Vrije Universiteit *Amsterdam*
Department of Computer Science
Faculty of Sciences

A Versatile Anycast Framework for Distributed Servers

Master thesis by:

Willem van Duijn
wdn700@cs.vu.nl
Specialization: Internet & Web technology
Student number: 1273310

Supervisor:

Dr. Guillaume Pierre
gpierre@cs.vu.nl

February 2008

Abstract

The use of a single server in client-service based services imposes limitations in sustainable load and availability guarantees. Within a distributed server, clients are distributed over multiple servers, disseminating the load. However, traditional mechanisms to distribute clients are seriously limited in functionality or scalability, and impose restrictions on the application.

Versatile Anycast is a new mechanism that offers great flexibility, but is difficult to use. This thesis discusses the requirements for a versatile framework, allowing one to build transparent distributed servers without additional complexity over traditional, single-node servers. Additionally, the framework allows one to handover a client in mid-connection, allowing a wide range of new applications.

The proposed framework has been implemented, and an example distributed server build using the framework has been successfully deployed on Grid'5000, an experimental grid platform.

Keywords

Virtual Anycast, Server framework, Distributed server, Client distribution, TCP handover, Anycast functionality.

Acknowledgements

I would like to thank assistant professor Guillaume Pierre for the support and help along the thesis, especially for the constructive criticism that helped me to improve both the framework and this thesis.

Secondly, I want to thank Michal Szymaniak, who has implemented Versatile Anycast on which my work is based and for helping me out resolving some of the issues encountered.

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https://www.Grid5000.fr). I would like to address special thanks to Yvon Jegou for giving me access to the platform and addressing my questions.

Finally, I want to thank my parents for their unconditional support and for giving me the opportunity to study, and my friend Arjan for motivating me at times of frustration.

Contents

Chapter 1

Introduction

Since the Internet gained a public face in the nineties the number of users worldwide has grown tremendously. Next to the major services like the World Wide Web, E-mail and Usenet, a large number of new services are gaining attention, such as Voice over IP and Video on Demand. Increasing numbers of companies are depending on their online presence, and they have commercial interest in giving their customers a high level of service in terms of availability, reliability and response times.

Many Internet services are designed around a client-server architecture. In this model a client application resides at the customer side, which issues requests to the company's application server. Typically, a single server is used to serve multiple clients, while each client uses only a single server to issue requests to. The use of a single machine imposes limitations in sustainable load and availability guarantees. When the server's resources become saturated, new requests cannot be processed until resources are freed, requiring clients to wait and causing new connections to fail.

As a single server is limited in its resources in terms of computing power and bandwidth, many popular services now depend on a cluster of servers to respond to incoming requests. These replicas share a single contact handle that is known by the clients, effectively turning the cluster into a single virtual server using the combined resources of its members. These clusters are often built using a proxying front-end that forwards incoming traffic to one of the servers, without the clients being aware of this distribution. Each server acts as a replica, being responsible for a fraction of the total number of requests. The use of multiple servers also potentially increases availability, because the failure of a single replica should not affect others. Instead, the proxying front-end stops redirecting to the failing replica, while still redirecting new request to the remaining ones.

Deploying multiple replicas increases the scalability of a single machine in terms of processing power, memory and/or network bandwidth, but it does not address all limitations of a single machine. A failure of the proxying front-end results in clients being unable to connect to any replica, even when they are fully operational. Furthermore, having all replicas at the same location makes the system vulnerable to network disruptions and threads like fire, earthquakes or power outages. Placing replicas at multiple geographically dispersed locations limits the impact of adversity, resulting in higher availability.

Another advantage of using multiple locations is that the system can have more network resources. Although single clusters are scalable in resources offered by computer hardware, all replicas typically share the same network. For bandwidth intensive applications like Video on Demand, this network can become saturated. Using multiple locations can distribute this load over multiple networks. Sometimes, the use of a single location is not even possible because the resources utilized by the distributed system are located on multiple sites. For instance, grid-applications typically depend on the combined computing resources of a large number of machines that are located on several interconnected networks.

Spreading replicas over multiple geographically dispersed locations can also improve customer satisfaction due to lower communication delay. With a single network location, a large fraction of the clients resides at a large network distance from the cluster. This large distance introduces a noticeable delay in communication, and for some services like VoIP, high response times can be frustrating for end-users. Additionally, by lowering the number of links between the replica and the client, congestion of traffic is less likely to occur, increasing the quality of service.

The challenge with geographically dispersed replicas is to attach clients to the most suitable replica. Ideally, this distribution should be done transparently to the client, i.e. the client application should not be aware that multiple replicas exist. Legacy applications and protocols often assume a single server entity that can be contacted using a single stable contact handle, offering no support for multiple replicas. But new applications will also benefit from distribution transparency. Revealing the set of replicas to the client would require the client to select a replica and handle subsequent failures by itself, introducing more complexity in the client application. Moreover, distributing and maintaining the list of replicas can be expensive or even impossible in case the set is very dynamic or the selection is based on volatile information like current replica load.

Server clusters using a switching frond-end can distribute clients transparently to the client application. However, the approach taken by these clusters is not suitable for geographically dispersed replicas because they are no longer located on the same network as the frond-end. For servers located on the same network as the frond-end we can neglect the delay introduced by forwarding traffic, so the round trip time to a replica for a client is equals the round trip time to the frond-end. But within a wide area network, this delay becomes noticeable by the end user, resulting in even higher response times. Additionally, a frond-end proxying all traffic does not make the distributed server more resistant to adversity because its failure will cause all replicas to be unreachable by clients.

1.1 Requirements

Because the techniques for distribution of clients over a local server cluster do not scale over multiple locations, other methods are needed to build distributed systems with distribution transparency. The distribution mechanism should be scalable in both the number of clients as well as the number of replicas, offering reasonable guarantees in terms of sustainable load and availability. It should support Internet scale deployment, so clients and replicas can be located anywhere on the Internet. The latency introduced by the distribution should be kept to a minimum, while still offering fine-grain control over how clients are distributed over the available replicas. In addition, the distribution should be resilience to changes in the set of replicas as replicas may join and leave the system over time, as well as to sudden changes in the volume of requests – for example during a flash crowd. The mapping of clients to replicas should be updated as soon as possible to guard against clients unable to connect because an outdated mapping was used to attach the client to a replica that left the network recently.

As different applications have different requirements, the distribution must be versatile in that it can be used in a wide range of applications. While some applications distribute clients according to resources at each replica, such as current load or remaining bandwidth, some may want to distribute clients according to the request being made.

Current solutions to distribute the clients over multiple replicas are discussed in chapter 2 of this thesis. Some of these solutions cannot be used on an Internet scale because they either require a forwarding frond-end or a shared network. Other solutions require support from the client application to either join an overlay network used for replica search or state dissemination. Solutions that do not require client side support offer only very coarse grain control over the mapping of clients to replicas. Changes in this mapping require some time to take effect, making them bad candidates for distributed systems with high replica churn or varying volumes of requests. Neither of them allow for content-based redirection in which the client's request dictates the most suitable replica. Additionally, the attachment is fixed so the client cannot be mapped to another replica during its connection.

1.2 Versatile Anycast

In [11] Versatile Anycast is proposed as a new mechanism to distribute clients using functionality provided by Mobile IPv6 (MIPv6) [2, 7]. MIPv6 is an extension to the IPv6 protocol that allows a node to be reached on its fixed home address, even when it is using a foreign network. Versatile Anycast exploits the address-translation mechanism of MIPv6 to switch traffic between the replicas. This traffic switching enables Versatile Anycast to handoff each individual request targeting the contact address to a selected replica.

MIPv6 is implemented within the client's Internet layer, leaving higher layers untouched. Therefore, Versatile Anycast can support legacy clients without any modification. Currently, most modern operating systems offer support for MIPv6 [19-21], while legacy operating systems can still be used to access the system, using a transparent fall-back mechanism.

Versatile Anycast allows the client-server architectures to evolve from a single server machine to a distributed virtual server without affecting the client architecture. It does this by decoupling the conceptual communication model from the actual communication model [9] using the above mentioned switching techniques. Additionally, Versatile Anycast provides means to handover a client in mid-connection, without the client application being aware. The replica serving the client can decide to handover the connection to another replica. Using Versatile Anycast, the traffic sent by the client is then switched from one replica to another, after which the first replica can leave the system without disrupting the client's communication. Besides supporting graceful departures of replicas this technique can also be used to distribute clients based on the content of the received request – enabling content-aware redirection. An example of such an application is a partitioned network file system, in which the name of the requested document is used to distribute the client to the replica currently in charge of the document.

1.3 Contribution to Distributed Servers

Although Versatile Anycast is transparent to clients, it requires support from the applications server. The server software has to create and maintain a contact address, which is used by clients to connect to. This address is then registered to a home agent, which is able to intercept traffic for this contact address, and forward it to a selected replica within the system. This selected replica must listen for new service requests issued by clients, and distribute each client individually. Therefore, most replicas do not accept a client directly, but service clients distributed to them.

The current implementation of Versatile Anycast offers a low level interface to control the binding of the contact address to a selected replica, and switch traffic from a client to another replica. The application itself is required to maintain the contact address and orchestrate the distribution of clients. These requirements break away from traditional server application design.

The main contribution of this thesis is to propose a versatile framework that allows one to build transparent distributed servers based on Versatile Anycast. Because different applications have different needs, we designed the framework to be versatile in that it can support a wide variety of applications. We do this by separating the distribution *mechanism* from the distribution *policy*. Versatile Anycast will be used as the distribution mechanism, while the policy is responsible for mapping clients to replicas. Because each application has specific requirements on this mapping, the distribution policy is not fixed, but can be controlled by the programmer. The framework will support a range of common drop-in distribution policies, but will also allow one to implement their own, based on the requirement of the application. Examples of common distribution policies are redirecting randomly or to the most nearby replica in terms of network metric. Applications distributing clients in a content-aware manner are likely to implement their own custom distribution policy.

This thesis is structured as follows. In chapter 2 we give an overview of traditional mechanisms of implementing anycast-like functionality. In chapter 3 the principles of Mobile IPv6 are explained before giving a more in-depth overview of the inner workings of Versatile Anycast. Chapter 4 presents the public interface to our framework, while a simple example illustrates the framework's ease of use. In chapter 5 an overview is given over the framework's architecture and implementation. In chapter 6 the framework is evaluated, by means of an example distributed application that has successfully been deployed on Grid'5000. We conclude with chapter 7, in which we discuss future improvements on the framework.

Chapter 2

Related Work on Client Distribution

Because the popularity of a service can overrun the capacities of a single server we need mechanisms to deploy multiple servers and distribute clients among those replicas. Different applications have different requirements, so the ultimate method for distributing clients should not only be scalable but versatile too. The distribution mechanism should fit any service, without imposing restrictions on the flexibility and control on how clients are mapped to replicas.

Before explaining Versatile Anycast and how it can be used to build an easy to use framework for building distributed systems, it is important to understand the requirements of the redirection mechanism necessary to support our framework. These requirements are stated in section 2.1. Section 2.2 discusses traditional client distribution mechanisms and shows that none of those satisfies all our requirements.

2.1 Requirements

Most services are implemented using a client-server computing architecture, partitioning functionality in a separate client and server application. Although both can be located on the same machine, they are usually located on two separate machines with a network interconnecting them. The client sends a request to a server, which accepts the request, processes it and responds with a reply. An example is querying a database server: the client application requests data using a query and the server responds with the result set.

Multiple clients can use the same single server, using a well-known contact handle: either a hostname or an IP address. In case the number of requests overruns the sustainable load of the single server, multiple copies of the server application can be deployed. Such a copy is often referred to as a replica. A *distribution mechanism* is responsible for distributing requests over the available replica servers, partitioning the load over those machines. Each replica then processes the request the normal way, and sends back the reply to the client, possibly without any interaction with other replicas.

No client-side support

In order to support existing client-server applications, the distribution mechanism should not depend on client-side support, i.e. the distribution should be done transparent to the client. Most applications assume a single application server to which they connect using a single fixed contact handle. But if popularity or quality requirements demand that multiple replicas are used, problems arise because the client is not aware of multiple servers being able to process its requests. Modifying the client application to support the new server architecture can be hard and requires users to update their software. As the service provider may have no direct control over the client application, changing the client software is not always possible. An example of such a situation is a video player, used to view videos from multiple video content providers. A provider cannot modify this software because it has no control over its development. Requiring the user to install additional software is intrusive and often regarded as user-unfriendly, potentially limiting the service's popularity.

Even for new services that require new client software, client-side support is sometimes not an option because of the complexity involved. First, the client should be aware of the current available replicas. Dissemination of this information is may not be straightforward as the set of replicas can be very dynamic. Moreover, the service provider might not want to reveal this information because of competition or security issues. The dissemination of information becomes even more problematic in case volatile replica state is used to determine a suitable replica, such as the least loaded server. Either the client should be updated continuously, or it should download this information before making a connection. Both solutions involve complex mechanisms and require bandwidth, which might be scarce in case a low-speed wireless network is used like GPRS.

One solution for new services is to implement a special redirect-primitive, like the Location-header found in HTTP/1.1. Although this can be a reasonable solution, it is difficult to implement redirection in the middle of a communication session, as it requires support from both the application and the replicas involved. Consider a client which playbacks a video stream from a video stream server wanting to leave the network. The client can be redirected to another server offering the same feed, but great care should be taken to make sure the playback will not stop or stall temporally. In order to guarantee uninterrupted playback the clients must listen for the redirection primitive and seamlessly connect to the offered alternative replica and request the next data segment of the stream. This makes client application software more complex. Ideally, the client application should not be aware that it has switched servers, thus requiring no support from it at all. We refer to this feature as 'handover', in which the current replica hands over a client to another replica. The servers are then responsible for making sure the client's requirements are met.

Separating policy from mechanism

Because different applications have different requirements, the actual distribution of the clients must be adaptable in order to fulfill the application's needs. A distributed web server may want to distribute clients based on their proximity to replicas in order to reduce latencies. Other services like Video on Demand may take a different approach, and use the current status of each replica to distribute new connections so the total load is balanced evenly over all replicas. For this reason, we need to separate the *policy* from the *mechanism* when distributing clients.

The distribution policy is responsible for defining a mapping between clients and replicas, while the distribution mechanism uses this mapping to execute the actual distribution of clients. This separation makes it possible to support a wide variety of applications, as each application is able to control the policy without requiring changing the mechanism.

Content-aware redirection

Some services like distributed file systems may partition the set of valid client requests amongst multiple servers, so each server (or set of servers) is responsible for processing only a subset of requests instead of all. As a result, they cannot process all requests, so clients must be distributed using information contained in the request. If a node within the distributed file server receives a request it cannot fulfill locally, the node should locate another node that is able to serve the client and distribute the client. This is referred to as content-aware distribution, in contrast to content-blind distribution. Content-aware distribution can also be used in distributed systems to redirect similar requests to the same replica, increasing the chance of a cache-hit in case caches are installed at each replica.

A second separation between policies can be made between stateless and stateful policies. Round robin and random replica selection are examples of the former, because they do not account for the current status of the replica group. Instead, they only use static information about the current composition of the replica group. This information includes the current IP address or network location of the replica, so clients can be distributed to a nearby replica [10].

Stateful policies also take the current state of each replica into account, requiring the dissemination of this type of volatile information. Examples of state information about each replica are current processor load, remaining bandwidth or available storage. The Video on Demand example given above can use the current bandwidth usage to balance the load evenly. Another application involves resources available in computer grids. A client can issue a job requiring certain resources like memory and CPU-time. The node receiving this request will use its state information about each node to select a target that meets the client's demands.

The separation between mechanism and policy is important because it enables the flexibility to adapt to the application's specific needs, without requiring changing the mechanism itself. But the mechanism itself must be flexible enough in order to support a wide range of distribution policies. Consequently, the mechanism must be versatile, and not depend on service-specific primitives because an equivalent primitive may not exist in other service communication protocols. Next to content-blind policies, the mechanism must support content-aware policies. The latter requires means to inspect a client's request, without requiring the client to resent its request after distribution, violating the distribution transparency.

Besides these requirements to execute the distribution transparent to the client, the distribution mechanism should meet a number of important non-functional requirements.

Internet scale

Replicas can be located on geographically dispersed locations, possibly close to clients in order to reduce communication latency. A wide-area network like the Internet is used for both the communication between the replicas and for communication between a replica and a client. Most – if not all – of the infrastructure used for this communication is outside the control of the service provider. Introducing new protocols or additions to existing ones will slow down or even stop acceptance. Therefore, the distribution mechanism should be based on standard Internet protocols only, and acknowledge their capabilities and limitations.

Furthermore, communication over the Internet may be delayed or even fail completely, and the distribution mechanism should be aware of this and handle accordingly.

Easy to implement/use

The implementation and/or use of the distribution mechanism should be straightforward. It should not abandon the traditional easy-to-use client-server paradigm in which a single server is used. This allows reasonably skilled programmers to build distributed servers, without introducing a steep learning curve. Advanced functionality may still be accessible, but only if the programmer wants to make advantage of the full possibilities of the distribution mechanism.

No unnecessary delays

The distribution mechanism should not introduce delays that are noticeable by the end-users, as this factor influences the level of quality perceived. Some services issue a large number of sequential requests for small data-units, for example fetching a webpage that consists of multiple images and other embedded objects. If each request is redirected separately, the total added amount of delay could easily become too high. Although this might not be a considerable problem for services with long lasting sessions, the distribution mechanism should be aware of this and offer means to redirect subsequent requests to the same designated server.

Resilience to replica group change

Because large-scale distributed systems are increasingly often composed of unreliable nodes the mechanism should support frequent changes in the set of replicas. The joining or leaving of a replica requires changing the mapping of clients to replicas. The first event introduces a new potential target and no harm will be done if it will take a while for the distribution mechanism to actually start distributing clients to the new replica. But in the latter event, clients may be distributed to a replica that is not available anymore. Communication initiated by those clients will therefore fail, and users may experience downtime in case the client application offers no failover mechanism. For this reason, the mechanism should reflect changes in the set of replicas as soon as possible, to prevent clients from connecting to a replica that is not dictated by the current policy. Besides supporting rapid changes in the mapping of clients and replicas, the remapping should not affect other clients in that their communication will be disrupted.

Easy replica deployment

Adding a replica to the distributed system should be as simple as possible, in terms of configuration of both the machine and the network. Moreover, the party deploying the replica may have only limited control over the network, because it is often shared with other users. The distribution mechanism should be aware of this limitation, and ideally it should support every machine with a standard Internet connection.

Scalable in number of replicas and clients

Services must be able to serve vast amounts of clients by distributing these clients over a large set of replicas. It should be possible to deploy or retract replicas without halting the service or breaking existing communication between clients and replicas. Ideally, the distribution mechanism should not impose a limit on the number of replicas nor the number of clients the distributed system can support.

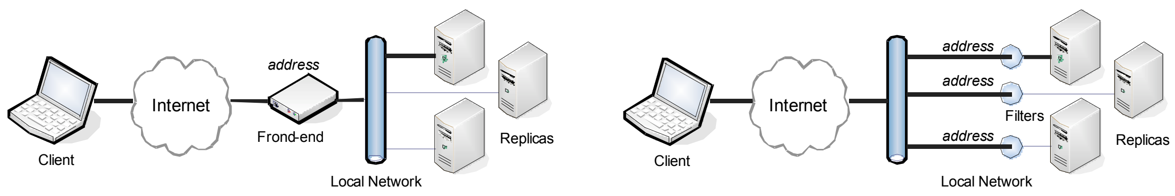

Fig. 1 Local server clusters (a) with frond-end and (b) shared address

Fine-grain control over the distribution of clients

A stateful distribution policy can exploit knowledge on the state of each replica to select an optimal target. Because these states are highly volatile, the mapping of clients to replicas should change continuously. The distribution mechanism should reflect these changes as soon as possible in order to support the policy correctly.

For content-aware redirection it is often not possible to compute the client-to-replica mapping in advance, requiring that those decisions are made for each client separately in real-time. Therefore, the distribution mechanism must allow this kind of fine-grain control over the distribution.

2.2 Traditional Distribution Mechanisms

Now we discussed both the functional and the non-functional requirements of the ideal distribution mechanism, this section discusses traditional mechanisms and show why they do not meet our demands. Recognition of these limitations is important because it proves the necessity of using Versatile Anycast to build a truly transparent framework for building distributed servers.

2.2.1 Local Server Clusters

Traditionally, service providers deployed servers with higher capacity in terms of bandwidth or computing power in case the volume of clients tended to overwhelm the server. Nevertheless, this solution only scales up to a certain point, requiring a scale out: deploy multiple servers, each server acting as a replica and responsible for processing only a part of the incoming requests [3]. A natural choice is to locate these replicas on the same network, with an external entity being responsible for distributing received request among the available replicas. This front-end forwards each request to a single selected replica and sends the response back to the client, as shown in Fig. 1 (a). It can do this content-blind at the network layer, or by inspecting the actual request at the application layer. The latter requires more logic but makes it possible to forward identical requests to the same server to improve on cache hits or partition the service over the available machines.

This setup succeeds because the frond-end can forward more requests than a single replica can handle, as the latter has to do execute the requested while the first only forwards traffic. However, this proxying frond-end has a severe limitation: it will function as a bottleneck in case the amount of traffic exceeds its forwarding capacities. One solution is to eliminate the need to forward traffic back from the replica to the client. Instead, the replica will do this directly, eliminating half of the traffic. In order to eliminate the web switch completely, some solutions partition the clients over the replicas in advance, for example by using the IP addresses of incoming traffic. Each replica is then hosted on the same physical network, able to listen for all traffic directed at the contact handle. A replica only processes and responds to traffic originating from clients within its own unique delegated set using a filter, as shown in Fig. 1 (b). But this solution is not very resilient to replica group changes in that it is impossible to repartition clients without disturbing current communication. Moreover, this solution still has scalability issues because of the limited capacity of the single shared network.

As explained in Chapter 1, clients at a large network distance from the cluster will experience high latencies degrading perceived quality of service. The use of multiple replicas or clusters close to the clients can reduce these latencies. However, the nature of the above shown solutions makes it impossible to implement them on separate networks. A web switch can be used to forward traffic to remote replicas, but because the reply has to travel back through the web switch even larger delays are introduced. Replies sent from a remote replica directly to the client (IP triangulation) must carry the web switch's IP address in order to get it accepted by the client, but spoof-prevention on routers cause these packets to be dropped.

2.2.2 Client-side Replica Selection

Instead of letting the service select the best-suited replica for a client, another method depends on the client to select its own. Before connecting to the service the client fetches a list of current available replicas. This can be done from a fixed URL, a directory service or using another bootstrap method. The client then selects a target replica from that list and continues as if the selected target replica was a traditional server with a single contact handle.

The main advantage of this solution is that the client can select a replica for which traffic has the lowest latency. For example, it can measure the round trip times by sending probe packets towards the replicas on the list, and select the one with the lowest round trip time. Probing without cooperation from the client can be far more tedious – especially in the presence of intrusion-detection software [6].

However, client-side replica selection can only support a limited set of distribution policies. Content-aware distribution policies require the client to download the mapping of requests to replicas, so it can select a replica that is able to process its request. This requires additional client application logic, and downloading the current mapping is likely to require substantial bandwidth.

Problems arise when the information contained on the replica list changes rapidly, for example due to high replica churn. The client must soon invalidate the old list and fetch a fresh one, imposing a high load on the bootstrap mechanism and utilizing more bandwidth.

Instead of regularly downloading a fresh list, updates can also be disseminated to the clients using push-functionality. However, efficient dissemination of these updates can introduce a prohibitive additional complexity. Moreover, the collection of state from a large number of replicas and the dissemination of this data to a vast amount of clients may impose restraints on the scalability of the distributed system.

For many applications, the complexity and delay involved with client-side replica selection are too high. Additionally, changing the distribution policy requires updating the client software or configuration.

2.2.3 Overlay Networks

The problem of outdated replica lists caused by transient replicas, or volatile information used as a parameter for the distribution policy, can be solved by using overlay networks such as distributed hash tables (DHT). These networks deliver traffic from a client to a single replica, without requiring the client to make the decision about which replica to use.

