one has been registered, and ask for a decision for the packet. First we check for kernel space modules, secondly userspace modules, if the userspace option is set, that is.

Before an ARP reply is transmitted, we overwrite the packets source hardware address with the clusters hardware address. This is necessary since we use an alias interface for the cluster IP. Alias interfaces normally always use the underlying interface's hardware address.

6.2.2 IP packet handling

In the *cluster_packet_handler()* function, we get to look at all incoming IP packets. We are only interested in those packets which are addressed for the cluster, so we check for this, and pass the rest along.

If the *conntrack* option is set, we use the connection tracking system used by the NATing code to keep an account of existing connections to the cluster. Usually, one host should handle one connection such as eg a TCP connection. If *conntrack* is enabled, packets belonging to an existing connection are passed up the stack directly, instead of deferring that decision to the extension modules.

Then we check if there are any extension modules to call. Before calling these we change the packet to look like a unicast packet. Recall that all packets to the cluster are multicast packets. If no extension modules are registered to handle the decision, we decide that it should be handled and the packet is passed up the stack.

6.2.3 Compile time parameters

Options such as the conntrack and userspace options can also be set at compile time. This is done in the *include/lb_core_conf.h* file. Disabling an option compile time makes the module compile without the code needed to handle the feature which minimises module size and optimises performance.

6.3 Userspace

If the lb kernel module is loaded with the userspace option, load balancing decisions are deferred to userspace algorithms. Packets are forwarded using an IP queue (libipq), and are read at the pace of the userspace program. If the userspace program cannot keep up the reading pace, packets will get queued, and might eventually be dropped. The userspace queue handling happens in *userspace/lb_userspace.c*.

Userspace algorithms, uses the exact same interface for registering packet handlers as in kernel space. An ARP handler is registered with *lb_register_arp_handler()* and cluster packet handler

is registered with *lb_register_packet_responsible()*. The callback functions are type-safe, and the registered functions must correspond to the defined prototypes.

The userspace environment is initialised with a call to *lb_init()*. This function blocks, e.i. it takes over the programs running thread. If it should terminate it is because it encountered a fatal error, and the userspace programs can then check the return value, and users the system log, to see what went wrong.

First it reads the systems configuration from the /proc directory and performs sanity checks on the configuration. It keeps the configuration in a structure very similar to the one maintained in the main kernel module. Configuration variables can be accessed by userspace programs using the functions defined in *include/lb_cfg.h*.

Next, it tries to connect to the kernel IP queue. If it succeeds it will enter the main loop. The main loop reads packets from the queue (*lb_read_packet()*, and decides what to do with them in *lb_handle_packet()*. The kernel tags each packet with an identifier, which are use by *lb_handle_packet()* to determine the type of the packet, and in turn what handler to call.

After calling a handler in an load balancing algorithm, it knows what to do with the packet. It will issue a verdict on the packet and pass it back to the Netfilter framework, which will either drop the packet, or pass it up the protocol stack.

Additionally, the userspace framework provides some convenience functions for packet handling and logging.

The include file *lb_packet.h* defines functions for convenient parsing of IP packets which can be used by load balancing modules. There are functions for extracting relevant fields of a packet such addressing, determining upper layer protocol etc.

lb_log.h defines functions for transparently logging to the system log.

6.4 Writing load balancing extension modules

An auto generated API documentation is provided as part of source distribution in the `doc/api` directory. However, to illustrate the principle, we provide an example of writing an extension module here.

6.4.1 Example: simple hash

This section covers writing new modules for the framework by using the simple hash algorithm (*ext/simple_hash*) as an example. This module is available both in a userspace version and a kernel space version.

We examine what the two versions of the module have in common and what differs. Both modules define a function for handling ARP replies and a function for handling cluster packets. The function prototypes looks the same in both modules:

```
enum lb_responsible simple_hash_arp_handler(lb_packet) enum
lb_responsible simple_hash_packet_handler(lb_packet)
```

These functions are used as arguments for the *lb_register_arp_responsible()* and *lb_register_packet_responsible()* functions, which registers the functions with the framework so they will get called when a load balancing decision needs to be made.

The decision is one of values defined in the *enum lb_responsible*, that both functions return. It is defined as:

```
enum lb_responsible {
    LB_RESP,    /* I am responsible */
    LB_NOT_RESP /* I am not responsible */
};
```

The ARP handler always returns *LB_RESP* for if the node is the first node in the cluster, while the rest of the nodes returns *LB_NOT_RESP*.

The packet handler instead hashes relevant parts of the packet Note that the kernel space module rutines for doing this is more complicated since it does not have access to the *lb_packet* functions:

```
  switch(lb_packet_get_protocol(pkg)) {
  case LB_PROTO_TCP:
  case LB_PROTO_UDP:
    hash = jhash_3words(lb_packet_source_port(pkg), lb_packet_dest_port(pkg),
                        lb_packet_saddr(pkg), SIMPLE_HASH_INIT_VAL);
```

It finds the responsible host by applying a modules operation to the hash and returns either LB_RESP or LB_NOT_RESP.

The differences between the kernel space and userspace module is mainly how the modules are started. The userspace is started from the normal *main()* function which after having registered the callback functions, call the blocking function *lb_init()*, that reads packets from the kernel IP queue. The kernel modules *simple_hash_init()* function gets called, when the module is loaded.

The kernel module also calls the *lb_unregister_arp_handler()* / *lb_unregister_packet_handler()* functions, which removes previously registered handlers. If a userspace module wishes to register a new handler, it also has to call the relevant unregister function first.

6.4.2 Modules written in C++

The principles for writing a load balancing extension module in C++ is essentially the same. However, all the API functions have object oriented C++ wrappers. They are all encapsulated in the LB namespace. Instead of registering callback functions you derive a class from the abstract class LB::LoadBalancer, which defines the two virtual handler methods. You then register an object of your derived class with framework, instantiates the singleton class LB::LBAPI, and pass the LoadBalancer object to it. The watchdog algorithm in *ext/watchdog* and described in Appendix A.2 is an example of writing an extension module using the C++ API. Also, see the API documentation for details.

7 Evaluation and Test

In order to prove the concept and performance we will in this section describe the performance of the framework. There are two main objectives for the tests carried out here. The first is to characterise the overhead introduced in filtering at a higher level. The second is to study the scalability of the framework.

7.1 Issues when testing

Before testing there are some issues that need special attention in order to get the most accurate data. First, on clusters the parallel execution involves little IPC and testing should therefore focus on computation rather than communication in order to match the clients computational powers with the clusters. That is, we need to make sure that the cluster hosts do not out-perform the client in the test environment. We do this by adding computational complexity on the server side to each request.

Another issue is the network traffic that might exist beside the traffic for the test. The OS might create traffic which we have no control over, eg ARP requests and replies or simply services communicating. In order to solve these issues a dedicated network should be used so there is no interference from hosts that are not related to the test setup. All services that are not related to the tests should additionally be made silent. But the only real effective way is to run as many tests as possible and discard test results that are outside what is reasonable.

7.2 Testbed and test description

The testbed and test case design was created with the issues presented in Section 7.1 in mind. We were however constrained by the limited resources available. It would have been more clear if a homogeneous cluster was used to test the scalability, however, this was not possible.

The configurations that the cluster hosts and the clients used in the tests are presented in Table 1.