Both clients and replicas are placed in a virtual network topology and an identifier is assigned to them. Each node knows only a small fraction of other nodes in the system, in order to eliminate the need to maintain a global membership view. Every node is equipped with routing tables that enable the sending of messages to another node owning a particular identifier. To this end, identifiers have a complete order and a function defines the distance between two identifiers. A node either knows the owner of a particular identifier, or has a routing table entry pointing to another node whose identifier is closer to the identifier of the final destination. Messages are iteratively routed until they are delivered to the node owning the destination identifier.

This scheme can be exploited to implement anycast-like functionality. The service itself is given an identifier, and a spanning tree is constructed containing all replicas, with the node owning the service identifier as the tree's root. This tree is constructed by requiring each replica to send a join-message towards the root. Intermediate nodes routing this message are added to the tree, until a node is reached that is already part of the tree. Requests are then sent to the service identifier, and due to the properties of DHTs, each request will eventually be received by a member of the spanning tree for the service. This member then starts a (dept-first) search for a replica ready to accept the request. This makes the overlay network very scalable, as no state information needs to be disseminated.

An example is shown in Fig. 2. Using the routing tables at each node, a spanning tree to the root is created that includes all the replicas, but clients may be included too. The client sends its request to the root of the service, using the identifier 770. As it does not know the real address of this node it will consults its routing tables, and discovers the real address of a node holding identifier 600, which is closer to 770. That node then routes the message to a node holding identifier 700. Because that node is a member of the spanning tree the search for a replica will start from there. This can be done by recursively examining the child nodes of the tree. If no descendant replica accepts the request, the parent of the node that initiated the search will continue the search so other branches will ultimately be considered.

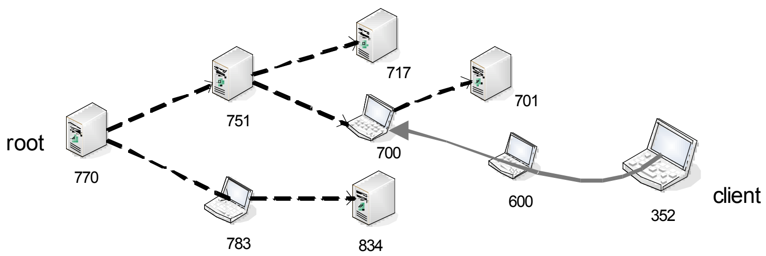

Fig. 2 Overlay network with spanning tree for replica search

Overlay networks make it possible for a replica to decide for itself whether or not it will accept a given request. If the replica cannot process the request, it will reject it and the search continues. Still, it will be difficult to achieve fair load balance with this mechanism because each replica is only concerned with its own state and will base its decision solely on that data. Additional mechanisms are required to disseminate state information of the replicas, so each replica can compare its own state against others. Moreover, the replica selection is more or less directed by the routing algorithm and the maximum number of search steps (iterations) in the tree. Consider for example the tree in Fig. 2. The replica holding identifier 834 will only be considered if all replicas in the branch drawn on top reject the client, forcing the search to continue from the root itself. If not carefully implemented, this can lead to situations in which one branch contains high-loaded replicas and another branch contains replicas serving no clients at all.

Besides requiring additional mechanisms to ensure the distribution policy is met, the overlay network mechanism requires major modifications to the client application. The clients must join the overlay network first before being able to send out service requests. Additionally, the client may be included in the spanning tree, requiring it to support and execute replica searches on behalf of other clients. This involves additional bandwidth and processing power, which can be sparse for some user hardware like cell-phones. This reduces its application to completely new services like peer-to-peer streaming video, in which a client is also potentially a replica. For our type of distributed systems, the added delay required to join the overlay network and the time required by the search to finish can easily become unacceptable to the end users. Most services require only a single lookup for a suitable replica server, and for these services the complexity introduced by this mechanism are too high. Additionally, it breaks away from the classical client-server architecture because it uses the overlay network for communication instead of direct communication with the replica.

In [6] a special-purpose overlay network is proposed named OASIS. Instead of requiring each service to build its own overlay network, OASIS tries to create a shared infrastructure that can be used by many services. It features a flexible and advanced mapping policy allowing for a reasonable level of control of how clients are mapped to registered replicas. Clients are required to first query the OASIS overlay in order to find the address of a suitable replica. For legacy applications alternative mechanisms can be used. The first one is to use a service-specific redirection primitive, like the Location-header in HTTP. The client first contacts node in the system that is running client standard OASIS-software, using a fixed contact handle. This node then queries the overlay network and sends back a redirection primitive pointing to the replica returned by OASIS. For most applications such

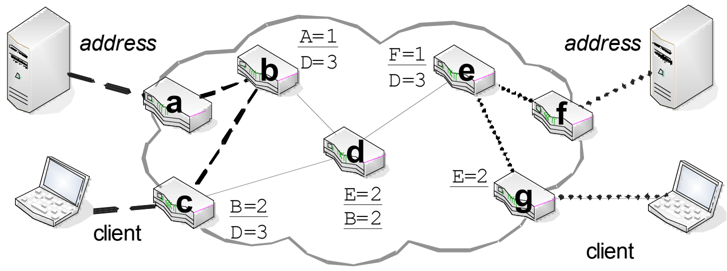

Fig. 3 Route-insertion based anycast

primitives may not be available, and therefore the authors propose to use DNS redirection, which is discussed in section 2.2.5.

2.2.4 Route-insertion Based Anycast

Instead of letting the client select a target replica, the routing protocols used on the Internet can be exploited to direct client traffic to the nearest replica. These routing protocols are designed to select the best or shortest path to a target network, enabling efficient inter-network communication on the Internet.

Within an anycast-based distributed server, all replicas use the same IP address (anycast address) and let their home networks advertise this single address. These advertisements will propagate through the Internet using the standard mechanisms. Each router receiving such an advertisement will consult its routing table whether or not the received route to that network is smaller than the one it has currently selected. In case the route has a lower network metric it will use the advertisement to route future packets towards the corresponding replica. Therefore, datapackets sent by a client will always be delivered to the nearest replica in terms of network metric.

An example is given in Fig. 3. Two replicas both advertise the same address using the routers on their home networks (router ‘a’ and ‘b’). Router ‘a’ advertises its route to the anycast address to router ‘b’ with a cost of 1, denoting the number of hops in this example. Router ‘b’ advertises the route to ‘c’ and ‘d’, which both remember that they have a route to the anycast address using router ‘b’ with cost 2. This process continues throughout the whole network, until every router knows how to route packets to the anycast address. Now the replica on the right instructs its router (‘f’) to advertise its presence. Router ‘e’ will receive this announcement and computes that the new route is shorter then the route its currently using, and therefore decides to use ‘f’ as the default route for the anycast address. This process continues, but as routers closer to the left replica receive the announcement the cost for the route will grow, and eventually the cost will be higher than the cost of the current known route. This will cause some routers in the network to route traffic for the anycast address to the left replica, while other route towards the right replica. The client on the left for example will use the path ‘c’-‘b’-‘a’-left replica, while the one on the right will use ‘g’-‘e’-‘f’-right replica, splitting the load over the two replicas.

Every network hosting a replica must advertise the network prefix of the anycast address in order to receive client traffic. In case a network is hosting multiple replicas it should be anycast aware in that it must partition traffic among the available replicas, for example by the use of a proxying frond-end. Most users do not have means to insert routes into the network, as it requires access to routers responsible for inter-network connectivity. The reason that insertion of routes is reserved to only a few people is that misuse can easily lead to problems affecting multiple networks, possibly even the complete Internet. Secondly, if many services were deploying route-insertion based anycast this would lead to an explosion of advertised prefixes that are difficult or even impossible to aggregate. This is considered bad-practice, as it will lead to larger routing tables, hurting the performance of routers.

Anycasting is also very inflexible: replicas are always selected based on the network metric and other parameters such as replica load are ignored. Because all traffic regardless of the content is routed towards the same replica, content-based redirection is impossible. The method is also very coarse grain as whole networks are assigned to a replica irrespectively of the numbers of clients contained within that network. Besides this, route-insertion based anycast does not perform well in the case of replica transience. If a replica goes down it will function as a black hole in the system. Traffic sent by clients on networks using routing tables pointing to the dead replica cannot be redirected to a functioning replica. The service will be discontinued for those clients until the route to this unavailable replica is withdrawn and a route to a functional replica is selected. Frequent changes in the set of replicas cause multiple route advertisements and withdrawals to propagate through the Internet. This results in inconsistent views of the network, again leading to situations in which no single replica can be reached by affected networks. Moreover, existing connections between clients and replicas are broken after a routing update is committed.

2.2.5 DNS Redirection

In DNS based redirection, anycast-like functionality is achieved by having all replicas share the same hostname. A client willing to send a request to the known shared hostname will first have to resolve that name to an IP address using the DNS system. The authoritative DNS server for the hostname is modified in such a way that it will not return the same fixed IP address on each lookup. Instead, it will select one of the addresses of the available replicas upon each lookup-request. This effectively distributes clients over all the replicas that are returned by this authoritative DNS server.

This scheme is completely transparent to clients, and exploits the standard procedures of the DNS system. Because this mechanism is based on hostnames it is also very versatile in that it can be used for virtually any service. It has successfully been deployed as the key mechanism to distribute a vast amount of users over a large number of replicas. One example is Akamai [5], a content delivery network owning thousands of servers throughout the Internet. An example hostname used by Akamai is 'a7.g.akamai.net'. Like all hostnames, this name is resolved from right to left. During the lookup of the complete hostname, the authoritative DNS server for 'g.akamai.net' responds with a second Akamai DNS server near the requester. This second DNS server is responsible for requests coming from a particular region, and maintains status information about the replicas within that region, for example their location and current load. That DNS server is then used to resolve 'a7.g.akamai.net' to the IP address of a replica. While Akamai uses a two-trap mechanism to partition the replicas according to their geographical location, a single trap solution can also be used. In that case only one modified authoritative DNS server is used that is aware of all replicas.

Although this mechanism could provide acceptable control over client distribution, caching mechanisms plague it. Typically, multiple clients are configured to use the same nameserver, which is responsible for resolving hostnames. The nameserver caches results, to reduce the number of hostname lookups and to improve speeds. Although DNS allows one to specify a time to live (TTL) value for returned results, many DNS servers are configured to ignore values that are below a certain threshold. Hence, all users of the same nameserver trying to resolve the anycast address (DNS hostname) will get the same replica address. So a single lookup at the authoritative DNS server for the anycast hostname may result in multiple clients mapped to the same replica. Research [4] has showed that on average only less then 5% of the total requests is controlled directly by this DNS mechanism, leaving a 95% that are distributed indirectly using the caches[1].

Caching also makes the mapping of clients to replicas stale, limiting its flexibility and resilience to peer transience. In case a replica goes down those stale mappings cannot be circumvented resulting in unavailability for some clients. Second, client applications typically only resolve a name once, ignoring the TTL altogether. Switching the client to a different replica is therefore almost impossible to do without using service specific redirection primitives. This makes content-based redirection impossible to implement.

Finally, the authoritative DNS server cannot determine the IP address of the client trying to resolve the anycast address using a nameserver. It can very well be the case the nameserver used in the lookup procedure is on another network, resulting in a suboptimal replica being returned in case replicas are selected based on their network distance to the client.

2.3 Discussion on Traditional Mechanisms

Different applications have different requirements. Some applications like DNS root servers only require coarse grain control over the distribution of clients to allow raw load balancing over a fixed number of stable replicas. Others, like Video on Demand, require finer control over this distribution because they require balancing the load more evenly. Webservers and other services that require the user to wait for the server's reply to be received may want to distribute the clients in such a way that the delay introduced by communication is minimal. Those types of services therefore suffer from distribution mechanisms that require long execution times, like joining an overlay network. However, such delays are less important if the distribution is only executed once at the beginning of a long-lasting session such as the playback of a video stream. Distributed file servers may want to inspect the content of the request before distributing the client, delaying the actual redirection until after the complete request is received.

In order to support all these different types of applications, the distribution mechanism must support all requirements discussed in chapter 2.1. An overview of both our requirements and traditional distribution mechanisms is given in Table 1. If a mechanism meets a certain requirement a plus sign is shown. A minus sign in case the requirement is not met at all. A plus-minus symbol is displayed if for some situations the requirement can be met, although it may be hard to accomplish.

Important to note is that none of the traditional distribution mechanisms meets all our requirements. This means that neither of them qualifies as a versatile distribution

[1] Although these figures were the result of research performed in 1998 we have very little evidence that the situation has changed considerably since.

	Local clusters	Client-side	Overlay networks	Route-insertion	DNS redirection
Versatile	+	-	-	+	+
Content-based	+	+/-	+	-	-
No client-side support	+	-	-	+	+
Fine-grain	+	+/-	+	-	-
Internet-scale	-	+	+	+	+
Ease of use	+	-	-	+	+
Easy replica deployment	+/-	+	-	-	+
Scalable	-	+	+	+	+
Replica transience	+	-	+	-	-

Table 1 Overview of classical distribution mechanisms

mechanism, supporting many different services. They can only be used if one understands their limitations, making it sometimes necessary to change the application requirements.

But some applications have a stringent requirement set that is supported by neither of these traditional mechanisms. For example, a distributed fileserver requires content-aware distribution, without breaking a legacy service protocol like NFS. Currently such applications cannot be built without sacrificing distribution transparency. To support such applications, the distribution mechanism must support mid-connection handovers, in which a client is taken over by a selected replica without disturbing communication.

We believe Versatile Anycast meets all our requirements necessary to build an easy to use framework that allows one to build fully transparent distributed systems without adding complexity over traditional single-server architectures, nor sacrificing application freedom caused by limited functionality or usability of the distribution mechanism. Versatile Anycast is described in detail in the next chapter.

Chapter 3

Related Work: Versatile Anycast

Versatile Anycast is a redirection mechanism that can potentially support all requirements discussed in the previous chapter to support our versatile framework for building distributed systems. We will first give a short introduction on the Mobile IPv6 protocol, as Versatile Anycast relies on it. The next section then shows how Versatile Anycast exploits MIPv6 to implement client redirection using a single contact handle. Understanding how Versatile Anycast works is important because its characteristics are reflected in the design of our versatile framework, which will be presented in the next chapter. A full in-depth description of Versatile Anycast can be found in [8, 9, 11].

3.1 Mobile IPv6

By default IPv6 has no build-in support for mobility. Therefore, if a mobile node leaves its home network and joins a foreign network, its IPv6 address must change in order to maintain connectivity. While the mobile node is away, traffic sent to its home address (HoA) cannot be delivered since standard network protocols route traffic to its home network instead of the foreign network it has joined. A correspondent node initializing a connection is likely to be unaware of the mobile node's temporal address, while existing connections between mobile node and correspondent nodes will be broken upon joining a foreign network, disabling communication. Mobile IPv6 is an extension to the IPv6 protocol that enables a mobile node to be reached on a fixed home address and continue communication, even when it is switching networks [2, 7].

3.1.1 Tunneling Mode

MIPv6 makes use of a home agent on the home network, which will act as the mobile node's representative while it is away. In case the mobile node joins a foreign network it will be configured with a new IP address, using standard IPv6 mechanisms such as stateful or stateless address auto-configuration. This foreign IP address will act as a care-off address. Traffic targeting the mobile node's home address will be intercepted by the home agent, and forwarded to the care-off address. A care-off address is not stable in that it is only used on the foreign network. Therefore, this address will change in case the mobile node moves to another foreign network, requiring the mobile node to report its new care-off address to the home agent each time it changes networks. Note that traffic carrying the mobile node's home address as source address must also be forwarded by the home agent, to prevent spoof-prevention on routers from dropping the packages.

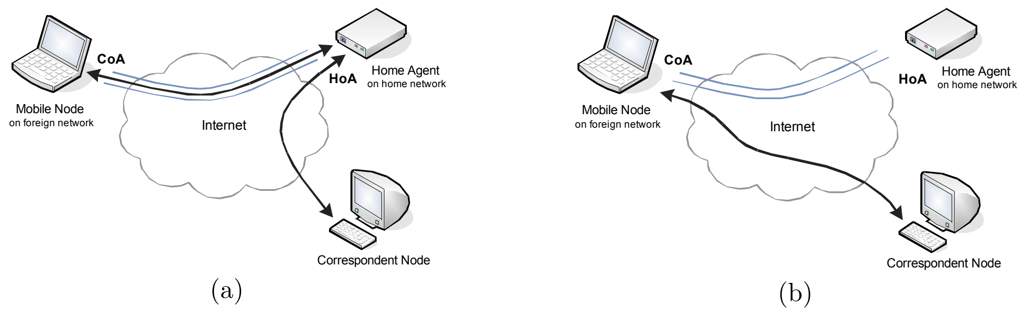

Fig. 4 Modes of operation in MIPv6: tunneling (a) and route optimization (b)

The home agent can be implemented by a router capable of intercepting all traffic sent to nodes that require mobility. Such a node must register its stable home address with the home agent prior to changing networks. During registration, the node and home agent create a security association using IPsec. The security association between the home agent and mobile node is shown in Fig. 4 by means of a tunnel between the two nodes.

In addition to the security associations, a home agent keeps track of the current location of mobile nodes using a binding cache. This binding cache consists of home address - care-off address pairs of nodes currently away. The home agent then listens to IP packets destined to these home addresses, and forwards them accordingly. Upon joining a foreign network, the mobile node sends a binding update to the home agent, to inform it of the new care-off address to use. This message is sent securely using the security association, guarding against hostile takeovers by malicious nodes reporting a false care-off address to the home agent. The home agent then updates its binding cache to reflect the mobile node's current location, so traffic is forwarded to the nodes new care-off address using an IPv6-in-IPv6 tunnel.

This forwarding is shown in Fig. 4 (a). In the figure, the correspondent node sends traffic to the home address, which is consequently tunneled to the mobile node by the home agent using the care-off address. Reverse traffic is first sent to the home agent, which then forwards it to the correspondent node using the home address as the destination address. Note that the forwarding of traffic is completely transparent to the correspondent node, since this node does not have to be aware of mobile node's current location.

This mode of operation is similar to the proxying frond-end discussed in section 2.2.1, although now nodes can be located anywhere on the Internet. Like the frond-end, the home agent is required to forward communication in both directions, resulting in additional communication delay. In case multiple mobile nodes are away from their home network the home agent's capacity or bandwidth might not be sufficient, reducing the bandwidth between a mobile node and a correspondent node.

3.1.2 Route Optimization Mode

Because of these drawbacks, the MIPv6 specification defines a second mode of operation, named route optimization. In this mode the correspondent node can be made aware of the mobile node's current care-off address. By doing so, the correspondent node can send and receive data directly using the temporally care-off address instead of using the home agent as intermediary, as shown in Fig. 4(b). This will reduce the round trip time and relieve the home agent, as it no longer has to forward communication traffic. Besides support from both the home agent and mobile node, route optimization requires MIPv6 support from the correspondent node to use the care-off address directly.

Initially, the correspondent node will regard the home address as an ordinary IPv6 address, and therefore it will use default IPv6 mechanisms to send traffic towards the mobile node's stable contact address. Depending on the current location of the mobile node, the traffic is either delivered locally, or tunneled by the home agent. The mobile node receives the traffic, and may decide to switch to the route optimization mode to increase communication efficiency. To this end, the mobile node must reveal its current care-off address to the correspondent node. This announcement is contained in a binding update (BU), which is used to update a binding cache at the correspondent node. Like the binding cache at the home agent, this binding cache consists of home address - care-off address pairs and additional administrative information about the binding. This binding cache is then used to switch traffic targeting a home address to the corresponding care-off address.

Because the mobile node and correspondent node have no security association setup in advance, other nodes can produce valid BU messages requesting any correspondent node to switch traffic to their IP address. To prevent against malicious parties trying to break or hijack communication, MIPv6 provides an authentication mechanism named return-routability. This mechanism proves to the correspondent node that the mobile node is the legitimate owner of the home address.

The return routability procedure executed before the first binding update, directly after the mobile node decides to switch to the route optimization mode. The mobile node simultaneously sends two messages to the correspondent node. The first is a Care-of Test Init (CoTI), which is sent directly to the correspondent node using the foreign care-off address. The second is a Home Test Init (HoTI), which is tunneled through the home agent using the stable home address. Upon receipt of both messages, the correspondent node responds with a Care-of Test (CoT) and a Home Test (HoT) message. Both are sent over the reverse path of their corresponding init message, so the home agent is required forward the HoT.

The HoT and CoT messages contain tokens that are used by the mobile node to reconstruct a key proposed by the correspondent node. Because the home agent only forwards (HoT) messages to authenticated mobile node's over a secure channel, the reconstruction of this key proves to the correspondent node that the mobile node is authorized to use the home address. Upon receipt of both messages, the mobile node can send a signed BU to the correspondent node. The correspondent node verifies the signature, and processes the BU in case the signature is made using the offered key. The BU is acknowledged using a binding acknowledgement (BA). After the receipt of the BA, communication can use the revealed care-off address, optimizing the path between correspondent node and mobile node. Note that the return routability procedure does not require initial configuration, such as shared keys, so it can be used with any correspondent node supporting MIPv6 route optimization.

3.1.3 Implementation of Mobile IPv6

Both the binding cache and the route-optimization procedure are implemented within the Internet layer of both the correspondent node and mobile node. These MIPv6 enabled layers add special routing headers to enable direct communication.

At the correspondent node, the Internet layer consults its binding cache upon receipt of a datagram from the local transport layer. If no entry is found, the packet is sent as normal. In case an entry for the home address is found, the mobile node's care-off address is used as the destination address and a routing header is added to the IPv6 header specifying the mobile node's home address. The package is then routed directly to the mobile node. The mobile node's Internet layer will detect the routing header, and process it like it was sent directly to the home address.

The same holds for reverse traffic: the mobile node's Internet layer will change the home address to the care-off address, and add a routing header specifying the home address so the correspondent node can identify the package. The latter then processes the package as like it was sent using the home address.

By implementing mobility in the Internet layer, it is completely transparent to the transport layer and onwards on both the mobile and correspondent nodes. Therefore, those layers are only concerned with home addresses and do not deal with care-off addresses directly. Even connection-oriented communication using TCP can be continued normally in case the mobile node changes its care-off address, without requiring any modifications to current implementations.

Next to retaining existing connections, another advantage is that applications running on the mobile node continue to use the home address for new outbound connections. One example is a client running on the mobile node, connecting to a server. The server application will only notice the home address, so security measures like address identification can still be used. The reverse is also true: if the mobile node is running a server application, clients can always use the stable home address to establish a connection, without the need for additional mechanisms to request the server's current address.

In case the correspondent node has no support for MIPv6, communication will be possible using the default mode of operation, requiring the home agent to tunnel all traffic. However, it is our belief that due to the increasing mobility of Internet devices such as laptops and PDA's, the required functionality will become standard in the near future. Currently, Windows XP and beyond [19] offer the functionality required for correspondent nodes to perform route optimization. Other operating system already have optional support for MIPv6 [20].

3.2 Versatile Anycast

Versatile Anycast can be used to create a distributed server, which appears to client as a single reliable server. This distributed server can be reached using a stable contact address, which stays the same during the communication session from the client's point of view. In reality, the distributed server consists of multiple replicas, possibly scattered over the Internet. Clients are accepted by some node in this replica group, and can be handed off amongst each other.

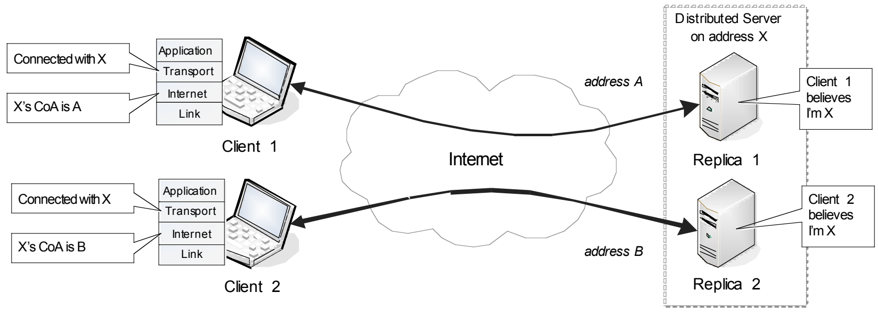

Fig. 5 Communication within a Distributed Server using Versatile Anycast

To this end, the replica group creates the illusion of a single mobile node, using the home address as the server's stable contact address. In the event of a handoff, the client will be notified that the server has moved to a new location, so it can switch traffic accordingly. Instead of moving the actual server, only the connection endpoint is passed from one replica to another, creating the illusion of a mobile node switching networks.

Each replica can reveal its own fixed address as the new care-off address, eliminating the need of a proxying front-end. The switching of traffic is done using standard MIPv6 primitives explained in section 3.1. Because each client must be notified separately of the fictitious mobile node's care-off address, we can attach different clients to different replicas by announcing alternative care-off addresses. This enables a distribution method in which clients can be individually attached to a selected replica, offering a very fine grain control on how clients are mapped to replicas.

An example is shown in Fig. 5. In this example two replicas share a single contact handle X, although both have distinct care-off addresses (A and B). Client 1 is connected with Replica 1, which is believed to be a normal mobile node using X as its home address. Client 2 also believes it is connected to a mobile node using X as its home address, but this client is connected to Replica 2. So, both believe to be connected with a mobile node owning address X, while in reality they are connected to different replicas using Versatile Anycast. Note that both the application and transport layers retain the illusion that they communicate with a single server listening on address X, as the translation between home address and care-off address addresses is confined within the Internet layer.

Versatile Anycast assumes a standard, unmodified MIPv6 implementation at the clients and home agent. The unmodified home agent must be able to receive traffic directed at the server's contact address, which is used as the MIPv6 home address. Other solutions using MIPv6's address translation as a distribution mechanism may assume a modified home agent to support the distribution, but because of our deployment requirement we will use standard router functionality that is currently already available. As a result, Versatile Anycast only requires support from the replica nodes.

In order to create and maintain the illusion of a single mobile server, Versatile Anycast acknowledges two important aspects. First, nodes within the system are unreliable and may crash or experience problems, even the node selected to receive traffic targeting the contact address directly. The second aspect is the need to offload clients from this contact node to other members within the replica group. These two aspects are discussed in the next two sections.

3.2.1 Anycast Address Management

One important characteristic of a server is that it services clients. To this end, it must wait for request, and accept them accordingly. The same holds for a distributed server using Versatile Anycast as the mechanism to distribute clients. At any time, one selected member of the distributed server must be responsible for processing traffic from clients that do not have a MIPv6 binding yet. Consequently, the home agent must forward traffic for the contact address to this single selected node. The replica in charge of this task is named the (current) contact node. Since the home agent is implemented according to the RFC specification, it is unaware of multiple replicas and as a consequence, there can only be a single contact node per contact address.

The contact address is created by one designated node during the initialization of the distributed server. Traffic targeted to this address must be interceptable by the home agent, requiring that the address has the same network prefix as the home agent. After creation, this unique contact address must be registered with the home agent and the security association must be setup. After registration, the bootstrap node will become the contact node.

In case this contact node crashes or leaves the network, the distributed server will be unavailable to new clients, because the home agent is not aware of alternative replicas to forward traffic to. To protect against this situation, the encryption key used in the security association between the home agent and mobile node is distributed over other replicas eligible to become the contact node. Such a nodes are referred to as a backup contact node.

In case a backup contact node wants to take over the contact handle, knowledge of the security association is used to impersonate the creator of the contact handle. This creates the illusion to the home agent that the mobile node switched networks and adopted another care-off address. After commitment of the new care-off address by the home agent, traffic sent to the server's contact address (home address as seen by the home agent) will be tunneled to the new care-off address. The old contact node will not receive this traffic anymore, and the backup contact node that changed the binding at the home agent will become the current contact node. By doing so, the distributed system does not rely on a single fixed node to accept clients, but increases its availability by allowing other nodes to take over this task.

3.2.2 Traffic Handoff

The contact node is likely unable to serve all clients by itself, requiring means to handover clients to other replicas. This handover mechanism can also be used by other replicas, enabling a wide range of new applications not possible before. One example is the downloading of a file that is striped over multiple nodes, each node hosting certain chunks of the file. In case the client has received all data from one node it can be handed off to the next node to continue the download - transparent to the download application.

Note that in this example the members of the distributed server may not be replicas in that they all own the same application data. Each node has its own distinct set of data, so it is only able to satisfy a subset of client requests, or clients during an limited interval of their

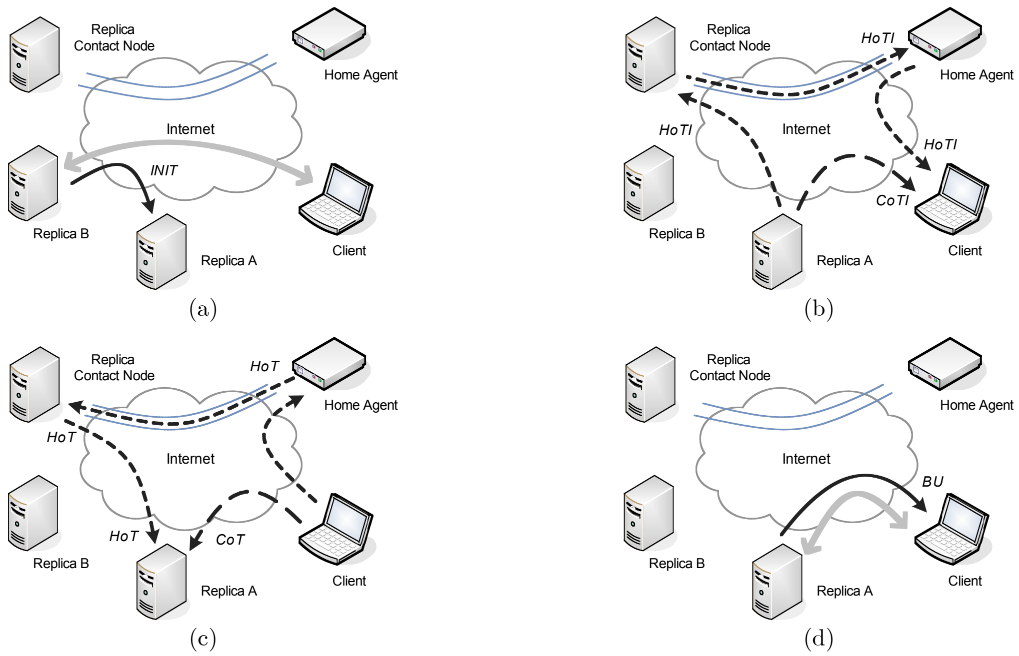

Fig. 6 Versatile Anycast handoff between two replica's

connection to the server. Although we acknowledge the differences between this behavior and that of true replicas, we will still refer to them as replicas because each replica offers the same type of service. Additionally, each replica should have the same means to handoff clients, allowing it to redirect the client to a more appropriate replica, depending on the application.

A handoff always involves a *donor*, initializing the handoff of a client to an *acceptor*, the replica that will take over the client, as shown in Fig. 6. In this example, Replica B acts as a donor handing off the client to Replica A, which will act as the acceptor. The handoff is initialized by the donor by sending a control message to the acceptor, as shown in Fig. 6 (a). After receipt of this control message, the acceptor will begin the execution of the return routability, so it will be able to send a binding update to the client to optimize communication.

The return routability procedure requires that a HoTI message is sent with cooperation of the home agent, as part of the proof that the initiator is the legitimate user of the address. The client then responds with a HoT message sent to the home address, which will be forwarded by the home agent. Receipt of the HoT proves to the client that the initiator is the owner of the address, because the home agent will only forward messages securely to the node that is registered for the specified address.

However, with Versatile Anycast the situation is different as the home agent is not aware of the existence of multiple replicas. The HoTI message sent by the acceptor directly to the home agent will therefore be discarded in case the acceptor's care-off address differs from the care-off address contained in the home agent's binding cache.

The same holds for the returning HoT message, which is sent directly to the home address by the client. The home agent will forward this message to the current correspondent node, as stated in its binding cache. From the perspective of Versatile

Anycast, the HoT message is therefore sent to the wrong replica, breaking the return routability procedure initiated by the acceptor.

For that reason, Versatile Anycast relies on the correspondent node to act as an intermediary, by requiring it to forward both the HoTI and HoT messages. Therefore, the acceptor first sends the HoTI message to the contact node (Fig. 6 (b)), which will then add a temporally mapping for the handoff. This mapping consists of the replica and client addresses involved in the return routability procedure. It then forwards the HoTI to the home agent, which delivers the message to the client. The resulting HoT message is then forwarded by the home agent to the correspondent node. The latter consults its mapping and forwards the HoT to the acceptor. The receipt of the HoT message is shown in Fig. 6 (c). After forwarding the HoT message, the contact node can delete the entry from its mapping, requiring only a minimum of state. The CoTI and resulting CoT messages are sent as normal.

After the acceptor has received both the CoT and HoT it can reconstruct the binding update key proposed by the client. This key is then used to sign the BU message, which receipt will result in a binding cache update at the client. After receipt of the BA, the acceptor can start sending data to the client, as shown in Fig. 6 (d). From this moment, the donor no longer has a valid binding with the client, so it will not receive further communication from the client unless the client is handed over back to donor.

3.2.3 Current Implementation of Versatile Anycast

Versatile Anycast requires support from the MIPv6 implementation running on the replica nodes. The current implementation [11] uses a modified MIPv6 daemon available from [21], running on the Linux operating system. This MIPv6 implementation puts only the absolute necessary infrastructural support in the kernel, moving most functionality to a daemon running in user space. The daemon is then responsible for installing and removing packet translation mappings required to process the special MIPv6 routing headers. Their goal is to get the modifications to the kernel accepted in the main kernel source, so users running Linux can add MIPv6 support fairly easily. As the current implementation of Versatile Anycast is based on this MIPv6 implementation, it is currently not possible to run these Versatile Anycast based applications on other platforms.

Replica nodes are then required to run this modified user space daemon instead of the original one. While this requires administrator rights over the machine running the replica, we believe it allows for easy deployment, as most machines running a replica will be in control of the service provider.

The implementation supports both aspects discussed in sections 3.2.1 and 3.2.1 using two sets of functions. The first set deals with the management of the contact handle, while the latter deals with handoffs. Application programmers can use these functions to create a Versatile Anycast based distributed server. As the actual mechanisms to control the contact handle and perform handoffs are contained in the modified MIPv6 daemon, all the API functions interact with this daemon instead of with the kernel directly. Communication between the distributed server application and the daemon is implemented using UDP packets.

Anycast Address Management

Before a client can be accepted or received using a handoff, the kernel must be given knowledge about the contact address used by clients to connect to the distributed server. The contact address must be attached to a network interface, so the distributed server application can bind sockets to it. Secondly, a binding cache entry must be created to allow rewriting of IPv6 headers according to section 3.1. Both actions can be accomplished by executing:

```
mip6_addr_passive(
        stuct in6_addr * contact_address,
        stuct in6_addr * careoff_address_handovers,
        stuct in6_addr * contactnode_address    );
```

The first argument is the contact address of the distributed server. The second argument is the care-off address revealed to a client in case it is handed over. The third argument is the address of the current contact node, required forward HoTI and HoT messages, as explained in section 3.2.2.

The MIPv6 implementation requires that the security association is setup in advance. Therefore, our Versatile Anycast implementation assumes that both the contact node and backup contact node's already have the knowledge to impersonate the creator of the address. In case one of these nodes wants to take over the contact handle, the binding at the home agent can be updated using:

```
mip6_addr_active(
        stuct in6_addr * contact_address,
        stuct in6_addr * careoff_address_handovers,
        stuct in6_addr * careoff_address_contactnode,
        stuct in6_addr * homeagent_address              );
```

The first two arguments are the same as for the `mip6_addr_passive` function. The third address will be revealed to a client directly after traffic is received. Although data can be tunneled until the client is distributed to some replica, the current implementation optimizes the communication path immediately to reduce load on the home agent. The home agent itself is specified using the fourth argument. Upon return, the calling node will be the current contact node.

Traffic Handoff

A second set of functions provides means to switch traffic sent by the client from the donor to the acceptor.

Because the binding updates sent to the client contain a sequence number, the donor needs to request the sequence number of the most recent BU sent from its local MIPv6 daemon:

```
mip6_handoff_mark(
       stuct in6_addr * client_address,
       stuct in6_addr * contact_address
       stuct in6_addr * careoff_address_handovers,
       stuct in6_addr * acceptor_address       );
```

The first argument is the address of the client that will be handed over. The second argument specifies the contact address to which the client is connected, and the third argument specifies the local care-off address as it is currently known by the client. The fourth argument specifies the address of the acceptor. The returned value is the sequence number requested.

The sequence number has to be included in the request sent to the acceptor, so the latter can send a valid binding update message to the client to switch traffic using:

```
mip6_handoff_start(
       stuct in6addr * client_address,
       stuct in6addr * contact_address,
       stuct in6addr * careoff_address_handovers,
       stuct in6addr * donor_address,
       int bindingupdate_seqnr          );
```

The first three arguments are the same as the mark-function, but the fourth function specifies the replica that initiated the handover. The fifth argument is the value of the next binding update sequence number.

The `mip6_handoff_start` function will execute both the return routability and route optimization procedures. The HoT message is routed to the current contact node's address as specified using the `mip6_addr_passive` function in case the calling node is not the current contact node. After the receipt of the BA the function returns and the calling replica will be responsible for the client involved. The donor then has to execute:

```
mip6_handoff_finish(
       stuct in6addr * client_address,
       stuct in6addr * contact_address,
       stuct in6addr * careoff_address_handovers,
       stuct in6addr * acceptor_address   );
```

to remove data structures not necessary anymore because the client is handed over.

Instead of executing both the return routability and binding update together, the latter can also be delayed. This allows one to execute the return routability in advance, while the donor is still in control of the client, allowing the donor to empty the transport layer's send-buffers. The acceptor then first executes a function that executes the return routability procedure, after which a second function is executed for sending the actual binding update to the client. This optimization reduces the total handoff time because no route optimization procedure needs to be executed once the donor has stopped serving the client, allowing the acceptor to take over the client faster. Experiments show that the total handoff time can be reduced by roughly one third [9]. However, it requires extensive synchronization between the donor and acceptor, making it harder to use.

Since the binding at the client side is on a per-address basis, closing the connection will not affect the client's binding cache. In case the client reconnects to the contact handle, it will be served by the replica that served the client most recently, provided that the binding has not expired. For some applications, this behavior can be undesired. Therefore, the implementation offers a function to reset the binding, which should be used after the connection has been closed.

```
mip6_handoff_clear(struct in6_addr * client_address, struct in6_addr
contact_address);
```

Upon return, the client's binding will be deleted so subsequent traffic will be delivered to the contact node.

3.3 Transport and Application Layer Handoff

As seen from the perspective from Versatile Anycast, handing off a client from one replica to another requires that future traffic sent by the client is delivered to the acceptor instead of the donor. But for most applications, simply switching the traffic is not enough. The accepting replica needs to be able to continue communication with the client, without the latter noticing it is handed off, enabling handoff transparency.

Therefore, a distributed server based on Versatile Anycast has to make sure that all layers above the client's Internet layer responsible for MIPv6 route optimization retain the illusion of communicating with a single entity. So besides switching IP traffic, a transparent handoff within a distributed server also involves executing a transport and application layer handoff. Both are explained in the next two sections.

3.3.1 Transport Layer Handoff

The transport layer is located on top of the Internet layer, and is responsible for multiplexing communication over a single IP address; so multiple connections can use the same destination IP address. Traffic is sent to a socket, which is a combination of an IP address and a port number, using a transport protocol. UDP and TCP are the two most widely used protocols on the Internet.

UDP effectively only adds port numbers to IP packets, offering the same delivery guarantees and as a result datagrams may get dropped, arrive out of order or get duplicated. It is up to the application to detect these events, and deal with them accordingly, making UDP a stateless protocol. This property makes UDP ideal for writing applications that are able to serve a large number of clients sending small requests. An UDP-based server treats each datagram as a single request, processes it, sends back the reply and forgets about the client. If a client does not receive its reply within a certain timeframe, it will deem either the request or reply lost and retransmits.

UDP-based applications usually create only a single UDP socket to receive data from many clients. Reading a datagram from such a socket can return data sent by any client. This behavior is ideal for Versatile Anycast, as switching traffic is enough to retain client handoff transparency. If a replica receives a request from a client it wants to handoff, it only has to request the target replica to execute the route optimization protocol. The client's Internet layer will switch traffic to the accepting replica, while the client application will eventually retransmit the request. This retransmitted request will now be delivered to the acceptor, and processed as normal.

However, merely switching traffic is only applicable to applications in which the reply consists of a single datagram, like for DNS. Other applications using UDP include Voice over IP and Video on Demand. These applications differ in that their replies cannot be contained within a single datagram. Their reply is sent using multiple UDP packets, possibly thousands in case of a high definition video stream. Depending on the moment of the handoff, only executing the route optimization protocol might not be enough to ensure client handoff transparency. If the donor replica already started answering the client's request (i.e. sending the video or audio) the acceptor should continue the reply without the client noticing. This requires an application level handoff, in addition to switching traffic, which is discussed in section 3.3.2.

While UDP only offers an unreliable datagram service for communication between sockets, the TCP protocol offers reliable, in-order delivery of a stream of bytes over a connection setup in advance. A TCP-based server listens to a special listen-socket to which clients connect, yielding a new socket that is used to communicate with the accepted client exclusively. Unlike UDP, TCP is a stateful protocol. It requires that each data packet sent from one side to another carries a sequence number to identify its content. Upon receipt, the receiver has to acknowledge the sender of the data so the latter knows that no retransmit is necessary and it can continue sending data. In case no acknowledgement is received within a certain timeframe the sender's transport layer must retransmit the data. This connection state is part of the socket, and therefore, merely switching the traffic will break the connection, since the acceptor does not have knowledge of the donor's TCP endpoint state in order to continue seamlessly.

For that reason, the TCP socket state must be transferred to the acceptor. This is commonly referred to as TCP handoff. The main goal of a TCP handoff is to ensure that the client's transport layer does not notice that the remote endpoint of the connection has moved. Consequently, applications making use of Versatile Anycast and TCP are required to move the endpoint prior to sending the binding update.

A TCP handoff does not change the connection endpoint, only its location. Within a Versatile Anycast based distributed server, clients are always connected using the distributed server's contact address and application port. Therefore, the same address and port number must be available to the accepting replica. This means that one cannot use different ports on each replica, but use one that is available on each replica instead.

Versatile Anycast guarantees that an acceptor is always able to accept a TCP endpoint by ensuring that a client can only be connected to a single replica at any time. TCP uses both the originating and destination IP addresses and port numbers to identify traffic. Although the distributed server's IP address and port number are fixed for each replica, traffic can always be identified using the client's IP address and port number. At the beginning of a handover, the client's binding for the contact address will always be set to the donoring replica. Therefore, it is impossible that the acceptor already holds an active socket for the same client, as it does not receive traffic from the client for the server's contact address. Hence, the socket can always be accepted and added to the transport layer.

The latter can only be the case if the distributed server removes sockets associated with clients that were handed over. Otherwise, one can run into the situation in which a TCP socket must be rejected because the client involved was served earlier. If we were able to insert the socket, the transport layer would have no means to identify traffic for the two sockets, as both use the same address-port pairs.

TCPCP

For Linux, a proof of concept implementation named TCPCP exists [1, 13], that allows one to extract a TCP socket from the transport layer from one node, and insert it into another. Because the transport layer is located inside the kernel, it consists of a kernel patch in addition to a user level library for extracting and modifying the socket involved.

The first step before switching the traffic is to make sure the TCP state is frozen, so its state cannot change anymore. This means that no packets are sent, while incoming packets for the socket are still recognized but silently dropped by the transport layer. After the socket is frozen, it must be serialized and sent to the destination machine. The latter then has to insert the frozen socket into its kernel. After inserting the endpoint, Versatile Anycast can be used to switch traffic from the donor to acceptor. After traffic is switched, the acceptor has to unfreeze the inserted socket so it can be used to receive and sent data as normal, while the donor can safely close the frozen socket.

The reason that the sockets are frozen is that during the handoff packets may arrive at either the donor or acceptor. The TCP protocol requires that if one receives a TCP packet proving that the sender is not in a legal state, it must reset the connection. A packet received by the transport layer for an unknown connection is therefore followed by a reset packet, sent its originator. During the handoff the socket at the client remains active, and therefore it is legitimate for the client to send TCP packets. It is important that these packets are still recognized but silently discarded, to prevent reset messages to be sent.

Additionally, the acceptor should only start sending data to the client after the binding update. Otherwise, the client might send a reset message to the acceptor, as it cannot identify traffic correctly yet. Therefore, the socket at the acceptor must be frozen until traffic is switched. After the client is handed over completely, the acceptor has to notify the donor so the latter can close the frozen socket and forget about the client. Packets sent by the donor but received after the binding update is processed will be discarded because the donor's address does not match the binding cache.

It is important to understand that while packets are dropped, the connection is not broken. The TCP specification requires the client must determine the retransmission timeout dynamically, depending on the characteristics of the connection. Most TCP implementations base the timeout on the average round trip time between sender and receiver, while the number of retries must be sufficient to overcome temporal network problems causing traffic to be temporally delayed or dropped. In addition to the TCP specification, many client applications implement a maximum timeout, so the user does not have to wait endlessly. Therefore, the handoff must be completed before either transport layer or the application gives up. Although these timeouts are highly variable, we expect that they are more then sufficient to complete a handoff.

3.3.2 Application Layer Handoff

For distributed servers that are stateless and UDP based, handoffs only require that traffic is switched. Therefore, a donor only has to request an acceptor to execute the route optimization protocol, making the usage of Versatile Anycast's handover functionality straightforward. But if a handoff is executed in mid-connection, additional measurements must be taken to retain the client's illusion that it is communicating with a single entity. Consider the Video on Demand (VoD) server example. The client requests a certain video, after which the server will stream the video to the client, using either TCP or UDP.

In case UDP is used to for communication, the donor's application not only informs the acceptor of the handoff, but also about the state of the client. This state includes information about the client's endpoint, identified by its IP address and the port number used and what the client expects next. This information is then used by the acceptor to change the binding at the client, and continue communication with the client. For a simple VoD server the information about the request will usually be limited to an identifier of the requested stream and a byte-offset to indicate the data the client expects next.

While for UDP the client's request is often contained within a single datagram, TCP allows for requests with arbitrary lengths. Additionally, data can only be read once since the client typically does not resent the request. Some services even allow multiple requests during a single connection. Therefore, a TCP based distributed server has to communicate the exact state of the communication session. An example is a database server offering transaction support. The client starts a transaction, and during the transaction a number of data update and retrieval queries are made. Update queries can potentially change the database, and change the outcome of queries issued next. Handing off a client in the middle of an transaction therefore not only involves informing the acceptor of the query issued last, but also the current state of the database, and possibly other administrative information.

Because the information required by the acceptor differs per application, it is up to the application to ensure the acceptor can continue communication without the client noticing.

3.4 Discussion

In this section we will discuss why Versatile Anycast is an ideal mechanism for transparent distribution of clients. We will briefly discuss implementation and deployment issues before presenting our framework in chapter 4.

3.4.1 Versatile Anycast

Versatile Anycast allows one to build a distributed server that requires no support from the client application. A client application will not be aware of multiple replicas, and from its perspective the distributed server acts as a single machine. Instead of modifying the client application itself, Versatile Anycast depends on functionality on the client side that is likely to become standard in the near future. Nowadays, Internet nodes like laptops or PDA's are highly portable, and offer the ability to seamlessly connect to different networks using wireless technologies like WiFi and WiMaX. Joining a different network requires the adoption of a different Internet address, breaking existing connections and making it more difficult to contact one that is away from its home network. MIPv6 solves these problems, and therefore modern operating systems already offer support for this technology. Versatile Anycast only requires modifications to the MIPv6 implementation running on the replica nodes.