Host	Operating System	CPU	Mem	File system	Web server
1	Linux 2.4.20-8, Red Hat 9.0	Intel Pentium Celeron, 400 MHz	64 Mb	EXT3	Apache 2.0.40
2	Linux 22.4.22, Slackware 9.0	Intel Pentium II, 266 MHz	64 Mb	EXT2	Apache 1.3.20
-	Windows 2000, v. 5.00.2195 SP 3	Intel Pentium III, 600 MHz	250 Mb	NTFS	-

Table 1: Cluster hosts and client setup

The testing was conducted with the ApacheBench 2.0.40-dev benchmarking tool that comes with the Apache Web Server 2.0. A series of tests was conducted with increasing intensity. The idea is find out how the server reacts at different loads. Ideally the server would spend a proportional amount of time on each request that is 1 request takes n msec, 2 requests takes $2 \times n$ msec etc.

This would be true for a sequential execution but to for a parallel. Sending multiple requests to the server increases the utilization of the resources, makes it more difficult for the service to find resources and as a result increases the response time exponentially.

The intention of the tests performed here was to show how our solution can spread out the load on the servers and put off the limit and increase the throughput of eg a web service.

Following tests were conducted:

Baseline (host 1) - Host number 1 is being tested as a single and unmodified server, that is no load balancing modules are loaded.

Baseline (host 2) - Same test as above but with host number 2.

Balanced (2 hosts, kernel space) - A cluster with two hosts is being used. This test uses the *kernel space* module which uses a simple hash algorithm to balance the load. This is described in Appendix A.1.

Baseline (host 1, userspace) - Host number 1 is being tested as a single server with a userspace module loaded.

Baseline (host 2, userspace) - Same test as above but with host number 2.

Balanced (2 hosts, userspace) - A cluster with two hosts is being used. This test uses a *userspace* module.

In order for the client to out-perform the cluster hosts we chose a client that is superior to these. Additionally the cluster hosts were slowed down by adding computational complexity to requests. This way we avoided that the client became the weakest link, which would have affected the test results.

7.3 Userspace tests

Using the userspace feature of our solution is expected to result in some overhead, as described in Section 5.5. In Figure 7 is depicted the userspace tests. First baseline tests with the two cluster hosts as single servers and then the two servers in a cluster. The X-axis shows the throughput, that is how many requests a second it was possible to get through. This should not be mistaken for how many requests we tried to get through. Up the Y-axis is the delay in msec per request. One might wonder about the unevenness of the line. This has to do with how the OS is handling the resources. At some point the OS needs to reorganise the resources which takes time and results in these variations. The values used in the graph are mean values and therefore easier to read and conclude on. The original data would be scattered all over the chart and difficult to read.