By putting the Versatile Anycast functionality inside a daemon that interfaces with a standard kernel module, deployment of replicas will be reasonable straightforward. Therefore, every node with a network connection is qualified to act as a replica, since no changes to the network are necessary. The service provider has to setup a single home agent on the network, to process traffic sent to the server's contact address. Traffic can then be forwarded to any node on the Internet. Because of this, replicas can be placed anywhere on the Internet enabling placement close to clients.

Versatile Anycast itself can support a large replica group, with both the donor and acceptor being any arbitrary node from this group. However, each handoff requires two messages to be routed over the home agent and contact node. The latter is required to add a temporal mapping for each received HoT message in order to forward the HoTI message back to the correct acceptor. Although this can potentially become a bottleneck in the system, we believe the number of clients the distributed server can sustain will be sufficient because those messages are the size of an IPv6 packet with an option header, not carrying any payload.

Versatile Anycast is highly resilient to replica group change. It offers means to guarantee availability of the contact address by allowing the user to control the binding of this address to a selected node in the system. This means that the contact address is not bound to a predefined node, but can be moved between nodes. Moreover, Versatile Anycast does not require that those nodes reside on the same network as the home agent, because the latter can act as a representative. Any replica, given knowledge over the security association, can take over the contact handle. Therefore, Versatile Anycast offers means to guarantee availability of the contact handle even in the presence of failures of both replicas or the network [11].

Instead of requiring a predefined mapping, Versatile Anycast allows the distributed server to handoff clients individually, resulting in a very fine-grain control over the distribution. For that reason, it can support both stateful and stateless distribution policies. Versatile Anycast delays the binding of a client to a replica until the actual moment of the handoff. This is in contrast with other distribution mechanisms that require the client-to-replica mappings to be defined in advance, such as true network anycasting explained in section 2.2.4.

While traditional techniques are limited to distributing clients at the beginning of a communication session, Versatile Anycast also allows one to handoff a client at any instance of time. This makes it easier to achieve fair load balancing in a system where communication sessions are relatively long-lived, such as Video on Demand. In addition, it enables a broad range of new applications not possible before. One application is handing off a client immediately after receipt of its request. This allows for content-based redirection, partitioning the requests over a number of replicas. Replicas can also be made responsible for clients during an interval of the communication connection. If a replica has satisfied its responsibility, it can handoff the client to the next replica responsible for further communication. An example is the download service in section 3.2.2, where files are stripped over multiple replicas. No replica has the complete file, requiring them to handoff clients to each other during client's download.

Handoffs also allow a replica to leave the network gracefully. A replica entering an error state can decide to first handoff its clients and leave the network, so its clients will not be affected. This feature can help making distributed systems more reliable.

Because the return routability can be executed before sending the binding update, a handoff can be executed as fast as the round trip time between the acceptor and client. However, attention should be given to data sent by the client to the donor, just before it switches traffic to the acceptor. In case UDP is used, the donor receiving the datagram must not start the handover procedure again, but instead discard the datagram. Otherwise, it tries to handover a client that is not under its control anymore. In case TCP is used, the donor's transport layer will automatically drop this data as it is either frozen or already removed. Normal TCP behavior at the client guarantees data will eventually be resent, in which case the IPv6 layer takes care that the traffic is sent to the acceptor. This will cause only a slight disruption in the flow of traffic.

Versatile Anycast achieves the distribution of clients without relying on a single proxying frond-end to forward traffic, reducing the network load and increasing availability. However, one may argue that the home agent still functions as a potential single point of failure. Although this is true we would like to point out that the home agent is likely to be running on a router also responsible for routing traffic to and from the Internet. Because one wants to guard against complete networks being cut-off by from the Internet, those routers are often implemented redundantly with a backup router capable of taking over the primary router in case of a failure. In addition, those routers often have multiple links to make sure Internet connectivity is maintained at any time. In order to protect against power failures, alternative power sources like generators are often on standby. Powering such a single router in case of a power outage is less complicated than making sure the needs of a complete data center hosting a complete server cluster are met in terms of power and the growing problem of heat dissipation. With such a protected, highly-available router acting as the home agent, the replicas can be spread throughout the Internet, limiting the risk that all replicas are unavailable due to adversity while reducing communication latencies to clients.

3.4.1 Security

Upon joining a foreign network, a mobile node needs to update the home agent of its current care-off address by sending a binding update. Without the use of additional security measurements, the home agent would have no means to authenticate the binding update message. Consequently, an unauthorized party would be able to change the binding at the home agent, hijacking traffic sent to the home address.

MIPv6 therefore uses IPsec to protect the signalling between the mobile node and the home agent. Besides protecting against unauthorized binding updates, the HoT and HoTI messages forwarded by the home are protected as well. This prevents against hijacking a communication session between a mobile node and a correspondent node by illegitimate return routability procedures.

In order to utilize IPsec, the home agent and mobile node need to agree on a security association. This security association includes information such as the algorithm used for encryption, keys and other parameters, and can be setup manually or using IKE. For normal MIPv6, restricting knowledge on this security association to the mobile node and its home agent is vital. But within Versatile Anycast, knowledge over this key is shared amongst backup contact nodes, so each of them can take over the contact address. Because compromise of the security association endangers the distributed server's security and availability, it is strongly suggested to restrain the number of backup contact nodes and ensure their security. For typical distributed servers, only a few such nodes are required to warrant high availability of the contact handle. One of them will be the contact node, and might be located near the home agent to optimize handoffs, by reducing the communication delay between the home agent and the contact node. The remaining backup contact nodes should be located on distinct networks. By doing so, adversary like power outage or network downtime should not affect all backup contact nodes, providing higher availability of the contact handle.

However, as the MIPv6 daemon on which the current implementation of Versatile Anycast is based is not mature yet, it does not yet use IPsec to secure communication. As the implementation matures, the modified Versatile Anycast should be changed to also include this functionality. Most likely, it will require that the security association is created manually beforehand, and all backup contact nodes are configured with this information before the Versatile Anycast daemon is started.

Although the static configuration of the security association provides reasonable security against hostile address takeovers, real-system deployments might require a higher level of security. A compromised key enables third parties to take over the contact address or clients connected to the distributed server, making the system insecure and unreliable. Such situations require one to take down the system and distribute a new security association. To guard against these situations, dynamic keying can be deployed so the security association can be updated regularly without restarting the system. The problem with this approach is that all backup contact nodes should be given knowledge over the security association each time it changes, requiring a secure distribution mechanism for distributing the newly generated keys. This problem has not gained much attention yet, and requires further investigation.

3.4.2 Distributed Servers

Despite these advantages, the current Versatile Anycast implementation does not allow one to build fully transparent distributed servers. The low-level interface only offers functionality to control the switching of traffic. Applications using this interface are required to orchestrate the handoffs themselves, requiring all replicas to listen for incoming handovers and process them accordingly. As explained in section 3.3, for many applications this also includes executing transport level handovers, requiring one to fully understand the implications as incorrect use will lead to broken connections or sockets that cannot be inserted at the acceptor.

Additionally, all nodes must also listen for new connections initiated by clients currently bound to that node, and hand them over if necessary. One of these nodes will be the contact node, and an application programmer should take great care not to overload this single entity. High availability of the distributed server can only be accomplished by carefully monitoring this node's health, requiring other nodes to take over the contact handle if necessary.

Distributing clients involves knowledge over the server's composition, to select an appropriate target for each handover. Distributed servers typically consist of many replicas. Any replica may fail, while new replicas may be added to the system at any point of time. Therefore, we need membership mechanisms to announce new replicas and to remove ones that left the system.

These requirements make it quite complicated to use Versatile Anycast directly. The main contribution of this thesis is to propose a versatile framework that allows one to build fully transparent distributed systems based on Versatile Anycast, without introducing additional complexity over traditional single-node servers. This framework will take care of executing handoffs, maintaining the contact handle and handling membership. This will allow system programmers to concentrate on the application, rather than on its deployment.

Because of the fundamental difference between the stream-oriented TCP protocol and the best-effort UDP datagram protocol, the framework presented in this thesis only supports TCP-based applications. While this decision covers the majority of webservices, we acknowledge the need for UDP and provide directions at the end of this thesis to add support for UDP to the framework as well.

Chapter 4

A Versatile Framework for Distributed Servers

In Chapter 3 we showed that Versatile Anycast meets all the requirements of the ideal distribution mechanism that were stated in Chapter 2. It allows one to build a distributed server that can transparently distribute many clients over many replicas. Instead of distributing clients at the beginning of the client-server interaction, a client can be handed off to a chosen replica at any moment of time, enabling a broad range of new applications.

Versatile Anycast only provides means to switch traffic from a donor to an acceptor. But merely switching the traffic is not enough to build a distributed server. Handovers need to be carefully orchestrated, so the client's illusion that it is communicating with a single entity is retained. Additionally, the distributed server also has to administer the contact address. This address must always be bound to a contact node responsible for accepting clients and forwarding return routability control messages. Failure of the contact node will render the distributed server unavailable; therefore most distributed servers must use Versatile Anycast's features to control the binding of the contact address. Furthermore, the distributed server programmer has to focus on distributed-server related technology, like membership management and information distribution.

This chapter presents a framework that hides this complexity from the application programmer, so the latter can focus on the implementation of the server's logic instead on managing the distributed server. Our goal is to propose a framework that allows one to build a distributed server without requiring knowledge about distributed systems by offering the same methodology as a traditional single-node server. Besides supporting easy development of distributed servers, the framework must allow one to take full advantage of Versatile Anycast. In addition, the framework must be versatile in that it can be used to develop a broad range of applications.

This chapter first discusses the features of the framework. We then present the application level interface of the framework that allows one to build distributed servers. Finally, we present a proxy application build using our framework to support existing software.

4.1 Features

Instead of requiring each distributed server to implement the same functionality in order to use Versatile Anycast, we must identify the common parts that can be abstracted. This

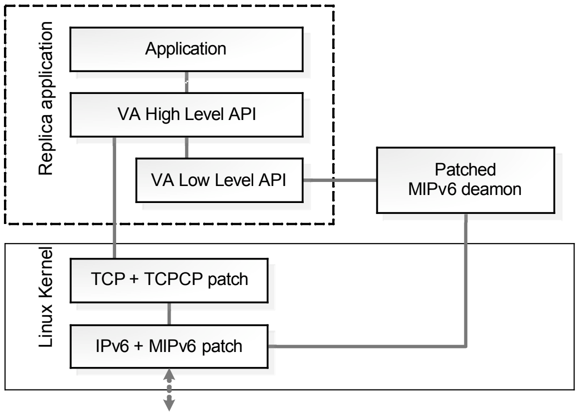

Fig. 7 Location of the framework and interaction with components

functionality must then be supported by the framework, allowing for faster and easier development of distributed servers. The framework must hide the intricacies of Versatile Anycast, so the application programmer can focus on the application itself. Additionally, it is better to confine the implementation using a single package that can be reused for multiple projects, so one does not have to implement it by itself, reducing the risk of programming errors due to insufficient knowledge about Versatile Anycast.

The actual distributed server application is then written on top of this framework, as shown in Fig. 7. The framework will control the modified MIPv6 daemon using the low level Versatile Anycast API discussed in section 3.2. Additionally, it will use a patched TCP layer to enable transport layer handovers. This section first discusses the features the framework must support before explaining the API available to application programmers.

4.1.1 Membership

A distributed server typically consists of multiple replicas. With no support from the client application or the network, the distributed server itself will be responsible for distributing clients amongst the available replicas. This means that at least one node within the system must have knowledge over (some) other nodes. For example, within a local server cluster, the proxying frond-end must have knowledge over the available servers, in order to forward traffic.

The same holds for distributed servers using Versatile Anycast. The contact node needs to know about replicas in the system, so it can handoff clients to them, instead of processing them all by itself. But knowledge over other replicas should not be confined to this single contact node. In order to build a reliable distributed server, a few backup contact nodes should be able to take over the contact address quickly at any time. Therefore, the backup contact nodes should hold the same information about the replica group as the contact node. Moreover, depending on the application and distribution policy, it can even be the case that every replica in the system has to be able to perform handoffs. Knowledge over the replica group composition may therefore not be limited to a small set of nodes, but involve all replicas.

Each replica that desires to receive clients must join the replica group. The easiest solution is to create a list of the replica group members before the system is started. However, nodes within distributed servers are potentially unreliable in that they may become unavailable any time, outdating this list. One important advantage of distributed servers is that one can add more replicas to the system to increase its capacity. Other nodes should learn about this replica, preferably without restarting the server itself.

Therefore, we need a dynamic membership protocol to manage the replica group composition and share this knowledge among the replicas. In case a replica is started, it should announce itself to become a member of the group. Consequently, if the replica decides to leave the system gracefully, it must use the membership protocol to announce its departure so the group view can be updated accordingly. In addition, the membership protocol must have means to detect replica failure, so replicas that left the distributed server ungracefully will eventually be detected and removed from the group.

Currently, our framework does not yet use a dynamic membership protocol to update the replica group continuously. For the sake of simplicity, we currently assume all nodes within the system are reliable. Before a replica is started, it is configured with a list of known replicas, which is then used to determine a handover target. However, we do support failure detection of a replica, so an erroneous replica can be removed from this list so subsequent consultation of the distribution policy does not return the same target. Information about failed replicas is not yet disseminated to other replicas, so each replica has to detect a failed replica individually.

Requirements

Although our framework currently does not support a dynamic view of the replica group, we anticipated its implementation in future versions by defining a base class consisting of methods to allow interaction between the framework and the membership protocol. This also allows one to change the membership protocol in case the default one does not meet the requirements of the distributed server application.

However, to allow easy development using the framework, the default protocol should already be able to support a wide range of applications. Some membership protocols depend on a central component that keeps track of the current state of the replica group composition. Others are distributed in that the membership protocol does not depend on one specific node in the system but instead relies on the replica group itself or a subset thereof. As our framework allows one to implement highly available distributed servers, it is important that the framework itself does not depend on a single entity other than the home agent. Since the home agent can be a standard router running a normal MIPv6 home agent implementation, the framework must use a distributed membership protocol that does not require a central component.

Often, a distributed membership protocols forms an overlay network to disseminate information about events, such as the addition or removal of a node. The overlay network must allow fast distribution of membership information and be resilient to high node churn, a property required by our framework. Additionally, we demand that the membership protocol is inexpensive in terms of complexity and system resources such as processing power, memory and bandwidth-usage. Finally, it is important that one can participate in the membership protocol, without being a member of the replica group itself. For example, the contact node must be able to maintain a current replica group view without being a replica itself.

One possible implementation could be based on Cyclon [12], a complete framework for inexpensive membership management. Cyclon forms an unstructured overlay network with characteristics of random graphs, making it robust and resilient to massive node failures. It then uses gossiping to disseminate replica group information, in such a way that it is able to cope with high replica churn, which results in frequent membership updates.

Cyclon requires that each replica that wants to join the membership protocol contacts one member already executing the protocol. This node then offers its knowledge over the replica group composition, so the replica will learn about more replicas within the system. The joining replica then uses this information to announce its presence to other nodes, which then share their knowledge about other replicas. By continuously announcing its presence and gathering information about other peers, each replica within the distributed server will learn about peer replicas. This information can then be requested by the application, or in our case the framework.

The framework must support both small-scale distributed servers consisting of only a few replicas, and distributed servers consisting of thousands of replicas. For such large-scale distributed servers, maintaining a complete view of the replica group can be very expensive. First, each replica has to store information about a large number of replicas, which is likely to be memory-intensive. Second, within large-scale distributed servers, replica churn is likely to be very high, requiring expensive information dissemination strategies. Therefore, Cyclon maintains a partial membership view at each member. The number of peers known by each replica is restricted by a configurable parameter. Applications requiring knowledge over many peer replicas can set this parameter to a high number, at the expense of bandwidth costs or freshness of the local membership list. For small-scale distributed servers, the number of available replicas can be equal or lower then the value of this parameter, allowing replica's to maintain a complete view. This makes Cyclon very flexible with respect to the distributed server's scale, and hence a good candidate membership protocol to use in the framework.

4.1.2 Multiple Distribution Policies

As each application has specific needs, the framework must support different kinds of policies. Despite the wide variety of applications, it is likely that multiple applications require the same type of policy. In order to ease the development of distributed servers, the framework can be accompanied by a set of ready-made policies to choose from. The programmer then only has to select such a predefined policy and request the framework to use it for distribution.

A policy can be stateless or stateful, depending on whether or not it exploits knowledge about the state of one or more replicas to select a target. By basing the decision for a target on current information, one can build more advanced policies. One example is using resource state to achieve fair load balancing among the members of the replica group. In order to support such a distribution policy, a replica must disseminate its state to other replicas and keep track of the state of peer replicas. One may use the framework's future membership protocol to exchange this information, or use it as a basis to build a secondary overlay network. In addition, advanced distribution policies may require cooperation from the server application. For example, the number of connected clients at each replica can be used to orchestrate the distribution of clients.

Although the set of pre-defined policies can support a large number of applications, some applications might have needs not covered by any of those. Therefore, an application programmer must be able to define a new policy, in order to custom-tailor the distribution of clients. An example of such a type of distribution is content-aware distribution, in which the client's request is used to select an appropriate target replica. As the format of the request and partitioning over replicas differs per application, these types of policies are difficult to standardize into a versatile predefined policy.

Even though the framework imposes almost no restrictions on the distribution policy, one must be aware about the inner workings of Versatile Anycast. It is important not to overload the contact node, by requiring it to perform expensive computations while selecting a target for each connecting client, as it will severely restrict the performance of the distributed server. Policies requiring expensive computation should implement a distribution scheme in which the contact node only executes a minimal initial distribution, for example redirect the client to a random replica. This replica is then required to compute a final target. This allows one to disperse computational costs among multiple nodes instead of concentrating these on the contact node.

An alternative to programming a custom content-aware distribution policy is to use a standard pre-defined policy and have the target replica's server *application* select a final target. By doing so, the contact node first hands off the client to a replica selected using a simple pre-defined policy. The target's server application then accepts the client, reads the request and manually hands off the client to a computed target. So, instead of relying on a distribution policy, the application logic is made responsible for attaching clients to the correct replica, by explicitly handing over a client to a target. We will support both methods in our framework to allow greater flexibility.

4.1.3 Contact Node

One node in the network must be responsible for processing traffic sent by clients not yet bounded to a replica. The framework must manage this contact node carefully, to guard against situations in which no new clients can be accepted or handed over. The next two sections will discuss how the contact address is managed, and discuss the primary task of the contact node.

Contact Address Management

The current implementation of Versatile Anycast only offers means to attach the contact address to a single fixed replica. But to build highly available servers, one server should monitor the health of the contact node, and change the binding in case the current contact node experiences problems. In order to do so, the distributed server should keep track of backup contact nodes, and decide which one should take over the contact address if necessary. Like the membership protocol, the framework is designed in such a way that it can easily be extended to support dynamic management of the contact address. Interaction between the framework and address management protocol is defined using a base-class, allowing one to implement or change the default protocol if necessary.

The primary task of the address management protocol is to ensure that traffic sent to the contact address is delivered to a healthy contact node, so the sender can be distributed to a replica. In order to do this, the framework must be configured with a list of backup nodes. Additionally, these backup nodes must be given knowledge about the MIPv6 security association, so they can impersonate the creator of the contact address to the home agent.

The protocol should then use the backup nodes to monitor the health of the contact node. Therefore, in contrast to the membership protocol, the address management protocol will only involve a small number of nodes, for security reasons discussed in section 3.4.1. Additionally, most service providers may want to control which node may take over the contact address, in order to optimize the quality of service. Besides accepting connecting clients, the contact node also has the obligation to forward HoT and HoTI messages sent by replicas during handoff procedures. Therefore, optimizing the network path between the home agent and contact node will reduce the total time required for a handover, as return routability procedures can be executed faster, while reducing the risk of dropped packets. Therefore, the framework allows ordering of the backup contact nodes. The protocol then should use this information in such a way that the backup node with the highest preference is the current contact node, provided that it is fit to carry out this task.

The address management protocol can be fully decentralized, allowing dynamic addition and removal of backup contact nodes. But we believe the next-discussed minimal protocol based on a static list of backup nodes will be sufficient.

This simple protocol assumes all n backup nodes are known during initialization of the distributed server, after which the it assigns each a rank to each of them according to their preference; 0 for the most preferred one to n-1 for the least preferred one.

Each backup node n is then given knowledge over this ordered list, and monitors its predecessor, n-1. In case this node is not available, it will monitor n-2, and so on. In case no live predecessor is found, the backup contact node must assume that all of them are unavailable, and therefore it should take over the contact address. If a backup node becomes available again, it should announce its status to all its successors. In case one of them is the current contact node, it should take over this role so at any time the backup contact node with the highest preference is the contact node.

Accepting Replicas

Unlike the relaying of HoT and HoTI messages, which is automatically done by the modified MIPv6 daemon, the contact node replica must explicitly accept connecting clients and hand them over accordingly.

One can think of separating the contact node and server application functionalities into two distinct programs. However, both must use the same local port to either accept or serve clients. Therefore, running both programs on a single machine would not be possible. A second strategy is to use a single executable for both contact-node and application logic. Because of this single executable, each node runs the same code and is potentially able to process clients or become the contact node (requiring it is configured with the security association).

We decided to use the single executable strategy, by making the framework responsible for differentiation. This makes deployment of the distributed server easier because only a single application needs to be distributed. Secondly, resources can be fully utilized, because a backup node on standby can be used as a normal replica. Even the contact node can process clients normally and only handoff clients in case its local resources are not sufficient. This allows one to build server applications that scale from a single node to thousands of nodes, without the need of rewriting software.

It is important to note that only the traffic originating from clients not yet bound to a replica will reach the contact node. If a client already holds a valid MIPv6 binding, any subsequent connection attempt will not reach the contact node, but the replica that produced the MIPv6 binding.

The MIPv6 specification requires that the binding update not only contains the care-off address address to use for route optimization, but also a binding lifetime. The client will use this binding lifetime to expire bindings and keep its binding cache fresh. A replica (acting as a mobile node) also removes old bindings from its own local binding cache. To this end, it may also use expiration timers, but reset this timer each time traffic is sent to the corresponding correspondent node.

In case a binding in the client's cache is about to expire, the client must send a binding update to the replica, requesting it to refresh the binding. If the replica has not removed the corresponding binding from its own binding cache, it will respond with a new binding update. Therefore, even for clients that disconnected from the replica, subsequent new connection attempts might not reach the contact node but the replica instead.

For this reason it is necessary that each node in the system listens for new connections and accepts connecting clients. One example of an application that is likely to encounter this behavior is an HTTP webserver. A web browser first requests an HTML document. If the machine running the browser has no MIPv6 binding yet, this request will reach the contact node, which may distribute the client to a replica located near the client to reduce latency. But after fetching the HTML document, the browser can make a large number of subsequent requests in order to retrieve the embedded objects, such as images or stylesheets. These requests will be sent directly to the replica, using the route optimization path.

This behavior is therefore helpful to reduce the load on the contact node, because the contact node only has to distribute the client once. For some applications however, this behavior is undesired, as they may require that the contact node processes each request. For example, because the contact node may be the only one capable of determining the optimal target. The framework can therefore be configured to clear the binding at the client upon closing the connection.

It is essential to understand that handing off a client from one node to another involves switching *all traffic* between two nodes, not only traffic belonging to a specific connection. In case a client has multiple connections to a single contact address, handing off one of these connections will affect the remaining connections. Consider the webserver example. If the client is downloading object 'A' and 'B' simultaneously, handing over the client to another replica to download object A will disrupt the downloading of object B. This behavior is inherent to Versatile Anycast, and such situations must be avoided by the application programmer.