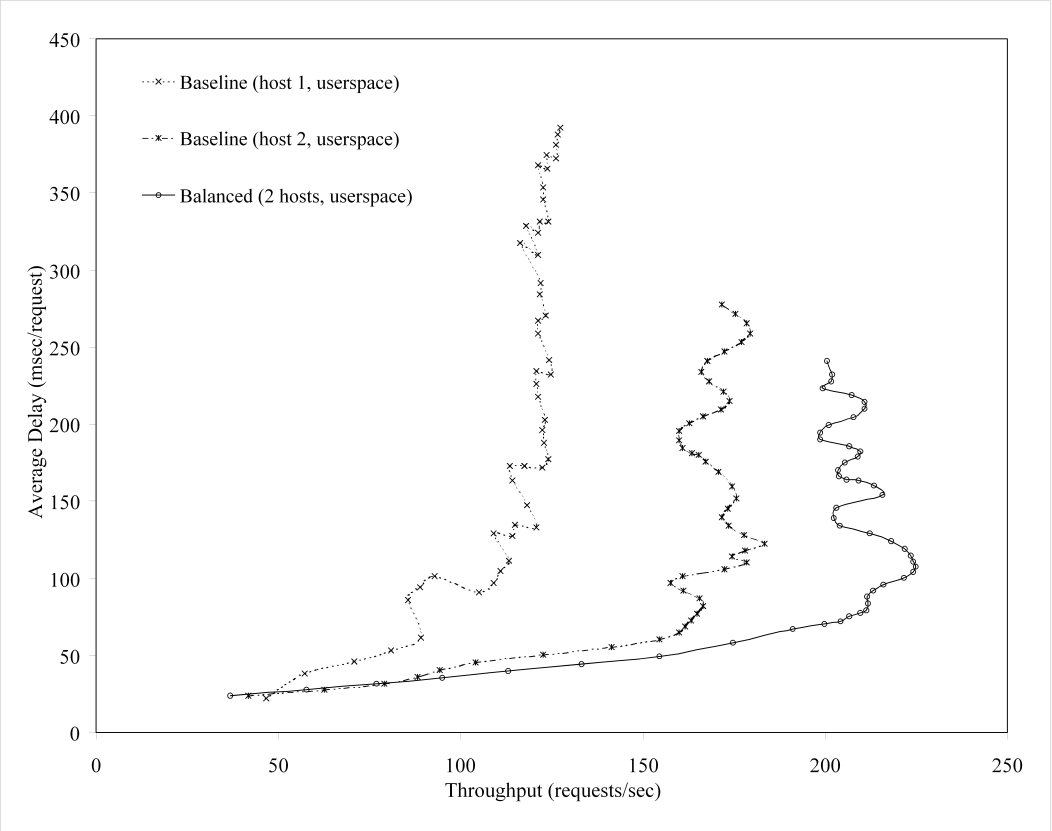

Figure 7: Performance of userspace modules

7.4 Kernel space tests

The kernel space feature was expected to be more efficient than the userspace as described in Section 5.5. In Figure 8 are the test data gathered from the baseline tests without any modules loaded and the kernel space loaded. The measurements are the same as for the userspace tests.

7.5 Evaluation on tests

Studying both Figure 7 and 8 it can be concluded that the framework we have developed can be used with benefits in the form of higher throughput compared to using a single host. In both figures we can see that the throughput is approximately 20% higher for the cluster than for the fastest host (host 2). Ideally it should have been the throughput from the two cluster hosts combined, but because of the unfair dispatching policy each host receives an equal amount of requests. The cluster is therefore, to some extend, restrained by the slowest host. It is however possible to conclude that it is beneficial to clustering using our framework, and even more beneficial if the right load balancing algorithm is used.

In the userspace tests we can see a more uneven graph that in the kernel space tests. This has to do with the way the OS handles userspace processes. Threads running in kernel space has a higher priority in respect to scheduling and access to resources and the fluctuation of the graph