If one requires multiple connections to the distributed server, one solution would be to use multiple contact addresses. The client can then use a distinct address for every simultaneous connection, allowing the distributed server to switch traffic for each connection individually. The framework must then multiplex these addresses so the application is not concerned with listening to all of them. However, this solution is not versatile. It is only applicable to selected applications in which either the client can be adapted or offers support for specifying distinct addresses for different connections. An example of the latter is a webbrowser. Each embedded object in the HTML file can use a different IPv6 address so handing off one download does not affect the downloading of the other ones.

For other applications, handing over a client with multiple connections requires that all other sockets for the client are transferred as well. For this to work correctly, the framework needs to keep track of sockets belonging to the same client. If one of these sockets is handed off, the framework should notify the application of the other sockets that will be affected by the operation. This makes programming the distributed server more difficult, as the programmer needs to understand the full consequences of a handoff, and anticipate on the many different scenarios. As a result, the framework offers no support for this kind of functionality. The framework assumes that each client has only a single connection to the distributed server at any instance of time.

The use of multiple contact addresses can also be used to reduce the load on the single contact node. One can use the DNS redirection mechanism discussed in section 2.2.5 to distribute clients over multiple contact nodes. Instead of resolving to a single fixed contact address, the authoritative DNS server for the distributed server's hostname selects one of the available contact addresses. Each contact address can be bound to a different contact node, increasing the system's client distribution capacities. However, this functionality is not available in the current framework.

4.1.4 Performing Handoffs

Handoffs can be executed for two reasons: the first is to distribute the load among replicas by offloading the donor and increasing the load on the acceptor. The second reason is enabling applications that were not possible before. Most traditional distribution mechanisms as discussed in section 2.2 only allow one to attach a client to a replica once during the time the client connects to the distributed server. Versatile Anycast allows one to do a handoff at any arbitrary moment, for example after receiving the client's request. One example of an application taking advantage of this principle is a distributed streaming server that offers multiple streams from different viewpoints, for example of an ice hockey game. If the client requests a viewpoint currently not available on the serving replica, it can be handed off to a replica that does provide the stream for the selected viewpoint.

Handing off a client from the donor to the acceptor should be fast. The donor must inform the acceptor of the client, after which the acceptor will execute the return routability and perform a binding update to switch traffic. Until the moment of the binding update, the donor can continue sending data. But immediately after the binding update, data received from the donor will be ignored, and the acceptor should continue correctly without the client noticing. The use of TCP makes the handoff more complex, because unexpected data packets may cause the TCP connection to be reset, as discussed in section 3.3.1.

As the exact TCP endpoint state must be available to the acceptor, it is contained in the handover request. Because the size of the serialized TCP stack may exceed the size of a single UDP packet, the framework must use TCP for communication between the donor and acceptor. The donor connects to the acceptor, and upon connection it sends all data required by the acceptor to execute the complete handoff. This information includes the address of the client, the MIPv6 binding update sequence number to be used and the serialized TCP stack. The acceptor then inserts the TCP stack into its kernel and requests the client to switch traffic using the low-level Versatile Anycast implementation. Upon return, it can unfreeze the socket and start processing the client. Before the handoff TCP connection is closed, it informs the donor of the successful handoff so the latter can remove the frozen socket from its kernel and forget about the client.

Because the endpoint state should not change after it is serialized, the donor has to ignore all data sent by the client. Hence, for a TCP handoff no data can be exchanged between extracting the socket at the donor and switching traffic at the acceptor. Therefore, the procedure must be executed within a reasonable amount of time, as some applications require a constant flow of data and are only resilient to a short interruption of the data flow. Additionally, the framework must deal with replica failure and recover from a target replica being unavailable.

Besides executing the transport layer handoff, the framework should offer means to applications to transfer information about the client to the acceptor, in case the client is handed off after communication has started. This per-user information can be used by the server application to implement the application-layer handoff, as discussed in section 3.3.2, allowing the acceptor to continue communication without the client noticing.

As each application has different needs, there is no unified format that can be used to support all server applications. Therefore, the framework allows one to specify a pointer to a buffer. During a handoff, the framework will transfer this entire buffer to the acceptor. As seen by the framework, the buffer functions as a black box, and the distributed server application must make sure it contains all the information required by the acceptor.

4.2 Application Programmers Interface

The framework aims at simplifying the use of Versatile Anycast. It allows one to run the same executable on many machines, and depending on the role of the node, the framework will execute specific tasks, so the programmer does not have to worry about the management of the distributed server.

The framework consists of a single library, with an API to control the distributed server and handoff clients. This section will discuss the current API available to programmers. In chapter 6 we will give an example distributed server using this API, and show that the complexity of writing server applications using our framework does not exceed the complexity of traditional server architectures.

4.2.1 Configuration File

Every instance of the distributed server-application requires information about the system in order to function properly. These parameters are contained in a configuration file, to allow easy modification of the distributed server. The server application only needs to specify the path to this file, so parsing of the file is done within the framework itself. As the configuration file contains global information about the distributed server, it should be the same at each replica, i.e. all replicas must use the same configuration. Otherwise, the behavior of the framework is unspecified.

The configuration file contains one directive per line. A directive is followed by an argument, specifying the value for the directive. Most directives can only have a single value, and in case the directive is found multiple times in the configuration file, the last encountered directive having a valid argument will be used by the framework. Some directives can have multiple values, which is the case with lists. In that case the first encountered valid argument will be the head of the list, and subsequent encountered valid arguments will be added to the list's tail. Lines beginning with the dash-symbol will be ignored and can be used for comments.

The following directives are valid:

`DebugLevel`

Integer

Specifies the verbosity of the framework. Output of events will be written to standard out (stdout), while errors will be written to standard error (stderr) The following values are valid:

0 (default)	No messages
1	Only errors
2	Only errors and important events
3	Verbose

See also section 4.2.2

`AnycastAddress`

IPv6 address

The contact address of the distributed server. Mandatory.

`ApplicationPort`

Integer

The port number to which clients connect. This port must be available and the user running the server must have privileges to open the port. Mandatory.

`ApplicationBacklog`

Integer

Defines the maximum length of the queue of pending connections initiated by clients. If a connection request arrives with the queue full, the connection attempt is rejected. By default a backlog size of 5 will be used.

`HomeAgent`

IPv6 address

The address of the home agent acting as distributed server's representative on the home network to which the contact address belongs. Mandatory.

`AddContactNode`

IPv6 address

Specifies a contact node. The first encountered instance will define the first initial contact node, and subsequent directives will set backup contact nodes in the order of preference, with the most preferred backup node on top.

All nodes specified should be configured with the security association.

It is mandatory to specify at least one contact node.

`AddReplicaNode`	*IPv6 address* Specifies a replica node. In the current implementation, this directive must be used to announce every replica within the system, i.e. to specify the complete replica group. In future versions of the framework, only a subset or replicas can be specified, which are then used to bootstrap the membership management protocol, so it can join the overlay properly.
`HandoverPort`	*Integer* The port number on which handover requests from peer replicas can be accepted. This port must be available and the user running the server must have privileges to open the port. Mandatory.
`HandoverConnectTimeout`	*Integer* Maximum time (in milliseconds) to wait for a TCP connection attempt from a donoring replica to an accepting replica to complete. By default, the framework will use one second, after which a second target is selected for a new connection attempt. The value for this parameter depends on the distribution of replicas, and should be based on the maximum round trip time between two replicas.
`ClearBindingOnClose`	*Integer* Specifies whether the binding at finished clients should be reset or retained until the binding expires by itself. By resetting the binding, all connects will be received by the current contact node. By not resetting the binding, subsequent connects will be handled by the replica that processed the client last, lowering the load on the current contact node for applications that make a large number of subsequent connects (i.e. HTTP webserver), as explained in section 4.1.3. This directive must be set to 0 in case a client may open multiple connections at once, as clearing the binding upon close of one connection causes parallel connections to be broken. 0 (default) — Retain binding on close 1 — Clear binding on close

4.2.2 Error Handling

Most of the API methods return a statuscode, so the application can check for errors. Successful execution of a method will return a 0, while an error will return a negative integer. The actual value of the negative error depends on the error encountered. Currently, our framework uses the following error codes:

`GECKO_ERROR_FAILED`	Generic error
`GECKO_ERROR_NOTALLOWED`	Method is currently not allowed
`GECKO_ERROR_BADFD`	The provided argument is not a valid file descriptor
`GECKO_ERROR_NOTIPV6`	The provided argument is not a valid IPv6 address
`GECKO_ERROR_TIMEOUT`	A timeout was encountered while executing the method
`GECKO_ERROR_NOTARGET`	No suitable target replica was found
`GECKO_ERROR_TARGETERR`	The target replica did not respond correctly

Most functions discussed in this chapter will return either a 0 or the generic `GECKO_ERROR_FAILED` error code, unless stated otherwise. Besides returning error codes, one can control the verbosity of the framework. Using the `DebugLevel` directive in the configuration file the framework can be requested print a certain level of information. Currently, the following levels of verbosity are available:

- Level 0 will print no information at all;
- Level 1 only prints information about errors. This includes errors encountered while initializing, but also information about failed handoffs and errors encountered during membership or contact node management;
- Level 2 prints the same information as level 1, but also information about important events. This includes information about accepted clients, outgoing or incoming handoffs and target selections made by the distribution policy;
- Level 3 is the most verbose, and can be used to track activity throughout the framework.

Errors encountered will be written to stderr, while information about regular events will be written to stdout.

4.2.3 GeckoFramework

The distributed server application must first initialize the framework. Every node within the distributed server must create an instance of the GeckoFramework class, regardless of whether it is the contact node, a backup contact node or a normal replica.

This object is used to connect all components of the framework together. It parses the configuration file and maintains the state of the Versatile Anycast network. Furthermore, it is responsible for starting background tasks for membership, contact address management and executing handoffs. For normal usage, a programmer should not be concerned with these tasks directly. Chapter 6 will show how this functionality is implemented, so the remainder of this chapter is only concerned with public interfaces available to the programmer of the application.

The GeckoFramework's public interface is shown below:

```
class GeckoFramework
{
	GeckoFramework(
		struct in6_addr *	local_addr,
		char *			cfg_filename,
		char *			ifname );

	virtual ~GeckoFramework();

	int SetPolicy(GeckoPolicy * policy);

	int GetServerSocket(GeckoServerSocket ** serversock);

	int JoinDistributedServer();

	int LeaveDistributedServer();
};
```

The constructor of the `GeckoFramework` class expects three arguments. The `local_addr` argument is the address to use for handoffs, i.e. the care-off address of the replica. This address is different for each replica within the distributed server, and it is up to the application to set the right value. One solution might be to use per-replica configuration file that is parsed by the server application itself, or detect the primary address of the node, using the `gethostbyname2()` function from the standard C library. The `cfg_file` argument is the path to the framework's configuration file, which is discussed in the section 4.2.1. The `ifname` argument is the network interface on which the local_addr is bound. This parameter is needed for inserting a serialized TCP socket in the kernel, as a TCP socket must be bound to a network interface.

After instantiation of the GeckoFramework object, the framework is not fully initialized yet. One important property of the framework is the separation of the distribution mechanism from the policy. As discussed in section 4.1, the framework will include a set of ready-implemented policies to choose from, and offer a programmer the possibility to write its own. Some policies might require additional configuration, and must therefore be created by the server application after the framework is initialized. The policy is then bound to the framework using the `GeckoFramework::SetPolicy` method, where the `policy` argument is a pointer to the policy to be used. The GeckoPolicy class is actually a base class used to derive actual policies. More information about creating policies will be given in chapter 5.

In case policy is set by the application, no automatic distribution will be used and all incoming connections will be offered to the local application. This includes clients accepted by the contact node. A programmer wanting to use this functionality should understand the implications, because one can easily overload the contact node if it fails to handoff clients fast enough.

After initializing the framework, the node can join the distributed server using the `GeckoFramework::JoinDistributedServer` method. This method will spawn a separate thread responsible for distributing clients, and start the address management protocol. As our current implementation does not support contact node failure, the `GeckoFramework::JoinDistributedServer` method will only bind the contact address to the kernel needed to create the listener socket. In case the invoking node is the backup contact node with the highest preference, it will also update the binding at the Home Agent, so it can start accepting clients. In case an address management protocol is implemented, this function should start monitoring the contact node in case the invoking node is a backup contact node, so the contact address can be taken over if necessary.

Although invoking the `GeckoFramework::JoinDistrbutedServer` method enables one to start distributing connecting clients, it will not allow the node's distributed server application to process clients immediately. When the server application itself is ready to accept clients, it must use the `GeckoFramework::GetServerSocket` method to create a listener socket from which clients can be accepted. The resulting GeckoServerSocket object will be assigned to the `serversock` argument given by the server application. The use of this object is explained in the next section.

At any time, a node may decide to leave the distributed server. The framework allows a node to leave the network gracefully, by invoking the `GeckoFramework::LeaveDistributedServer` method. This method will first request the membership management protocol to leave the distributed server. As we currently do not support dynamic membership, no changes are actually made to the replica group view at peer replicas. Additionally, this function informs the address management protocol, so it too can announce its departure. The latter enables other nodes to take over the contact address in time, so the departure of a contact node does not result in the server being unavailable.

The `GeckoFramework::LeaveDistributedServer` method does *not* close an opened GeckoServerSocket from which clients can be accepted. In case a dynamic replica group membership protocol is in use, it will take some time for the departure-announcement to propagate to all other nodes. Therefore, clients can still be accepted from donors not aware of the node's leave yet. One has to manually close the application socket, in order to forcefully reject future clients.

Although `GeckoFramework::LeaveDistributedServer` should remove the node from the replica group, it is still possible that clients connect directly to a departing replica due to an unexpired binding. In case there is no open GeckoServerSocket, these clients are accepted but not offered to the distributed server application. Instead, the connection is closed immediately and the client is requested to remove its binding so subsequent connection attempts will yield the contact node.

It is up to the application programmer to either wait for all bindings to be expired after closing the GeckoServerSocket, or accept that the distributed server will be temporally unavailable for clients still holding a binding to the departing replica. The default current configuration of the modified MIPv6 daemon requires that bindings be refreshed every eight minutes, hence, if the first strategy is chosen a departing replica server application should wait eight minutes before terminating execution.

4.2.4 GeckoServerSocket

The GeckoServerSocket is not instantiated by the server application itself, but using the `GeckoFramework::GetServerSocket` method.

```
class GeckoServerSocket
{
       GeckoServerSocket (GeckoFramework * framework);

       virtual ~GeckoServerSocket ();

       int Close();

       int Accept(GeckoSocket ** clientsocket);
};
```

During its construction, the membership protocol is requested to join the replica group so other replicas will learn about the new node and start distributing clients to it. However, in the current framework the membership management protocol is only a stub and all replicas are be configured statically before the system is started.

A client can then be accepted using the `GeckoServerSocket::Accept` method. This function will construct a new GeckoSocket object for the accepted client, which interface is discussed in the next section.

The accepted client can be either distributed to the replica by the distribution policy, or handed over manually by a peer server application. The server application can differentiate between these two by examining the client's application state, which is discussed in the next section.

If a distribution policy is in place, connecting clients will always be distributed according to this distribution policy. Consequently, a replica server application to which a client connects will only serve the client if it is explicitly distributed to the replica. The contact node's server application is therefore not concerned about connecting clients. From the perspective of the application programmer, it will function as a normal replica accepting handovers from peers, including itself. However, in case no distribution policy is set, the `GeckoServerSocket::Accept` method will return all clients that are connecting to the machine, as discussed in the previous section.

If no client is distributed or handed off to the invoking replica, `GeckoServerSocket::Accept` may block forever. The behavior of the `GeckoServerSocket::Accept` is therefore similar to the `accept` system call on a blocking listener socket.

A replica may decide to stop accepting clients, using the `GeckoServerSocket::Close` method. Like the traditional close-method used on a regular listener socket, this method ensures that no new clients will be accepted anymore and future connecting clients will be rejected. The membership management protocol is requested to leave the replica group if the replica is still part of that group. It is up to the application to either close the connection for existing connected clients, continue to serve them, or hand them over manually.

Note that this method differs from the `GeckoFramework::LeaveDistributedServer` method in that it does not only announces the replica's departure, but also rejects future handoffs or connecting clients. However, the `GeckoServerSocket::Close` does not inform the contact management protocol. As a result, the invoking node may continue to function as a (backup) contact node.

After closing the socket, the object can be destructed safely. When the server application wants to re-open the server socket, it should invoke `GeckoFramework::GetServerSocket` again.

4.2.5 GeckoSocket

Instead of using raw file descriptors for communication, the server application has to use instances of the GeckoSocket class to read data from, or write data to a client. The primary reason for this is that we want to make the transport layer handoff transparent to the application programmer making use of our framework.

```
class GeckoSocket
{
	bool                              failed_handoff;
	struct in6_addr                   failed_handoff_target;

	GeckoSocket (GeckoFramework * framework);
	virtual ~GeckoSocket ();

	ssize_t Write(const void * buffer, size_t nbytes);
	ssize_t Read(void * buffer, size_t nbytes);

	void setState(void * buffer, size_t bytes);
	size_t getState(void ** buffer);

	int Close();
	int Close(bool resetbinding);

	int Handoff(bool nonblocking);
	int Handoff(struct in6_addr target, bool nonblocking);

	int SetNonBlocking();
	int SetBlocking();

	int Connect(struct in6_addr destination, int port);
	int Bind(int sd);
};
```

During a handoff, the socket is extracted from the donor and inserted at the acceptor using the TCPCP package, as discussed in section 3.3.1. This packet has some restrictions. The first is that the buffer containing received and acknowledged data is not transferred. The authors of the package decided to exclude this buffer in order to limit the file size variance of a serialized TCP socket. The framework has to manually read the contents of this buffer and add it to the handoff request sent the acceptor. Without a wrapper around the file descriptor, the acceptor's server application should be aware of this buffer, as it contains data that should be read first.

Additionally, the TCPCP package does not create an exact copy of the socket at the acceptor. It moves most of the kernel's structures to ensure correct continuation of communication, but options set on the socket by the application will be ignored. Among these are options like blocking/nonblocking IO and flow control. To make the transport layer handoff fully transparent to the application's programmer, we therefore decided to hide the file descriptor from the application programmer, and provide methods to communicate with the client and control the actual socket.

GeckoSocket objects are constructed by the GeckoServerSocket object after accepting a client using the `GeckoServerSocket::Accept` method. Upon return of this method, the server application should begin processing the client. This involves reading requests and/or sending a reply to the client.

Writing data to the client is performed using the `GeckoSocket::Write` method. Both arguments required by this method are equal to the arguments given to the `write` function found in the standard C library. The only difference is that the GeckoSocket's `Write` method does not accept a file descriptor, as it always uses the encapsulated file descriptor of the subject object itself. Consequently, `Write` writes `nbytes` of data to the client belonging to the GeckoSocket object, from the buffer pointed to by `buffer`. In case the connection is closed or handed off, `GECKO_ERROR_NOTALLOWED` will be returned, in case an error occurs while writing the data, `GECKO_ERROR_FAILED` is returned. If no error is encountered, the number of bytes written is returned, which might be smaller than the `nbytes` specified.

Received data can be read using the `GeckoSocket::Read` method. This method reads `nbytes` of data, and writes the data to the buffer pointed to by `buffer`. In case the connection is closed by the client 0 will be returned, in case an error occurs while reading data, `GECKO_ERROR_FAILED` is returned. For clients that are handed off or closed, `GECKO_ERROR_NOTALLOWED` will be returned instead. If no error is encountered, the number of read bytes is returned, which might be smaller than the `nbytes` specified. This function might return data received but not read by a replica that was serving the client before. However, the framework guarantees that the properties of TCP are obliged, so subsequent reads will return fresh data in the stream.

The connection with a client can be closed using `GeckoSocket::Close`. In case the connection is handed off or is being handed off, `GECKO_ERROR_NOTALLOWED` will be returned. Depending on the configuration of the framework, the binding of the client will either be left intact or removed. One can override this behavior by using the `GeckoSocket::Close` method accepting the argument `resetbinding`. By setting this argument to true the binding will be cleared, false will leave the binding intact. Note that the framework cannot guarantee the binding is cleared in case it is instructed to. MIPv6 uses unreliable IPv6 packets for signaling the client, hence the request sent to the client requesting it to clear its binding may get lost. The current modified MIPv6 daemon does not retransmit these requests.

Client Application State

The framework allows transparent distribution of clients connecting to the contact address. In case a distribution policy is set, the framework will transparently distribute all connecting clients using that policy. A replica receiving such a client will have to treat the client as if it was connected directly to the server application. Invocation of the `GeckoSocket::Read` method will therefore return data sent by the client after it was connected.

In contrast to this initial distribution of clients, explicit handoffs initiated by the server application itself are *not* transparent. The distributed server may already have received the client's request or sent data. The second motive for the GeckoSocket class is therefore bundle a file descriptor with an application-specific buffer.

This buffer is handed off together with a client, and consequently provides a means to the application programmer to inform the acceptor about the client's expectations, enabling the application level handoff as explained in section 3.3.2. One example is a Video on Demand server. Such an application might use a structure to combine information like the name of the video being streamed to the client and an offset so the acceptor can send the data required by the client to enable gapless playback. The memory buffer containing this structure is then attached to the GeckoSocket object, so it can be handed over upon invoking `GeckoSocket::Handoff`.

Because different applications require different knowledge over a client that is being handed over, it is up to the application to define a structure bundling the information required by the acceptor. This per-user structure can be attached to the GeckoSocket object using the `GeckoSocket::setState` method, which expects a pointer to the buffer containing the application structure and the size of the structure. The framework does not make a copy of this buffer, and therefore changes made to the structure after invoking this method will be available to the framework. Additionally, the framework is be made responsible for freeing the buffer holding the client state structure. Normally, the framework will free the buffer after a handover or upon destruction. One can detach the structure by providing the NULL pointer.

A pointer to the buffer can be retrieved using the `GeckoSocket::getState` method. The method will return the size of the buffer, or 0 in case no client state buffer is available for the client. Like the `GeckoSocket::setState` method, this function does not make a copy so subsequent changes will also be available to the framework.

Although the framework imposes no limit on the size of the application buffer, it should be kept to a minimum, as it needs to be transferred during a handoff. Additionally, one has to be aware that only the buffer holding the application structure is transferred to the acceptor. Any referenced objects will not be copied, while specified resource descriptors, such as file descriptors or database handles, cannot be transferred. It is up to the application programmer to include all required information in the buffer itself, and re-open files or open additional resources at the acceptor when necessary.

Handover

Handing over a client can be done either by specifying a target replica, or by having the distribution policy decide for a target. Both use the `GeckoSocket::Handoff` member, and depending on the presence of the `target` argument either of the two is used. The `nonblocking` argument species whether the caller should block on the method or not. If set to true, a background task will be made responsible for executing the handoff, and the function will return directly after the initial target is known. If set to false, the handoff will be executed within the thread of control of the caller and return only when either the client is handed off, or an error is encountered.

It can be the case that the distribution policy cannot select a target, either because no peer replicas are known or none of them is suitable. In such situations, the `GeckoSocket::Handoff` method reports this problem to the application by returning the `GECKO_ERROR_NOTARGET` error code.

A handoff may also fail because the target peer is unavailable. If no target is explicitly defined, the handoff method will automatically retry to handover the client by notifying the distribution policy of the failed handover and requested it to select an alternative target. By doing so, the framework recovers from replica group view that is slightly outdated because failures are not detected and known real-time.

But in case a target was explicitly defined or no alternative peer could be selected, the application must be notified. In case a blocking handover is used, GECKO_ERROR_TARGETERR is returned to the server application. But if a nonblocking handover is used, things are more complicated as the error is encountered after the method has returned. In such situations, the client will be handed off back to the application itself. Therefore, `GeckoServerSocket::Accept` may return clients that were deemed handed off before. To differentiate between a normal handover and a failed handover, the `GeckoSocket::failed_handoff` will be set to true, while set to false otherwise.

Although this member may introduce additional complexity for the application programmer, for normal usage it can be ignored. The framework is responsible for detecting replica failures and adjusts the replica group view accordingly. Therefore, if the application decides to handover the client again, the distribution policy should not re-select the problematic target, but instead immediately conclude that no suitable target is available. This error can then be presented to the application, by means of the `GECKO_ERROR_NOTARGET` error code discussed above.