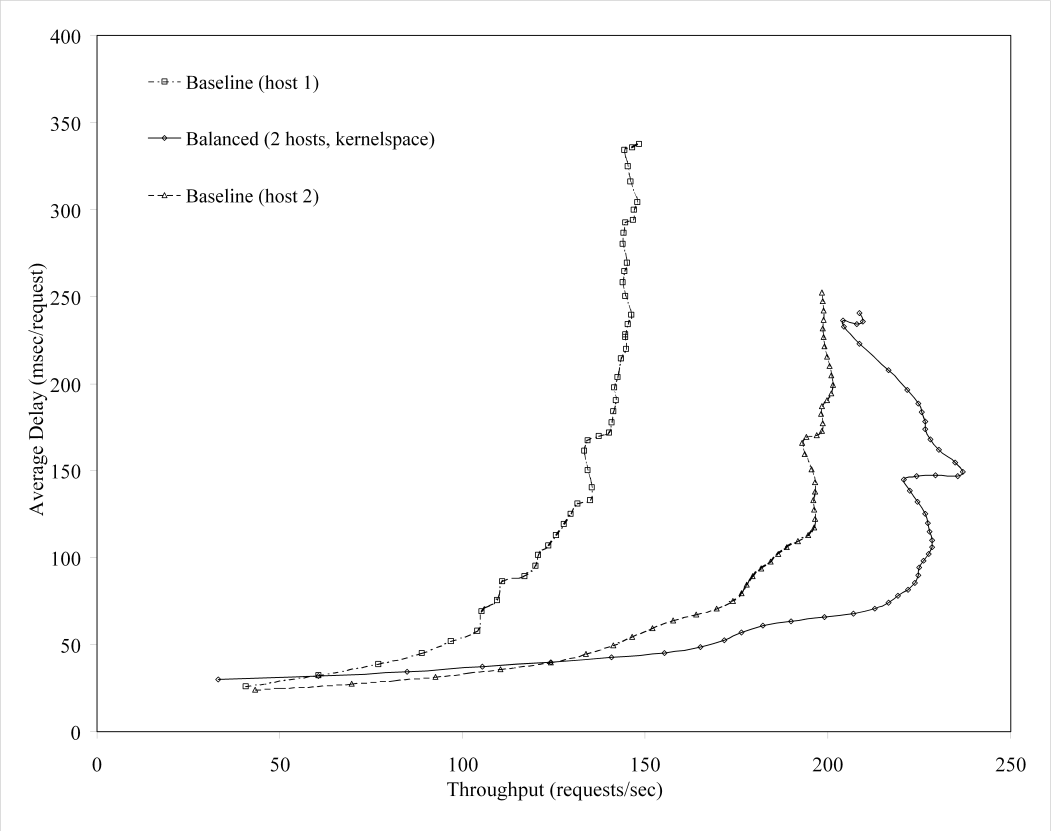

Figure 8: Performance of kernel space modules

will therefore be smaller than those of the userspace.

Comparing the userspace and kernel space we can conclude that there is a significant performance loss ($\approx 10\%$). Whether this is acceptable would have to be weighted against the complexity of implementing load balancing algorithms as kernel space modules.

The neighbour surveillance algorithm described in Appendix A.2 was not included in the graphs because it as such does not differ from the userspace algorithms. What is interesting about the neighbour surveillance algorithm is that it has a dynamic membership structure, and the only time there is a significant performance penalty is when the cluster group reorganises due to a host leaving or joining the group. The algorithm is however still interesting as it demonstrates the potential of the framework.

All the test results would be easier to conclude upon if we had used a homogeneous cluster instead of the heterogeneous setup we were restrained by.

8 Summary and Future Work

We have in this report described the fundamentals of webserver scaling and in particular virtual web clusters which would be fair to classify our framework as. Furthermore is the basic terminology described which lays the foundation for the short survey of research and available solutions in

this area. From the survey the problem of inflexibility was derived and a solution was proposed, implemented, tested and concluded upon.

We believe that the solution presented in this report has a high potential of becoming widely used. Just as with Netfilter our solution does not restrain the user to use a fixed solution but invites every potential user to contribute, develop their own solution or modify an existing.

Future work should include polishing the implementation and make it freely available under the GNU Public License in order for people to freely use and contribute to. Some optimisation in the userspace module should also be made in order for the modules to perform optimal. The module using the neighbour surveillance algorithm should be modified so it distribute the load in a fair way to the cluster group members.

References

[AB00] Luis Aversa and Azer Bestavros. Load Balancing a Cluster of Web Servers - Using Distributed Packet Rewriting. In *IEEE International Performance, Computing, and Communications Conference*, Phoenix, AZ USA, February 2000.

[ACT97] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. In Marios Mavronicolas, editor, *Distributed algorithms*, volume 1320 of *Lecture Notes in Computer Science*, pages 126–140, Sept 1997.

[BCLM98] Azer Bestavros, Mark Crovella, Jun Liu, and David Martin. Distributed Packet Rewriting and its Application to Scalable Server Architectures. Technical Report 1998-003, January 1998.

[BEH+97] E. Basturk, R. Engel, R. Haas, D. Kandlur, V. Peris, and D. Saha". Using network layer anycast for load distribution in the internet. Technical report, IBM T.J. Watson Research Center, 1997.

[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Company, Greenwich, 1996.

[Bri95] T. Brisco. RFC 1794: DNS Support for Load Balancing, April 1995.

[CCC01] V. Cardellini, E. Casalicchio, and M. Colajanni. A performance study of distributed architectures for the quality of Web services. In *Proceedings of Hawaii Int'l Conf. on System Sciences (HICSS-34)*, pages pp. 3551–3560, Maui, Hawaii, USA, January 2001. IEEE Computer Society.

[CCCY02] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed Web-server systems. *ACM Computing Surveys (CSUR)*, 34(2):263–311, 2002.

[CL02] Yiu-Fai Sit Cho-Li. Socket Cloning for Cluster-Based Web Servers. In *4th IEEE International Conference on Cluster Computing*, pages 333–342, Chicago, Illinois, September 2002.

[Cri91a] F. Cristian. Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4(4):175–188, 1991.

[Cri91b] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[DCH+97] Om P. Damini, P. Emerald Chung, Yennun Huang, Chandra Kintala, and Yi-Min Wang. ONE-IP: Techniques for Hosting a Service on a Cluster of Machines. In *The Sixth International World Wide Web Conference*, Santa Clara, CA, April 1997.

[DGLS99] Bill Devlin, Jim Gray, Bill Laing, and George Spix. Scalability Terminology: Farms, Clones, Partitions, and Packs: RACS and RAPS. Technical report, Microsoft Research, Redmond, WA, December 1999.

[DH95] S. Deering and R. Hinden. RFC 1883: Internet Protocol, Version 6 (IPv6) Specification, December 1995.

[exs03] Experimental Support for CARP (Common Address Redundancy Protocol). Newsgroup, TAC News Gateway, November 2003. `http://news.gw.com/netbsd.bugs/26197`.

[Gre00] John Green. Win2K Network Load Balancing. *Windows & .NET Magazine*, November 2000. `http://www.winntmag.com/Articles/Index.cfm?ArticleID=15724`.

[Jal98] Pankaj Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, Upper Saddle River, New Jersey, 1998.

[Jen97] Robert J. Jenkins. The Hash. *Dr. Dobbs*, 1997. `http://www.burtleburtle.net/bob/hash/doobs.html`.

[KMR95] Thomas T. Kwan, Robert McCrath, and Daniel A. Reed. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer*, 28(11):68–74, 1995.

[KT92] M. Frans Kaashoek and Andrew S. Tanenbaum. Efficient Reliable Group Communication for Distributed Systems. Technical report, Faculteit Wiskunde en Informatica, Vrije Universiteit, Amsterdam, June 1992.

[man03] Common Address Redundancy Protocol - Man Page. Man page, OpenBSD, 2003. `http://www.openbsd.org/cgi-bin/man.cgi?query=carp`.

[nlb00] Network Load Balancing Technical Overview. Technical report, Microsoft Corporation, 2000. `http://www.microsoft.com/windows2000/docs/NLBtech2.doc`.

[PMM93] C. Partridge, T. Mendez, and W. Milliken. RFC 1546: Host Anycasting Service, November 1993.

[Ray03] Eric Steven Raymond. *The Art of Unix Programming*. Addison-Wesley, September 2003.

[SBC+94] L. Snyder, F. Baskett, M. M. Carroll, D.M. Coleman, D. Estrin, M. L. Furst, J. Hennessy, H. T. Kung K. Maly, and B. Reid. *Academic Careers for Experimental Computer Scientists and Engineers*. National Academy Press, Washington DC, 1994.

[SSB00] T. Schroeder, G. Steve, and R. Byrav. Scalable Web Server Clustering Technologies. *IEEENetwork*, 14(3):38–45, May – June 2000.

[STA01] Anees Shaikh, Renu Tewari, and Mukesh Agrawal. On the Effectiveness of DNS-based Server Selection. In *IEEE INFOCOM 2001*, Anchorage, AK, 2001.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Upper Saddle River, New Jersey, 1995.