We will illustrate this with an example. Consider a distributed server consisting of only two replicas. Requests for file 'a' must be processed by replica 'A', while requests for file 'b' must be served by replica 'B'. If replica 'A' receives a request for an unknown file, it may handoff the client, relying on a custom distribution policy to distribute the client to the correct replica. It executes a nonblocking handoff, but at the same time replica 'B' goes down. The policy at 'A' is not immediately aware, and selects replica 'B' as the target and the `GeckoSocket::Handoff` method returns. The handoff fails and the distribution policy cannot select an alternative target. Therefore, the client is handed over to 'A' itself. The replica continues to use the `Accept` method to accept new clients, and will eventually receive the client requesting 'b' again. Like normally, it will use the `Handoff` method, but because the policy is now made aware of the unavailability of 'B', the method will return `GECKO_ERROR_NOTARGET` immediately. By doing so, replica 'A' can return an error code to the client, such as the 404/not found exception defined by the HTTP protocol, and close the connection.

In case the target was explicitly defined, a nonblocking handoff will also handoff the problematic client to the local application in case the target was unavailable. Again, the `GeckoSocket::failed_handoff` will be set to true so the application can detect the failed handover. But to prevent the application from selecting the same target, the `GeckoSocket::failed_handoff_target` member is set to the IP address of the failed target. By examining this member, the application can learn about failed replicas and select an alternative.

Destruction

After a client is handed off, the corresponding GeckoSocket object must be destructed, as C++ offers no garbage collection to free up this memory automatically. For blocking handoffs, the server application must do this manually. But for nonblocking handoffs two approaches can be taken. The first is to copy the GeckoSocket object, so the server application can destruct these without interfering with the framework. The second approach is to let the framework use the GeckoSocket itself, and require the server application to leave the data untouched after the `Handoff` call returns successfully.

We choose to use the second approach for nonblocking handoffs, because it eliminates the need to deepcopy the object. Therefore, the server application should forget about a client for which the nonblocking `GeckoSocket::Handoff` returned success. Only in case an error was returned or a blocking handoff was used, the server application must free up these resources by itself. Because after a handover the GeckoSocket object is not bound to an active socket, invoking methods on it will result in `GECKO_ERROR_NOTALLOWED` error codes being returned.

Destruction of a GeckoSocket, either done by the framework or server application, will also free up the client state buffer, attached to the GeckoSocket object. Note that it will only delete this buffer; any other memory associated with the client must be freed by the application itself.

Because the file descriptor is a private member of the GeckoSocket instance, a programmer cannot set socket options directly. Instead, the framework provides methods to allow one to control the behavior of the socket. The `GeckoSocket::SetNonBlocking` method will make the socket nonblocking, while `GeckoSocket::SetBlocking` will make the socket blocking. By default, all sockets are blocking. In the future, the number of options that can be set may be extended according to the application programmer's needs.

One important property is that handoffs actually involve connected sockets. Therefore, one can handoff any communication endpoint, whether it is a connection to a client, a server or even a peer. Because of this, we added methods to the `GeckoSocket` for either connecting to another host or binding the object to an existing (socket) file descriptor. After construction of a GeckoSocket one can use `GeckoSocket::Connect` to connect to the IPv6 address specified by `destination`, using the port specified by means of the `port` argument. Alternatively, one can use `GeckoSocket::Bind` to bind the existing socket file descriptor denoted by `sd` to the GeckoSocket. The object can then be handed of as usual.

4.2.6 GeckoSelect

Servers can either serve one client at a time or multiple concurrently. The first type accepts a client and finishes serving it before accepting another client. For most TCP-based servers, this behavior leads to performance issues since other clients are unable to connect to the server when it is busy serving another client. Additionally, the server application cannot fully utilize the resources of its host since most of the time it has to wait for data to be sent or received.

Concurrent servers on the other hand serve multiple clients at once, increasing its capacity. The two most commonly used methods for programming this type of server either use a separate thread of control to process a single client, or depend on the `select` system call. The first method can be programmed using separate threads or processes. The framework does not support forking off new processes, as this would require inter-process communication to orchestrate the framework's functionality across multiple processes. Therefore, a programmer can only use threads, which run inside a single system process.

The alternative to using multiple treads of control is to use only a single one that serves all connected clients at once. Instead of waiting for `read` or `write` to complete, one can make the sockets nonblocking. The `select` system call is then used to wait for a socket to become available for reading and/or writing. Because this method requires access to the file descriptor, this method cannot be used in conjunction with our framework, which hides the file descriptor from the application programmer, as discussed in section 4.2.5.

The framework therefore includes the GeckoSelect class, which mimics the behavior of the traditional select function.

```
class GeckoSelect
{
	GeckoSelect(GeckoFramework * framework);
	~GeckoSelect();

	int MonitorSocketForRead(GeckoSocket * subject);
	int MonitorSocketForWrite(GeckoSocket * subject);

	int Select(
		GeckoSocketList *   readyForRead,
		GeckoSocketList *   readyForWrite,
		bool                monitorServerSocket,
		struct timeval *    timeout );

	int ResetReadSet();
	int ResetWriteSet();
};
```

The GeckoSelect object has to be instantiated with a pointer to the framework. After instantiation, one can add a GeckoSocket for monitoring using the `GeckoSelect::MonitorSocketForRead` and `GeckoSelect::MonitorSocketForWrite`, for reading or writing respectively.

After declaring all sockets that need to be monitored, the `GeckoSelect::Select` method can be invoked. It either blocks until one of the declared sockets has activity for which it is monitored, or for a restricted period if the `timeout` is a non-NULL pointer.

In case a monitored socket has activity upon return of the select function, it will be inserted in one of the GeckoSocketList's given as an argument to the `GeckoSelect::Select` method. If the socket is monitored for read and has data is available, it will be inserted in the specified `readyForRead` list. If the socket is monitored for write and data can be written, it will be inserted in the specified `readyForWrite` list. These lists are standard template library sets of GeckoSocket pointers, defined as:

```
typedef std::set<GeckoSocket *, GeckoSocketListCompare> GeckoSocketList;
```

Upon return, the server application can use a standard iterator to service all GeckoSockets that need attention.

The `GeckoSelect::Select` method returns 0 on success, or `GECKO_ERROR_FAILED` in case an error was encountered. It can also be used to monitor the GeckoServerSocket, so the program can be notified if a new client is ready to be. If `monitor_serverport` is set to true and a client is ready for acceptance, the `GeckoSelect::Select` method will return 1 instead of 0. It is important to give priority to accepting a new client, as the donor server application may be waiting for the handover to complete, and may decide to abort the handoff if the acceptor does not respond in time.

4.2.7 Deployment

This section discusses the requirements on the deployment of a distributed server written using our framework.

Kernel dependencies and modified MIPv6 daemon

Our framework depends on the current implementation of Versatile Anycast. As discussed in Chapter 3, this implementation consists of a kernel patch and a modified daemon. Consequently, this patch must be used on the kernel's source in order for the daemon to function. The modified Versatile Anycast daemon is based on the Linux-2.6.8.1 version of the MIPv6 implementation. The latest available version of the MIPv6 implementation provides hooks for the Linux-2.6.16 kernel, but is not supported by Versatile Anycast.

Besides the MIPv6 kernel patch, the framework also requires a modified TCP transport layer in order to support socket handoffs. The TCPCP package discussed in Chapter 3 is responsible for this, and as the transport layer is located inside the Linux kernel, it too includes a kernel patch. As the Linux kernel version is dictated by the underlying Versatile Anycast implementation, we use release 9 of the TCPCP package, which is modified by the creator of the Versatile Anycast implementation to support Linux-2.6.8.1 and TCP connections over IPv6.

The dependence on the old Linux-2.6.8.1 kernel is not a desirable situation. As the MIPv6 implementation used as the basis for the Versatile Anycast implementation is not actively developed anymore, it might be wise to change the MIPv6 basis to FMIPv6, which kernel hooks have been accepted into the main kernel source supporting the current Linux-2.6.23 kernel [14]. Additionally, support for IPsec must be added to guard against hostile takeovers as discussed in chapter 3. However, TCPCP only supports up to the Linux-2.6.11 kernel, and its successor TCPCP2 only supports the Linux-2.6.15 kernel [18]. Therefore, TCPCP should be ported to more recent kernels, an issue which is currently under investigation by the creators of the original Versatile Anycast implementation.

Configuration and Home Agent

The MIPv6 Versatile Anycast implementation requires one configuration file on each replica. This configuration instructs the daemon to use the route optimization mode, and specifies its own home address. For now, IPsec must be disabled explicitly because the current Versatile Anycast implementation does not support this functionality yet. If this is enabled in the future, each backup contact node should also be configured with the security association used between the Home Agent and the distributed server.

At present, only Cisco Systems has a MIPv6 home agent implementation for selected routers [16], although protection of the signaling between the mobile node and home agent using IPsec is planned for the future. Alternatively, one can configure a regular computer to act as a MIPv6 home agent. To this end, the unmodified MIPv6 implementation from [21] can be used and configured to run in home agent-mode. The machine running the home agent must be able to intercept all traffic destined to the contact address of the distributed server. Therefore, one should configure this node to act as a router, so it will accept traffic destined to the contact address, and process it accordingly.

Replicas

Replicas can be placed throughout the Internet. They all need to be configured with the same configuration file that contains the parameters required by the framework to control the distributed server. See section 4.2 for a full discussion on these parameters.

As our framework allows one to create a single executable inhibiting all distributed server roles (contact node, backup contact node and replica), the deployment of the distributed server application only depends on the application itself. For simple applications, all replicas may be exact copies, allowing one to use a single package that can be deployed anywhere. More advanced applications might deploy replicas inhibiting different functionality, for example because only a subset of the content is available at any replica. It is up to the application provider to manage the distribution of the replica software.

4.3 Proxy application

As our framework requires one to use the GeckoSocket class instead of a file descriptor, it is not immediate to modify existing software to use our framework. Existing server applications should be partially rewritten in order to cooperate with our framework.

But in some cases the existing software cannot be modified or rewritten. For example, because one does not own the source code, or the application is too complex. Moreover, our framework is only available for C++ programs, prohibiting applications written in another language to take advantage of Versatile Anycast client distribution.

For that reason, we offer an alternative to using the framework directly. We wrote a simple proxy application using our own framework that allows one to distribute clients without adapting the original server application. Like a normal distributed server application build using our framework, clients are accepted by the contact node's framework and distributed according to a distribution policy. Consequently, each replica application is able to accept such a handed over client. But instead of processing the client by itself, the replica application opens a connection to the actual server application and acts as an intermediary. It will write data read from the client connection to the server, while data read from the server will be written to the client, effectively joining the connections together.

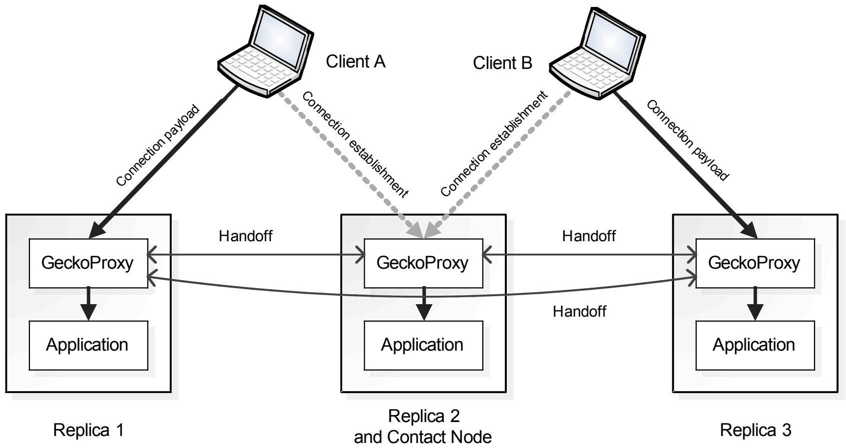

Fig. 8 Proxy application between clients and application servers

This behavior is illustrated using Fig. 8. On each replica runs the application server, with the proxy application in front of it. Clients do not connect to the application server directly, but to the proxy application. Our framework within the proxy application will then distribute the client directly after connection establishment to any peer proxy within the system. Once distributed, the proxy server application will open a new connection to the actual application and forward the connection payload accordingly.

The proxy eliminates the need to modify the application server. However, it can only distribute clients at the beginning of the communication session, as no application level handover is possible. The proxy application must be deployed according to the requirements described in section 4.2.7. It is configured using the following command line arguments:

`-g`	`string`	Path to the configuration file for the framework. Mandatory.
`-a`	`integer`	Port number used by the actual server application. Mandatory.
`-l`	`string`	Local IPv6 address to use for handovers. By default the first IPv6 address returned by `getifaddrs` will be used.
`-i`	`string`	Interface to use for handovers. By default eth0 will be used.
`-s`	`string`	IPv6 address of the application server. By default the loopback address will be used.
`-c`		If set, clients may only be distributed by the contact node, replicas must process incoming connections themselves.

Chapter 5

Architecture and Implementation

The API presented in the previous chapter allows one to build distributed servers with relative ease. Instead of requiring each application programmer to implement the same functionality, the framework abstracts the common parts. This means that the framework has to deal with the complexity of Versatile Anycast and transport layer handovers, in addition to common distributed server tasks like membership. The framework must be able to support a variety of applications and be scalable in both the number of replicas and clients. This made the implementation of the framework itself nontrivial, as many concerns needed to be addressed.

This chapter discusses the implementation of the framework. The first section will give a global overview of the framework's architecture together with a rationale for our design. We then show the main components are implemented and describe the issues encountered.

5.1 Design Choices

As the framework's architecture dictates functionality, the design decisions made while designing the framework influenced the API available to users. During the design, the ease of use of both the API and replica deployment was the most important.

Additionally, the framework must interoperate with the low-level Versatile Anycast implementation and the TCPCP library for the transport layer handovers. Versatile Anycast requires us to create and manage a contact address, which is bound to a contact node necessary for accepting clients and forwarding the return routability messages for every replica in the system, as discussed in section 4.1. The TCPCP library also influenced the architecture of the framework, as it does not move acknowledged –but unread data from the donor replica to the acceptor, as well as options set on the socket, as discussed in section 4.2.

5.1.1 Socket Wrappers

Because we wanted to make the transport layer handoff transparent to the application programmer, the properties of the TCPCP library required us to make a wrapper around the file descriptor used for communication with the client. This wrapper then takes care of the transport layer handoff. For clients that are accepted and processed on the same replica, such a wrapper is not necessary as the traditional functions for reading or writing data can be used without modification. But having two types of clients (with wrapper, and without) increases the complexity of our framework, and therefore we decided to make the wrapper compulsory for every client connection.

Additionally, the wrapper allows us to bundle relevant data. For each client, the server application can attach a buffer, containing application-specific information about the client as discussed in section 4.2.5.

5.1.2 Contact Address Management

A client can already have a valid binding for the contact address because it was connected to a replica recently. In case the client reconnects to the system, traffic will not be sent to the contact node but to the replica directly. Therefore, every replica has to listen for new clients, as well as handover requests. Consequently, there are two sources from which clients can be accepted by the server application.

Offering both the listener socket and a secondary source for accepting handovers to the application programmer will make the usage of the framework more complex. Consider a server wanting to accept a new client. It should first try one source; in case that source has no clients it should try the next. If the method for accepting a client from the first source is blocking, it may block indefinitely on that source and not consider the second. If the first also returns if no client is available, the worker may be blocked indefinitely on the second source. Making both sources nonblocking results in polling, which is undesirable.

The framework could provide a method that multiplexes both sources using the `select` function, provided they are both implemented using sockets. However, this method must be invoked by the user application, and one cannot force the application to invoke this method on a regular basis. Therefore, clients connecting to the replica directly may never get accepted, while other replicas might be willing to process the client. This is especially true for the contact node, which may even never try to accept a client. The solution we took is to spawn a second thread of control responsible for accepting connecting clients. Those clients must then be distributed according to the distribution policy, not only at the contact node but also on every replica within the distributed server. If the handover does not complete within a configurable timeout, this background thread should re-invoke the policy for an alternative target. This process must be continued until the client is successfully handed over to a replica. By doing so, clients can still be accepted and distributed to peer replicas, even in case the local server application is busy and does not accept new clients. If no policy is set, a client will always be distributed to the local application server, making the application programmer responsible for distributing clients timely and correctly.

5.1.3 Client Distribution

The distribution policy will regard the local server application as a regular replica that is potentially able to serve the client. Therefore, a client can be handed over from the background task to the local server application. One option is to queue these clients, and return them first on invokement of the GeckoServerSocket‘s Accept-member. But in case distributed (i.e. bounded) clients frequently make new connections, those new connections are given precedence over clients being handed over mid-connection. The latter group of clients can therefore starve as no replica ever may accept them, causing connections to be broken. Therefore, both types of clients must at least be given equal chances.

Secondly, queuing new connections alone is not enough. Like above, the application cannot be forced to accept new clients regularly. Because of this, clients being in the queue too long should be removed and an alternative target should be selected. Therefore, the background task should treat local handovers identically to as remote handovers.

For that reason, we need a single queue containing both the clients being handed over by a peer replica, and the clients handed over locally. Note that in both cases, the server's distribution policy selected the replica as the target. We also must allow the withdrawal of a handover request from the queue, in case the client is not accepted on time. Additionally, we should be able to limit the length of the queue, so handover requests that are likely not to be accepted in time can be rejected immediately.

A default TCP listener socket provides these properties. A handover request can be accepted one at a time, while pending requests will be queued up to the size of the backlog parameter. Therefore, our framework uses a single TCP listener socket from which both remote and local handover requests are accepted. Additionally, we use TCP for handover requests because due to the variable size of the application buffer and serialized TCP client endpoint, the complete request is likely not to fit into a single UDP datagram.

As a result, the `GeckoServerSocket::Accept` method does not accept a client, but a handover request instead. The handover request is then processed and yields a new GeckoSocket object to be returned to the server application. In case an error is encountered, the handoff is aborted and the thread is blocked again on accepting a new handover request.

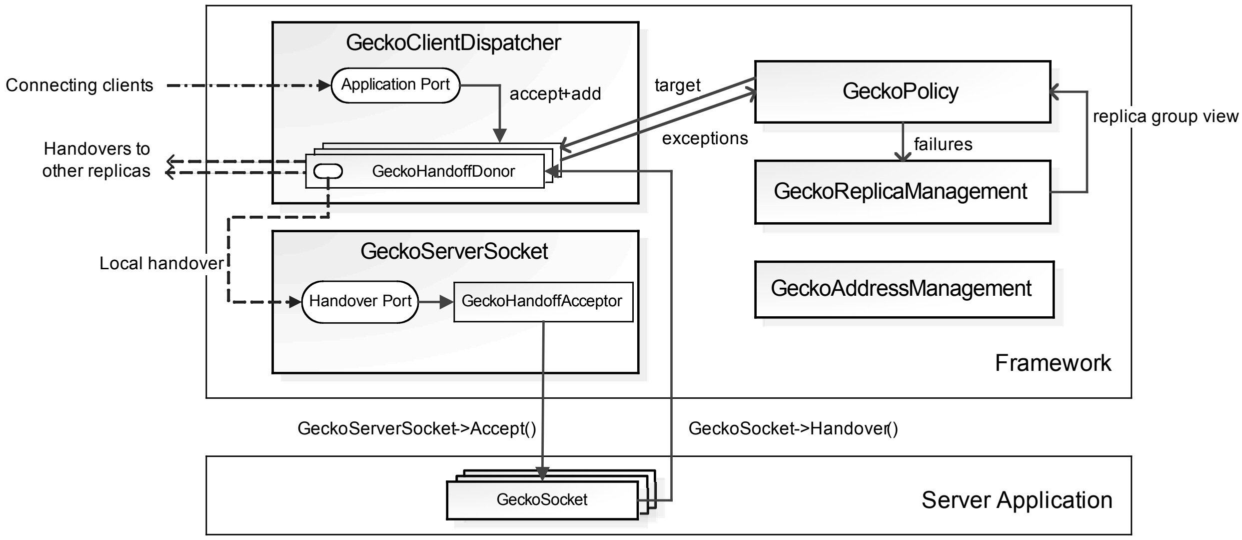

Fig. 8 Global overview of the framework

5.2 Architecture

The architecture of the framework is summarized in Fig. 8. At the bottom of the figure is the server application, containing the code to read the client's request and send the reply to the client. At the top is our framework, responsible for accepting and distributing clients. The GeckoClientDispatcher, GeckoServerSocket, GeckoReplicaManagement and GeckoAddressManagement are created by the GeckoFramework class, which is used to parse the configuration file and glue all objects together by holding references to them.

The actual handover procedures are contained within the GeckoHandoffAcceptor and the GeckoHandoffDonor classes. The donor instantiates a GeckoHandoffDonor object to coordinate and execute a single handover of a given GeckoSocket object. To this end, the GeckoHandoffDonor object will connect to the acceptor's handover port, and the connection will be queued until the server application at the acceptor executes the `GeckoServerSocket::Accept` method. This method will create an instance of the GeckoHandoffAcceptor class, which will read the handover request and reconstruct the GeckoSocket object on the accepting replica. If successful, the donor will be acknowledged and the GeckoSocket will be offered to the server application. Upon receipt of the acknowledgement, the both objects should be destructed.

The GeckoClientDispatcher class accepts new connections to the distributed server. Instantiation of this class will spawn the background task described above, for accepting clients connecting to the replica's application port. After accepting such a client, it will invoke the policy and construct a GeckoHandoffDonor to execute the handover. The framework is designed to execute multiple handovers at once, making the distributed server more scalable. Additionally, this architecture allows us to run the same software on all nodes in the system, instead of separating the contact node from the replica software.

The GeckoPolicy class is responsible for selecting a target replica. It is actually a base class, and multiple policies can be defined by deriving this class, as explained in section

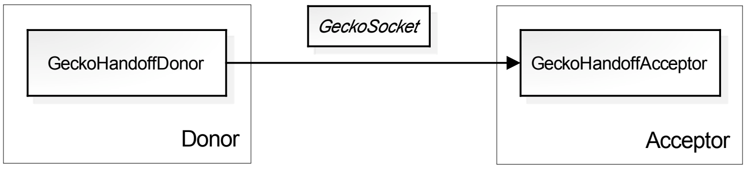

Fig. 9 Handover of a GeckoSocket object

4.1.2. The policy should interoperate with the GeckoReplicaManagement object, as the latter maintains the replica group view. Handover exceptions are also reported to the policy, so it can deal with them accordingly, by for example removing the replica involved from the group view or temporally exclude the replica from target selection. Hence, upon failure the policy should not return the same target but select an alternative instead. More details on the GeckoPolicy class will be given in section 5.4.

Currently, the GeckoReplicaManagement class does not support a dynamic membership protocol but uses a static list of replicas defined using the configuration file. A dynamic membership protocol like Cyclon discussed in section 4.1.1 can be implemented by deriving this class. It is likely that another thread of control should be spawned to support the protocol, as the frameworks gives no guarantees about how often one of the GeckoReplicaManagement's methods is invoked.

Management of the contact address is contained in the GeckoAddressManagement class. This class is created during initialization, but currently it only supports a fixed contact node. In case the node instantiating this class is the first backup node defined in the configuration file, it will register the contact address at the home agent, so clients can start connecting to the system. In the future, the class should manage the contact address dynamically. Backup contact nodes should actively monitor the current contact node and decide for a replacement if necessary. Hence, the GeckoAddressManagement class could implement the contact node management protocol explained in section 4.1.3.

Additionally to monitoring, the backup contact nodes should broadcast the address of the new contact node in case the contact address is taken over to all replicas in the system. In the event of a contact node failure, all handovers to peer replicas will fail, as the return routability cannot be performed. Handover functionality is restored upon registration of the new contact node's care-off address at the local MIPv6 daemon.