[VC01] Sujit Vaidya and Kenneth J. Christensen. A Single System Image Server Cluster using Duplicated MAC and IP Addresses. In *26th Annual IEEE Conference on Local Computer Networks (LCN 2001)*, Tampa, Florida, USA, November 2001. IEEE Computer Society.

[WDC+97] Yi-Min Wang, Om P. Damani, P. Emerald Chung, Yennun Huang, and Chandra Kintala. Web Server Clustering with Single-IP Image: Design and Implementation. In *Proceedings of the Int. Symp. on Multimedia Information Processing*, December 1997.

[wlb99] Windows NT Server, Enterprise Edition - Microsoft Windows NT Load Balancing Service. Technical report, Microsoft Corporation, 1999. `www.microsoft.com/ntserver/docs/WlbsTech.doc`.

A Load Balancing Algorithms and Approaches Used

We will here present the two algorithms and approaches used in the example modules.

A.1 The simple hash algorithm

As the name suggests this is a simple algorithm. It provides no fail-over safety and it works with a fixed number of hosts in the cluster.

Every host in the cluster knows the number of hosts in the cluster, and what number they are in cluster. The hosts are sequentially numbered starting from one.

For each packet received, the algorithm creates a hash based on the senders address, the source port and the destination port. It uses a derived version of Bob Jenkinks Hash [Jen97], which should have relatively good distribution and performance.

The outcome of the hashing algorithm is a 32 bit integer which is then divided with the number of hosts in the cluster. The remainder (plus one) will correspond to one of the hosts host id. That host will claim responsibility of the packet. The rest will not.

The algorithm is fast but it has the drawback that the slowest host in the cluster becomes a bottleneck. It aims to distribute load evenly between the hosts in the cluster, but if the hosts have different configurations the slowest hosts might become overburdened.

A.2 Neighbour surveillance algorithm

The example here is used for reaching agreement of the group membership in the cluster. It uses a similar approach as the *neighbour surveillance protocol* described in [Cri91a], although some significant differences are present. The algorithm presented here uses a connection oriented approach just as the way a host joins the cluster is different. The algorithm does not balance the load of the cluster hosts but merely keeps the membership list updated. The neighbour surveillance algorithm used in this example can be described as following.

task Membership:
  var *clean*: **Boolean initially true**
    membershiplist: Set-of-Members **initially**
    timeout: **Time initially** 3 seconds
    schedule(Listen-for-broadcasts)
    schedule(Neighbour-surveillance)
    schedule(Keep-membership-alive)
    schedule(Send-present)

task Time-out(E: **Time**, m: **Member**)
    block until $current - clock > E$
    remove-member(m)

```
        clean ← false

task Listen-for-broadcasts
    cycle
        if receive("join", m)
            create-member(m)
        if next(myid, membershiplist)= m
            clean ← false

task Keep-membership-alive
    cycle
        if | membershiplist |< 2 ▷We have not joined any group yet
            broadcast("join");

task Neighbour-surveillance
    cycle
        connect to predecessor (myid, membershiplist)
        cycle
            if receive("present")
                Time-out(current − clock + timeout, predecessor(myid, membershiplist))
            else
            if receive("membershiplist", M)
                Time-out(∞, predecessor (myid, membershiplist))
                membershiplist ← M
                break cycle ▷ So we can establish a new connection to a new neighbour

task Send-present
    cycle
        if | membershiplist |< 2
            if clean = false
                send("membership-list", membershiplist) to next(myid, membershiplist)
                clean ← true
            else
            cycle while clean = true
                send("present") to next(myid, membershiplist)
```

Where $next(m, M) \equiv$ **if** $m = max(M)$ **then** $min(M)$ **else** $min\{p \in M \mid m \prec p\}$ and $predecessor(m, M) \equiv$ **if** $m = min(M)$ **then** $max(M)$ **else** $max\{p \in M \mid m \succ p\}$. In both cases is total order required.