5.3 Handover Procedure

The handover procedure is used to distribute connecting clients and to support applications that require handing over a client in mid-connection. During a handover, a GeckoSocket object is replicated at the acceptor and the original is removed from the donor. Two classes orchestrate this handover. The GeckoHandoffDonor class is instantiated at the donor to handover a GeckoSocket object, while a GeckoHandoffAcceptor class is instantiated at the acceptor to receive a GeckoSocket object, as shown in Fig. 9. The same procedure is used for both handovers to the local server application and to peer replicas.

5.3.1 Communication Between Replicas

Both classes communicate using a TCP socket. Each replica willing to accept clients must create a GeckoServerSocket, as explained in chapter 4. During the construction of this object, a listener port is created for accepting incoming handover requests. For communication, a structure is used which specifies the type of the message and parameters to use.

```
struct gecko_comm {
	enum gecko_opcodes gecko_opcode;
	union {
		struct gecko_op_oh_prepare		params_op_oh_prepare;
		struct gecko_op_oh_handoff		params_op_oh_handoff;
		struct gecko_op_handoff			params_op_handoff;
		struct gecko_op_finished		params_op_finished;
	};
};
```

The message type is specified using the `gecko_comm::gecko_opcode` member. The donor can choose to either execute the normal or optimized handover protocol. Currently, the framework only supports the normal handover protocol. The optimized handover protocol requires more interaction between the framework and the server application, since the latter needs to notify the framework that a client will be handed over in the near future, complicating the design of both the framework and server application.

Optimized handovers also require more communication. First, the donor has to notify the acceptor so the latter is able to execute the return routability procedure in advance. Secondly, the donor has to send the actual client to the acceptor, so the acceptor can take over by recreating the GeckoSocket object and switching traffic. The first action is communicated by sending a message containing the `GECKO_COMM_OP_OH_PREPARE` operation code, the latter using `GECKO_COMM_OP_OH_COMMIT`. Execution of both have to be carefully orchestrated, as the return routability has to be finished first by the acceptor, before executing the binding update, making optimized handovers hard to implement in the presence of failures.

A normal handover procedure is initiated by sending a message containing the `GECKO_COMM_OP_HANDOFF` operation code. The anonymous union's `gecko_op_handoff` member is then used to communicate the data required by the acceptor:

```
struct gecko_op_handoff {
	int		mipv6_seqnr;
	struct gecko_client_socket client_socket;
	size_t		tcp_ici_sz;
	size_t		tcp_recvbuf_sz;
	size_t		appl_data_sz;
};
```

The mipv6_seqnr member is set to the sequence number of the previous binding update, and is retrieved using the mip6_handoff_mark function discussed in section 3.2.3. The client_socket member is a structure holding the state of the GeckoSocket object and various options set on the socket, such as nonblocking-mode. As the size of the serialized TCP endpoint, the buffer holding acknowledged but unread data and the application client state buffer are highly variable, they are not included in the structure themselves. Instead, their sizes are communicated using the tcp_ici_sz, tcp_recvbuf_sz and appl_sata_sz members respectively. In case one of them is not applicable, for example because the client does not have an application state, the value of the corresponding member will be set to zero.

After writing the gecko_op_handoff message-structure to the acceptor, the donor must write the contents of the buffer holding the serialized TCP endpoint. The acceptor can reconstruct the buffer holding this endpoint using the corresponding size-member of the handover request structure. After the endpoint, the TCP read buffer must be written, followed by the application state buffer.

After the acceptor has received all data, it can reconstruct the GeckoSocket object. The result is communicated to the donor using either the GECKO_COMM_OP_SUCCESS or GECKO_COMM_OP_FAILED operation code. The first denotes that the acceptor has successfully taken over the client; the latter notifies the donor that an error was encountered, so the donor can select an alternative target. Both communicate the client's IPv6 address using the params_op_finished member of the anonymous union:

```
struct gecko_op_finished {
      struct in6_addr client;
};
```

5.3.2 Parallel Handovers

A GeckoHandoffDonor object can either be created by the background thread responsible for accepting clients, or by an application server thread that manually invoked the handoff method on a GeckoSocket object. As explained in the previous chapter, a user can specify whether the handover method should return upon finish, or immediately.

One solution to return immediately is to create a new thread responsible for executing the handover. However, the cost of creating a new thread is significant compared to the time required for a handover to complete [9]. Additionally, if a separate thread is created for each connecting client, we will have no control over the number of simultaneous threads. The contact node must be able to sustain a high number of connecting clients, but a high number of threads is likely to cause performance degradation due to all threads competing for the same processor and other resources.

One solution would be to construct a queue of pending handovers and have a separate thread execute the handovers consecutively. Clearly, having a single thread will limit the rate at which handovers can be executed, as most of the time the thread will be waiting for read or write operations to complete. The use of a multiple such threads will increase the throughput, although synchronization mechanisms will complicate the design of the framework.

We decided to use nonblocking communication with an acceptor, and all pending handovers are executed in parallel using the `select` system call. The GeckoClientDispatcher class is responsible for managing these pending handovers. Inside the class is a standard template library set of GeckoHandoffDonor object pointers for every GeckoSocket being handed over. Access to this set is protected by means of a mutex. The background thread responsible for accepting clients will lock the mutex and collect all file descriptors of pending handovers. After unlocking the mutex, it will use `select` to wait for activity on any of those descriptors. After select returns, it will request the corresponding GeckoHandoffDonor objects to continue execution. This GeckoHandoffDonor class is discussed next.

5.3.3 GeckoHandoffDonor

A simplified definition of the GeckoHandoffDonor class is shown below:

```
GeckoHandoffDonor {
	GeckoHandoffDonor(GeckoFramework * framework, GeckoSocket * clientsock);
	int StartNormalHandoff(struct in6_addr * target, bool nonblocking);
	int ReadyToReadOrWrite();
	int HandleTimeout();

	bool   waitingToWrite;
	bool   waitingToRead;
	bool   use_timeout;
	struct timeval      timeout;
	int    acceptor_sd;
}
```

The class is instantiated with a reference to the GeckoSocket to be handed over. The `GeckoHandoffDonor::StartNormalHandoff` is then used to start execution of the handover. The first argument is the address of the target replica, which determined by the policy or provided by the server application. The second argument specifies whether blocking mode (false) or nonblocking mode (true) should be used.

In case blocking mode is used, this function will return after successfully handing over the client or upon a failure. In case nonblocking mode is used, the function will return directly after the nonblocking connect function call returns and the object is added to the GeckoSocketDispatcher‘s set of pending handover.

The `GeckoHandoffDonor::waitingToWrite` and `GeckoHandoffDonor::waitingToRead` members are used to notify the GeckoSocketDispatcher’s background thread in which file descriptor sets the handover connection file descriptor must be added. The former member is set to true if the GeckoHandoffDonor object is waiting for the file descriptor to become ready for writing the request, while the latter is set to true if the object expects an acknowledgement from the acceptor.

After the filedescriptor has become ready for either reading or writing, the background thread will use the `GeckoHandoffDonor::ReadyToReadOrWrite` method to return control to the GeckoHandoffDonor object. This method checks the current status of the handover, and continues accordingly. In case an error is encountered a negative value will be returned to the background thread, so it can select an alternative target and retry. In case the handover is executed successfully, a zero will be returned so the object can be removed from the set of pending handovers. A one (1) is returned in case the handover is not finished yet and the object is waiting for the file descriptor to become ready again.

During the handover, the donor will go through a number of steps. Each step corresponds to a state the GeckoHandoffDonor can be in. These are listed below:

`STATE_UNINITIALIZED`	Object constructed, but no handover is in progress yet.
`STATE_CONNECTING`	The connect system call has been executed but the connection is not established yet.
`STATE_CONNECTED`	The connection to the acceptor has been established, and the request is being constructed.
`STATE_WRITING_REQUEST`	The handover request is being written to the acceptor.
`STATE_WRITING_TCPICI`	The serialized TCP endpoint is being written to the acceptor.
`STATE_WRITING_RECVDATA`	The buffer containing acknowledged but unread data is being written to the acceptor.
`STATE_WRITING_CLIENTSTATE`	The buffer containing the client's application state is being written to the acceptor.
`STATE_WAITING_ACK`	The complete request is written to the acceptor, and the donor waits for the acknowledgement to be returned.
`STATE_CLEANING`	The acknowledgement is received, the connection is being closed and the GeckoSocket object will be destructed in case nonblocking mode is used.
`STATE_FINISHED`	The handover is completed successfully.

These states are traversed in the exact order as shown above, although some states may not be applicable as the three buffers may not be present. Upon encountering an error, the GeckoHandoffDonor will go into the `STATE_FAILED` state and the distribution policy will be notified, which is discussed in section 5.4.

Note that the request is constructed *after* the connection to the acceptor succeeds. In the mean time, the connection with the client is still active although the server application can not send any data to the client. The socket is frozen only upon succesful connection to the acceptor, minimizing the time the distributed server's connection endpoint does not respond to traffic sent by the client. Hence, the connection should overcome multiple sequential unresponsive replicas.

Additionally, the user can specify a maximum time to wait for a handover attempt to complete, so one has not to wait for the kernel's connect timeout to expire or for a broken connection to be detected, but instead recover earlier unresponsive replicas. Before execution of the `connect` system call, the GeckoHandoffDonor sets the `GeckoHandoffDonor::timeout` member to the current time added with the timeout specified in the configuration file using the `HandoverConnectTimeout` directive. The background thread will inspect this member only if the `GeckoHandoffDonor::use_timeout` member is set to true. In case the timeout is reached, the object is notified using `GeckoHandoffDonor::HandleTimeout()` and an alternative target is selected. It then computes the maximal interval for `select` to wait for other timeouts, so control is returned in time to handle those timeouts.

The maximum time for a handover to succeed currently only applies to nonblocking handovers. A blocking handover may therefore block for a longer period of time, before the failure is reported.

The donor only forgets about a client after it has received a `GECKO_COMM_OP_SUCCESS` message. If for whatever reason the connection with the acceptor is broken before the positive acknowledgement is received, it will always deem the handover failed and retry. Note that an alternative acceptor should do no harm in case the client has been handed over successfully, since the new binding update will be rejected due to an old sequence number. However, the current Versatile Anycast implementation does not handle exceptions correctly, preventing the framework from detecting these events. The next chapter will discuss these issues in more depth.

Besides executing nonblocking handovers, the background thread is also responsible for monitoring the local application port to which clients connect. It will accept new clients and invoke the policy to select a target for the client. Consequently, it will create a GeckoHandoffDonor object to distribute the client nonblocking, and add it to the set of pending handovers after exclusive access to the set is granted by means of the mutex.

5.3.4 GeckoHandoffAcceptor

As explained in section 5.1, the `GeckoServerSocket::Accept` method will accept a handover request from the `GeckoServerSocket`'s handover listener socket. After a donor is accepted, a GechoHandoffAcceptor class is constructed. This class will read the request and additional buffers if necessary, and construct a new GeckoSocket object using the data received by the donor. After traffic is switched successfully, the transport level handoff is completed by invoking methods on the created GeckoSocket object, which is discussed next. Upon success, a handover request structure is written to the donor.

In case the client could not be reconstructed or rebounded to the acceptor, the acceptor will inform the donor so the latter can retry to handover the client to another target. In contrast to the GeckoHandoffDonor class, the GeckoHandoffAcceptor class does not support a nonblocking mode of operation. Therefore, the thread accepting a handover will always be blocked until the handoff is complete.

5.3 Transport Level Handover

The public interface of the GeckoSocket available to the user of our framework is discussed in section 4.2. Besides these public methods, a number of additional methods are available for the framework itself to enable the transport level handover.

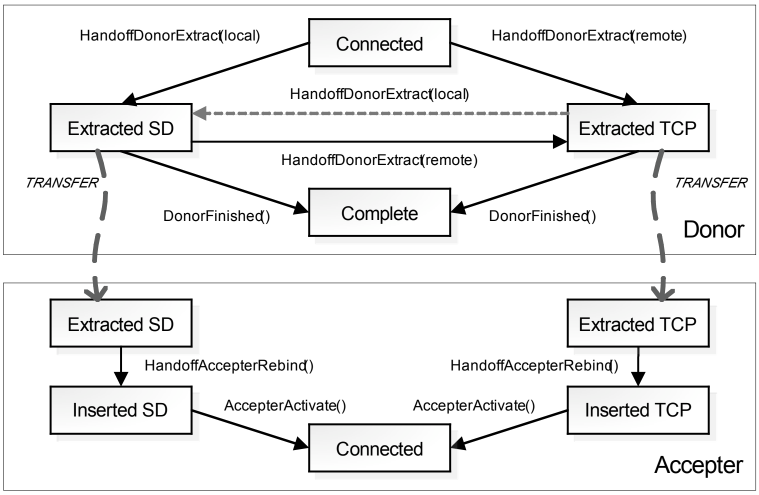

Fig. 10 Handover states and methods of the GeckoSocket class

The transport level handover involves serialization of the TCP endpoint and reading the buffer containing acknowledged data not yet read by the server application. Both are then transferred by the GeckoHandoffDonor and GeckoHandoffAcceptor objects, as discussed above. For clients that are handed over in mid-connection, an additional client application state buffer may be copied to the acceptor as well, to enable the application level handover.

In case a client is accepted at a replica and distributed to the local application, special measurements must be taken. For those clients, serialization and transfer of the TCP endpoint is not necessary because the TCP endpoint is not moved between machines. Moreover, it is impossible to reinsert the serialized socket, as the old socket has to be kept until the handover is complete. A machine can never have two sockets for the same connection, so the old socket should be closed and removed before the handed over socket is inserted. But this means that during removal and insertion the transport layer has no knowledge over the connection so any traffic sent by the client will result in RESET messages, as explained in section 3.3.1. Additionally, extracting and insertion of a TCP socket will result in additional overhead, which is undesirable. Therefore, in case of a local handover the file descriptor is communicated instead of the serialized endpoint.

The differentiation between local and remote handovers and number of carefully orchestrated steps required by a TCP handover leads to some complexity. The GeckoSocket abstracts this complexity from the rest of the framework, and provides a set of functions that extract, insert or activate a socket, as shown in Fig. 10.

Normally, a socket is in the CONNECTED-state, in which data can be sent or received by the server application. The application can either close the connection, which will transition the socket into the CLOSED state (not shown), or start a handover. Next to manually handover a client, the handover can also be initialized by the framework to distribute a client, as explained in the previous section.

The GeckoHandoffDonor object invokes the `GeckoSocket::HandoffDonorExtract` function to prepare the socket for serialization. This method accepts a single parameter, which is set to true in case the handover is local, or false in case the handover is made to a peer replica. This will transition the GeckoSocket into the EXTRACTED_SD (handover using the socket's (file) descriptor) or EXTRACTED_TCP state respectively. These states prohibit the use of any other function except those shown in Fig 10, so the state of the transport endpoint cannot change anymore.

Information about the socket such as object state, operation mode and file descriptor are contained in the `GeckoSocket::socket` member, which is always included in the handover request, using the `gecko_op_handoff::client_socket` member of the handover message structure. In case the socket is extracted, the `GeckoSocket::tcp_ici` member will point to the buffer containing the serialized socket. In case data not yet read by the distributed server is available, the `GeckoSocket::tcp_rbuf` will point to a buffer holding this data. In case client application state is available, the `GeckoSocket::appl_buf` member will point to a buffer holding the application structure. These pointers are used directly by the GeckoHandoffDonor object to create the request and write their contents to the acceptor.

The GeckoHandoffAcceptor object running on the acceptor then creates a new GeckoSocket object and overwrite the blank GeckoSocket's socket member with the received one. If applicable, each additional buffer (serialized endpoint, received data and application state) is read into a new buffer and subsequently assigned to the corresponding GeckoSocket member. Effectively, this will create an exact copy of the GeckoSocket object residing at the donor, including its current state.

The acceptor then executes the `GeckoSocket::HandoffAcceptorRebind` method. In case the replicated object is in the EXTRACTED_TCP state, this will insert the serialized TCP endpoint and transition the state into INSERTED_TCP. In case the object is in the EXTRACTED_SD state, it will transition the object into the INSERTED_SD state. In case the donor is another machine, both the donor and acceptor will now hold a frozen TCP endpoint and consequently traffic can be switched by the GeckoHandoffAcceptor object using the low level Versatile Anycast implementation, relieving the donor replica.

The GeckoSocket object is transitioned into the CONNECTED stated by invoking the `GeckoSocket::HandoffAcceptorActivate` method, which may unfreeze the endpoint in case it was inserted (INSERTED_TCP). Additionally, the acceptor has to acknowledge the donoring replica. Upon receipt of the acknowledgement, the donor must invoke `GeckoSocket::HandoffDonorFinished` prior to destructing the old GeckoSocket object. This method will close and remove the socket from the kernel *only* if the client is in the EXTRACTED_TCP state, as closing a socket handed over locally (EXTRACTED_SD) will affects the reconstructed GeckoSocket object.

If one of these methods fails, it will report the error and transition the object into the STATE_FAILED state and close the client socket. In addition failures caused by GeckoSocket methods, a handover may fail, for example because the acceptor reports an error or the connection fails. In such cases, an alternative target should be selected, and a new handover should begin from scratch. As a result, a GeckoSocket object being in the EXTRACTED_SD state might need to be handed over to a remote replica. For that reason, the GeckoSocket allows the framework to re-invoke the `GeckoSocket::HandoffDonorExtract` method to transition the object into the right state for either local or remote handover. However, the current implementation of TCPCP does not allow one to re-activate a frozen socket at the donor. Instead, the socket will remain frozen and hence communication will be broken. This issue should be fixed in future versions of TCPCP.

5.4 Target Selection

The target for a handover can either be specified by the distributed server application itself, or implicitly by the distribution policy. The latter has to cooperate with the membership protocol to gain knowledge over the current replica group. We first discuss how one can implement a distribution policy, and describe the interaction between the handover procedure and policy. Then we discuss how the policy interacts with the membership protocol.

5.4.1 Distribution Policies

As discussed earlier, the distribution policy is separated from the mechanism. This allows us to include a set of ready-implemented distribution policies to choose from. Additionally, a programmer can write additional ones by deriving the GeckoPolicy base class:

```
class GeckoPolicy {
       GeckoPolicy(GeckoFramework * framework);
       virtual ~GeckoPolicy();
       virtual int SelectTarget(GeckoSocket * client, struct in6_addr * target);
       virtual void NotifyReplicaException(
              struct in6_addr * replica_addr,
              enum gecko_replica_exception exception);
}
```

The server application has to instantiate the selected policy by itself. By doing so, policies may require additional configuration by the server application before selecting the first target. After instantiation of the policy, the framework is requested to use the policy for future handover target selection. If no policy is set, the default policy is used which always distributes an accepted client to the local server application.

A target is selected using the `GeckoPolicy::SelectTarget` method. It accepts two arguments: the GeckoSocket that is to be handed over, and a pointer to a buffer to which the target's IP address must be written. Any non-zero return indicates an error. In case no suitable target is found, the policy should either return the local replica's address to make the local server application responsible for the client, or return GECKO_NOTARGET, causing the client's connection to be closed.

For various reasons, the handover to the selected target replica may fail. In such cases, the policy is notified by the GeckoHandoffDonor object using the `GeckoPolicy::NotifyReplicaException` method, prior to selecting an alternative target. This method will report the IPv6 address of the failed replica, and a value indicating the type of exception encountered. Currently, the framework supports these exceptions:

`GECKO_REASON_NOREASON`	No specific reason.
`GECKO_REASON_ETIMEDOUT`	A timeout was encountered while connecting to the replica.
`GECKO_REASON_ECONNREFUSED`	The target replica explicitly refused the connection.
`GECKO_REASON_ENETUNREACH`	The target replica could not be reached.

It is up to the distribution policy to handle these exceptions accordingly. One option might be to immediately remove the replica from the replica group, or temporally exclude it as a target. At any time, the distribution policy must make sure the same target is not selected again for a failed handover, as the client must be handed over within a reasonable amount of time.

It is important to understand that multiple threads of control can access the distribution policy at the same time. For example, the background client accepter thread may invoke the policy at the same time as the server application executes a handover. Therefore, one must ensure no race conditions occur by using the synchronization mechanisms offered by the pthread library.

5.4.1 Replica Group View

Knowledge about the current replica group is not maintained by the distribution policy itself. It is done by the GeckoReplicaManagement class, as discussed in the beginning of this chapter. This class administers a set of IP addresses of peer replicas to which clients can be handed over. Currently, the list is bootstrapped with the complete set of replicas available to the system using the framework's configuration file. In case an replica exception is encountered, the replica can be removed from this list using the `GeckoReplicaManagement::DeleteReplica` method. Within a dynamic group membership management protocol, this update could be propagated to other replicas as well to prevent them from using the problematic replica. Additionally, a dynamic protocol should offer means to add replicas to the distributed server, which is currently not supported.

A current view of the replica group can be obtained using the `GeckoReplicaManagement::GetReplicaList` method, which returns a pointer to a standard template library set of IP addresses. The distribution policy then uses this list to select a target. Because copying this list upon each distribution policy consultation may result in excessive overhead, no copy is made but a pointer to the original list maintained by the GeckoReplicaManagement class is retuned instead. Therefore, we need a synchronization mechanism that prevents the list from being alternated while the policy is deciding for a target. This is done using `GeckoReplicaManagement::LockList` and `GeckoReplicaManagement::UnlockList`, to either get exclusive access to the set or return it.

Although a distribution policy may retrieve the replica group view each time the policy is invoked, this can be expensive in case the handover rate tends to be very high or a reasonable amount of computation is needed for each target selection. More advanced policies may therefore cache their result, so subsequent consultation may be executed more rapidly. One example of such a policy is client distribution based on network location, where a mapping of clients to replicas may be computed in advance, as explained in [10]. Additionally, a background thread may be spawned to execute administrative tasks to support the policy.

Some policies may require dissemination of information. Although a policy may create its own overlay network, this common functionality may be added to a dynamic membership protocol. Examples of such policies are distribution based on load or available bandwidth, or to improve cache hits by redirecting similar requests to the same replica.

Chapter 6

Evaluation

The objectives for this thesis stated in chapter 1 are to build a framework for distributed servers that is easy to program and deploy, while retaining the good properties of Versatile Anycast. In the first section of this chapter we evaluate our framework against the non-functional requirements of a good distribution mechanism. We then focus on a demonstration of the ease of use of our framework, after which we present a performance analysis of the framework. We conclude this chapter with a discussion of the problems encountered during a large-scale deployment, which gave us new insights and revealed a number of issues in the current Versatile Anycast implementation.

6.1 Evaluation on Requirements

In Section 2.1 we discussed a number of non-functional requirements that a good client redirection system should have, and showed that none of the current solutions satisfies them all. We believe that our framework satisfies all the required properties, as discussed here.

No Client Side Support

Because the framework uses Versatile Anycast to switch traffic, the client's networking stack must offer support to allow for route optimization. Currently, Windows XP SP1 and beyond [19] offer this functionality by default. Other operating system have optional support for MIPv6 [20], and it is our belief that the required functionality will be widely available at the time IPv6 will be the standard Internet protocol [17]. If the client does not support route optimization, the system reverts to tunneling as discussed in section 3.1. Because Versatile Anycast operates within the Internet layer, the client does not have to be aware of multiple replicas. Therefore, the framework can support most clients that use TCP to communicate with the server.

Distribution Customization

One advantage of Versatile Anycast is that every client can be distributed individually. In contrast to most of the mechanisms discussed in section 2.2, no mapping of clients to replicas has to be defined in advance. Additionally, no caching mechanism plagues the distribution by making attachment stale, except for the binding cache at the client, which can be cleared upon connection termination.

The framework separates the distribution policy from the mechanism, allowing its users to control and change the policy according to the application's needs. Besides using a ready-implemented policy, one may also design and implement its own, allowing for content-aware redirection.

In addition to managing the distribution by means of a policy, the server application can also manually hand over clients to a specified target, as explained in chapter 4, allowing for mid-connection handovers. To facilitate the application level handover, the framework uses application-specific buffers, which are transparently moved from donor to acceptor during a handover. As each application has specific needs, it is up to the programmer to structure this buffer and provide all data to the acceptor to guarantee uninterrupted communication with the client.

Resilience to Replica Group Change

The framework is also able to recover from replicas that left the replica group, either graceful or due to an error. Although the framework currently does not use a dynamic membership protocol to propagate changes to the replica group, the handover procedure uses timeouts to recover from unresponsive target replicas. This exception is consequently reported to the distribution policy so it can update the group view accordingly and select an alternative target.

One issue not solved by the framework is recovery of clients from a failed replica. If a replica application crashes or the node running the replica experiences problems like power outage or network disruption, clients currently bound to that replica will experience unavailability of the server. Not only is their connection lost, new connections will fail because the binding cache at the client still holds a valid binding to the unresponsive replica. One solution is to have the replicas monitor each other and take over the failed-replica's clients if necessary. However, such functionality is very complex and application specific.

Because of the static membership of replicas, replicas must currently be known before the system is started. This limits the scalability on the number of replicas, as one cannot add a new replica once the server is running. Additionally, temporally failure of a replica can cause other replicas to remove the subject replica to prevent it from being selected. With dynamic membership, the replica should rejoin the replica group and announce its identity upon recovery.

No Unnecessary Delays and Internet Scale Deployment

Versatile Anycast allows one to distribute clients instead of connections. Because a handover switches all traffic from a client targeting the contact node, future or parallel connections will use the same replica. This behavior can be helpful for services that issue a large number of sequential requests for small data-units, for example a webpage that consists of multiple images. The client needs only be distributed once at the beginning of the first connection, eliminating the need to distribute subsequent connections, leveraging the load at the contact node. Additionally, successive connections can be setup without any added delay or load at the contact node.

During a handover, no communication between the distributed application server and client is possible. After the connection with the acceptor is established the donor freezes and extracts the TCP endpoint, also stalling transport layer communication. TCP communication is restored upon inserting the endpoint at the donor and switching traffic.

Because the framework is designed to perform multiple handovers at once, the handover can start immediately after a target is chosen or the application manually hands off the client. The time required to finish the handover is mostly dictated by communication latency due to interaction between the nodes involved. Therefore, the handover delay is composed of:

- The sending and receipt of the handover request: initialization of the TCP connection and the transfer of the request;
- The return routability procedure: the time required by the HoTI to travel from the acceptor through the contact node and home agent to the client, and the HoT to travel the reverse path;
- The route optimization procedure: the time required by the binding update and acknowledgement to travel to and from the client respectively;

Communication latency depends on the location of each node involved, bandwidth, and the size of the serialized TCP endpoint. Experiments in [9] show that with a communication latency of 20 milliseconds between the donor and receiver and a 40 ms latency between the acceptor and client, a handover can be completed within 400 to 950 ms, depending on the state of the TCP buffers and available bandwidth. Although latencies in the Internet may be a multiple of the latencies used in the experiments, we believe a handover is finished well before the client's TCP endpoint times out or the client experiences problems.

As communication over the Internet is unreliable, packets may be dropped or delayed. Lost MIPv6 messages are automatically resent every second, while the frameworks relies on TCP to deliver the handover request to the acceptor. In case this TCP connection is broken before the complete request has been written, an alternative target will be selected and the procedure will start from the beginning. Additionally, the framework assigns a configurable timeout value to recover from an unresponsive replica before TCP ultimately gives up. Such events will increase the handover time, but in section 6.3 we present an experiment in which the framework overcomes multiple successive replica failures without breaking the connection to the client.

In addition to the delay caused by Internet communication, one must be aware that the policy needs to select a target prior to execution of the handover. We recommend keeping the delay caused by the policy to an absolute minimum. Also because the background dispatching has to wait for an alternative target in case the first target failed. Consequently, during the execution of the policy, it cannot serve pending handovers until control is handed back by the policy.

6.2 Ease of Use

One of the important goals of this project was focussed on the ease of use of the distributed server platform. We first discuss the design of a simple application, namely the proxy implementation presented in section 4.4. We then present a more complex example application taking full advantage of our framework, by using mid-connection handovers. The actual deployment of both distributed servers is then discussed in section 6.3.

6.2.1 Proxy Application

The proxy application allows one to distribute clients without adapting the original server application. Hence, clients can only be distributed at the beginning of their connection. See section 4.3 for a full description on the proxy.

The proxy application itself contains only 200 lines of code, and is listed in appendix A. This is achieved by making use of standard framework functionality. It is configured using command line arguments, and must be started after the unmodified server application is started. At the beginning of execution, the proxy initializes the framework and activates a policy that distributes clients randomly among the available replicas. It then creates a listener socket from which clients can be accepted.

The proxy executes a loop in which it waits for activity using the GeckoSelect class discussed in chapter 4. Upon receipt of a new client, it will open a second connection to the unmodified server application using the GeckoSocket class. Any data received from the client must then be sent to the unmodified server, while data received from the server must be written to the client.

The GeckoSelect class is used to monitor both sockets for read activity. Upon activity, data must be read from the active GeckoSocket object, and subsequently written to the corresponding GeckoSocket. For example, data sent by the server must be read and sent to the client for whom the server-connection was opened. To know to which socket data must be written, the proxy application uses the per-client buffer to let two corresponding GeckoSockets point to each other. Therefore, after data is read the pointer contained in the active GeckoSocket's application buffer is used to locate the corresponding GeckoSocket object for writing.

The application specific structure is used to link the two GeckoSocket objects to each other, and both are added to the read-set of the GeckoSelect class so they can be monitored for activity. Upon activity, data is read from the active socket and consequently written to the corresponding party by following the pointer contained in the active socket's application specific structure.

To allow multiple connections at once, nonbloking connections are used to both the server and client. In case not all data could be written to the corresponding socket at once, the remaining data is written to the application specific structure. The socket is then added to the write-set of the GeckoSelect class to wait for the socket to become ready for write. To guard against the sender overloading the receiver, the sending socket is consequently temporally removed from the read-set, so data is only read if it can be written to the receiver.

Although the proxy application allows one to easily transform a traditional server application into a distributed one, it cannot be used in all circumstances. In case a client is distributed to another replica, all traffic is switched. This means that concurrent connections originating from the same client will be broken upon distribution. Therefore, for applications that may issue multiple concurrent connections, only the contact node should redirect clients, requiring that all other node process incoming connections themselves. This is achieved by using the "–c" argument. Clients can be re-distributed once the binding is removed due to inactivity using the binding's lifetime properties.

6.2.2 Example Application

This section presents an example application, to show how one can easily build a fully transparent distributed server using our framework. The presented example server spawns multiple worker threads that iteratively serve a single client. Each worker thread accepts a client, processes it, and subsequently closes the connection to wait for the next client.

Therefore, the example server can service multiple clients at once. However, the number of simultaneous served clients depends on the number of worker threads. As the number of allowed worker threads is finite, the example application depends on our framework to instantiate multiple replica servers to increase its capacity.

The example application itself is very simple: it reads data from the client, and echoes the data back to the client upon reading the end-of-line character. A client can instruct the server to close the connection by sending a 'Q'. Sending an exclamation mark makes the server echo all data received from the beginning of the connection. Receipt of the 'H' character causes a handover to a target selected by the distribution policy, after which the accepting replica reveals its identity. The full code for this example application is listed in appendix B. In this section, we only discuss the most important routines involving our framework. We do not provide a client application, as one can use a standard, unmodified telnet application to interact with the distributed server.

Initialization

Each replica has to initialize before it can join the distributed server. Our example application uses its main-routine for this:

```
int main(int argc, char *argv[])
{
       if (!getConfig(argc, argv)) return tellUsage();

       // Initialize framework
       GeckoFramework * framework = new GeckoFramework(
              &(config.local_addr),
              config.configfile,
              config.ifname   );

       // Create policy, configure and bind to framework
       GeckoRandomPolicy * policy = new GeckoRandomPolicy(framework);
       if (config.distribute_remote) policy->DoNotDistributeLocally();

       // Join the network
       framework::JoinDistributedServer();

       // Get serversocket
       GeckoServerSocket * serversock = NULL;
       framework->GetServerSocket(&serversock);

       // Start workers
       startWorkers(config.nr_workers, serversock, config.nodename);

       ...
}
```

The example application uses command line arguments to retrieve its configuration. These parameters are verified by the `getConfig` function, which adjusts the `config` structure used for holding the example server's configuration. As a minimum, these parameters are required:

`-c`	`string`	Path to the configuration file for the framework.
`-l`	`string`	Local IPv6 address to use for handovers.
`-i`	`string`	Interface to use for handovers.
`-n`	`string`	Name of the replica, will be revealed to the client application.
`-w`	`int`	Number of workerthreads; the client will be able to process this number of clients at once.

Optionally, one can suppress the echoing of data (-s) or force distribution to peer replicas (-d), so clients are only processed locally if no suitable target could be selected. Although in our example these parameters are specified using the command line, they can also be contained in an application-specific configuration file, which should then be parsed by the application before initialization of the framework.

After retrieving the configuration, the server application creates an instance of the GeckoFramework. Because the example application does not distribute clients by itself, it attaches a distribution policy to the framework. Currently, only a random policy is available which distributes a given client to a random chosen policy. Policies may require additional configuration, which should be done prior to attaching the instantiated policy to the framework using the `GeckoFramework::SetPolicy` method.

Instantiating the framework's GeckoFramework object and assigning a policy is all that is needed to initialize our framework. The replica itself is added to the distributed server using the `GeckoFramework::JoinDistributedServer` method. This function will automatically start the address management protocol, and make the kernel aware of the server's contact address.

Transparent to the application programmer, the (first) contact node will update the binding for the address at the home agent, while others will become standby to take over if necessary. Additionally, all replicas will begin listening for connecting clients, and distribute them according to the policy set. The local server application can begin accepting clients for processing after instantiating the GeckoServerSocket listener class.

Workers

After creation of the listener socket, the application spawns multiple workers, which are responsible for accepting clients and processing them. Each worker executes a single routine that first accepts a client, processes it and re-executes the routine. A client is accepted using:

```
// gain exclusive access to GeckoServerSocket::Accept()
if (pthread_mutex_lock(params::mutex) < 0)
printAbort("Worker failed to pthread_mutex_lock");

// accept a client
params->serversock->Accept(&client);

// release exclusive access to GeckoServerSocket::Accept()
if (pthread_mutex_unlock(params->mutex) < 0)
      printAbort("Worker failed to pthread_mutex_unlock");
```

The `GeckoServerSocket::Accept` method is guarded by a mutex, prohibiting multiple workers to invoke this accept-method simultaneously, as it can result in undefined behavior.

After accepting a client, the server application should determine whether the accepted client is served before or just started a new connection. In the first case, the client will already have written data to the server, which should be remembered because this data must be sent upon receipt of the exclamation mark. Moreover, if a client is handed over because it has written an H in the middle of a line, the acceptor should have knowledge of the data written before the H was read by the donor, as this data needs to be echoed upon reading the end of line character.

To facilitate this application-level handover, the example application uses a structure to memorize and transfer the data received from the client, as well as offsets to determine what should be echoed upon reading the end-of-line character:

```
struct client_state {
       char readbuf[MAXREAD];           /* data sent by the client */
       int  bytesread;                  /* number of bytes received */
       int  start_echo;                 /* number of bytes to skip for echoing most
recent line */
};
```

In case the client was handed over, the `GeckoSocket::GetState` member will return a 1, and change the argument pointer to point to the client's application structure. For new clients it will return a 0, and hence the replica should initialize the client's application structure:

```
// initialize client application state
struct client_state * client_state;

if ( client->GetState((void **)&client_state) == 0) {
       // create and initialize new application state for client
       printf("Worker %d: Client is not served before\n", params->workerid);
       client_state = new struct client_state;
       client_state->bytesread  = 0;
       client_state->start_echo = 0;
       client->SetState(client_state, sizeof(struct client_state));
} else {
       // client is handed off by peer
       printf("Worker %d: Client is handed off\n", params->workerid);
}
```

The application only needs to set the client state once, as the buffer will not be copied. Therefore, modifications made after invoking the `GeckoSocket::SetState` method will be available to the framework upon manually handing over the client.

After determining the state of the client, the worker announces its identity to the client. This is only done to make the handover visible to the user, as communication is continued as normal. The worker-thread then enters a loop in which one byte is read each time, using the `GeckoSocket::Read` method. Each byte is then processed, and depending on the value an action is chosen:

- end-of-line character: the data received since the previous end-of-line character is written to the client using the `GeckoSocket::Write` method, and an '/' is added to the `client_state::readbuf` buffer (no data is echoed if the –s argument is set);
- exclamation mark: all data received since the client connected to the system is echoed;
- capital Q: all data received since the client connected is echoed, and the connection is closed (only a 'Q' is echoed if the –s argument is set);
- capital H: the client is handed over.

Other characters received are added to the client's readbuffer.

Since handovers are fully executed by the framework, one only has to invoke a single method:

```
case 'H':  // Handle 'H' (handoff)
       printf("Worker %d: Nonblocking handing off client..", params::workerid);
       if (client->Handoff(true) != 0) continue;
       client = NULL;

printf("Worker %d: .. nonblocking handoff completed\n", params::workerid);
```

Since the framework is instructed to use the random distribution policy, the application does not have to specify a target explicitly. And because the `nonblocking` argument is set to true, the actual handover will be performed by a background task after consultation of this policy, so the worker can continue serving another client as soon as possible. Note that the application does not delete the client or its application state buffer, because this will be done by the framework (for nonblocking handovers).

In case the handover fails immediately because no target could be selected by the distribution policy, it will continue serving the client. In case the handover of the client fails after the handoff method has returned, it will be handed over to the local replica itself. The client will then be re-accepted by the replica, and treated as if a peer replica handed it off.

The full source is given in appendix B. Below, a possible execution is given for a distributed server consisting of two replicas, named ‘Glass’ and ‘Reich’. Reich is the fixed contact node, but also functions as a replica.

```
[willem@trio]$ telnet 2001:610:110:6c2:2:2:2:2 4000
Trying 2001:610:110:6c2:2:2:2:2...
Connected to 2001:610:110:6c2:2:2:2:2.
Escape character is '^]'.

Hi, I'm worker 1 on Glass and I will continue echoing your data!
Please echo my data
echo >Please echo my data
Some more data
echo >Some more data
!Echo all up to now
Written so far: Please echo my data/Some more data/
echo >!Echo all up to now
I want a Handover
Hi, I'm worker 1 on Reich and I will continue echoing your data!
echo >I want a andover
Another Handover please
Hi, I'm worker 3 on Glass and I will continue echoing your data!
echo >Another andover please
I want more Handovers directly after each other: HiHaHoHeHuHa and echo this
sentence
Hi, I'm worker 3 on Reich and I will continue echoing your data!
Hi, I'm worker 2 on Glass and I will continue echoing your data!
Hi, I'm worker 1 on Reich and I will continue echoing your data!
Hi, I'm worker 4 on Glass and I will continue echoing your data!
Hi, I'm worker 2 on Glass and I will continue echoing your data!
Hi, I'm worker 1 on Glass and I will continue echoing your data!
Hi, I'm worker 4 on Reich and I will continue echoing your data!
echo >I want more andovers directly after each other: iaoeua and echo this sentence
!Echo all up to now
Written so far: Please echo my data/Some more data/Echo all up to now/I want a
andover/Another andover please/I want more andovers directly after each other:
iaoeua and echo this sentence
Echo all up to now
Q
Written so far: Please echo my data/Some more data/Echo all up to now/I want a
andover/Another andover please/I want more andovers directly after each other:
iaoeua and echo this sentence/Echo all up to now
```

A client connecting to the server’s contact address will first be distributed by the framework. Therefore, it may not be served first by the contact node, but by another replica instead.

The worker accepting the client announces its identity once a client is accepted. Note that this is not necessary, as the client application itself is unaware that it is connected to another replica. The telnet application buffers all characters written by the client, and sends the whole line upon return. These sentences are printed in bold in the above example. The example application then reads these characters one at a time, and echoes the sentence upon reading the end of line sequence.

In case a capital H is encountered, the client must be handed over by the worker thread. Therefore, the worker invokes the `GeckoSocket::Handoff` member, after which it can forget about the client and start processing another one. The framework is responsible for transparently handing over the client, by replicating the GeckoSocket object at the acceptor and switching traffic. Note that there may be data waiting to be read in the client's TCP read buffer. This data must be read first at the acceptor, and the framework makes sure this requirement is met by transparently transferring this data.

Upon receipt of an exclamation mark, the worker processing the replica must echo all characters written by the client so far. However, it does not necessary have read this characters by itself. The per-client application buffer is therefore used to communicate this data, so other workers can take over each client and continue communication normally.

Although the example execution of the demo application given above only involves two replicas, it can scale up to hundreds without requiring one to rewrite the application. The complexity of managing such large-scale distributed servers is fully contained within the framework, so the application programmer can focus on the application instead.

Initialization only requires four lines of code: instantiate the GeckoFramework class, instantiate and bind a policy and join the distributed server. Like in a traditional server application, the application has to create a single listener socket to accept clients from. These clients are either processed by several 'worker'-threads akin to the example application, or by a single thread using the GeckoSelect class and non-blocking communication. The latter approach is taken by the proxy application, which resembles the use of the conventional `select` system call.

After accepting a client, the application can communicate with the client as normal, using the Read and Write methods provided by the GeckoSocket class. However, handing over a client in mid-connection is not transparent to the server application, as the donor has to inform the acceptor so it can continue communication without the client noticing. The per-client application buffer must therefore be used to transfer the client's state. The example application uses this buffer to save received data and administer indexes to echo fresh data. Because every replica uses this buffer in the same way, invoking the Handoff method is all that needs to be done to handover a client to a peer replica.

6.3 Performance Analysis

Versatile Anycast itself is lightweight. Due to the use of small-size control messages, the server's scalability is likely to be limited by the computing power of the contact node and home agent, instead of (cheap) available bandwidth.

This section discusses the scalability of servers build using the framework presented in this thesis. The proxy server was used to measure the number of initial, transparent distribution executed by the framework, while the example application was used to measure mid-connection handovers.

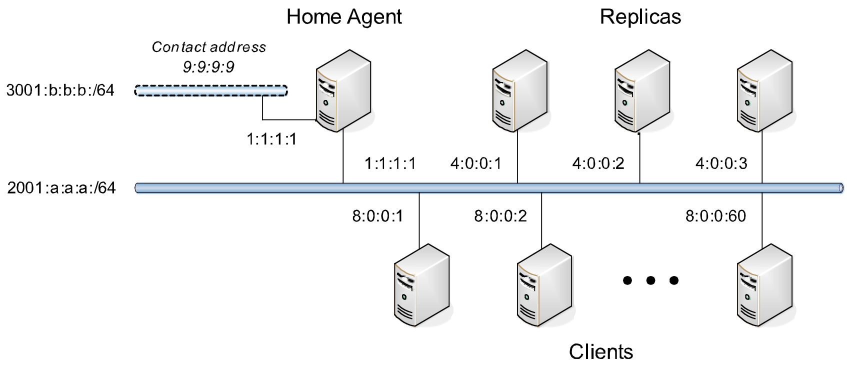

Fig. 11 Handover states and methods of the GeckoSocket class

6.3.1 Test Platform

Both the proxy application and the example distributed server have been deployed on a simple testbed, consisting of two replicas and a single client, and on Grid'5000, which allowed us to run more clients and replicas.

The Grid'5000 project aims at building a highly reconfigurable, controllable and monitorable experimental Grid platform [15]. It does not make assumptions on the middleware or operating system, but instead allows users to deploy their own environments. Currently it is harnessing 9 sites geographically distributed across France, each site featuring between 99 and 342 nodes contained in one or more clusters.

Each site is connected to all other sites using a 10G backbone network. Unfortunately, only IPv4 interconnectivity is currently supported, although most sites allow IPv6 communication between local nodes. Additionally, our dependence on the Linux-2.6.8.1 kernel limits the number of nodes able to run our software, since this dated kernel has no support for most of the recent hardware used within Grid'5000. Despite these limitations, we were able to deploy both applications on two server clusters located at the University of Rennes 1. One cluster consists of 64 Sun nodes, each containing two AMD Opteron 248 2.2Ghz processors, 2Gb memory and equipped with gigabit Ethernet. The other cluster consists of 99 HP nodes, containing two AMD Opteron 246 2.0 Ghz processors, also equipped with 2Gb memory and gigabit Ethernet. All nodes of both clusters are interconnected using a single switch (Cisco 6509).

The tests presented in the next section were performed using a single home agent, 3 replicas and up to 60 clients. All nodes run Ubuntu 5.10 with a patched Linux-2.6.8.1 kernel to allow MIPv6, and support for symmetric multiprocessing. The home agent and clients run the 2.0-rc1 version of the MIPv6 daemon available from [21]. Each replica runs the modified MIPv6 daemon described in section 3.2, and their kernels are additionally patched to enable transport layer handovers using the TCPCP package.

As all nodes are contained in a single broadcast domain, we had to emulate two separate networks. Versatile Anycast requires that the home agent is able to intercept all traffic targeting the contact address, which is thwarted by the central switch. Therefore, the home agent is configured to act as a router. All clients and replicas are configured with an IPv6 address within the 2001:a:a:a/64 prefix. The home agent is also configured with a

3001:b:b:b/64 address, and all nodes use the home agent as their default gateway. Therefore, for any address within the latter prefix, clients have to route their packets to the home agent, as shown in Fig. 11.

The server's contact address is then set to 2003:b:b:b:9:9:9:9. Hence, packets targeting this address can be intercepted by the home agent's MIPv6 implementation and are consequently forwarded to the contact node. Traffic is only sent directly between any replica and client after route optimization has been executed.

6.3.2 Contact Node Scalability

Although the underlying Versatile Anycast mechanisms are very lightweight, they introduce two potential performance bottlenecks. First, each distributed server can have only one contact node where initial client connections are addressed. Second, mid-connection handoffs involve two centralized components, namely the contact node and the home agent. In this section, we evaluate the scalability of distributed servers with respect to initial connection handoff.

All tests were performed using three replica nodes running the proxy application presented in section 4.3 before an unmodified Apache/2.0.54 web server. A total of 60 clients run a shell script that iteratively fetches a static 10kb document from the distributed server using the 'wget' utility. These clients are distributed by the contact node to its two peer replicas using a policy that selects a random target from the replica group view.

Each request consisted of 174 bytes of HTTP request headers while the reply was preceded with a 196 bytes long HTTP header. After the reply was received its contents was checked for integrity after which the MIPv6 daemon was restarted to clear the client's binding cache. By doing so, each new request must be accepted and distributed by the contact node, as if it was a new client.

We then measured the time between initialization of the TCP connection and closing it using the 'tcpdump' utility on the client machine. Additionally, we enabled debug output in our framework to measure the time required to handoff a single client, and gather statistics to determine the number of handovers per second. The 'top' utility was used on the contact node to measure system load. The experiment was run 5 times and the values measured were consistent. Values expressed below are averaged.

Due to problems with clearing the contact node's binding cache, fully discussed in section 6.4, we were only able to execute a fresh request every 6.4 seconds. Therefore, on average the server received 9,4 requests per second. Nonetheless, by starting all 60 clients at the same time we were able to achieve an initial 56 handoffs executed within the same second (not all nodes, because they were not able to start within exactly the same second).

Even during the initial peak, the top utility showed that the percentage of CPU time consumed for both the application and MIPv6 daemon was zero. After running the experiment for 60 seconds, the absolute CPU time used by the application was 0.12 seconds, and for the MIPv6 daemon 0.01 seconds. Within this period, 549 handovers were performed. Assuming a combined CPU time of 0.16, we can roughly estimate that the contact node should be able to distribute (60/0.13) * 549 = 250,000 clients per minute, or 4,200 clients per second.

The Versatile Anycast implementation starts route optimization immediately after a client has connected to the contact address. This causes two messages to be sent and received, all small in size. After selection of a target, a TCP connection is setup to the accepting replica, and the request is written. The size of the serialized TCP endpoint was 96 bytes since no data was sent to the client prior to the handover. However, the client already started sending the request and therefore in 78% of the cases data was already acknowledged before freezing the endpoint, resulting in a 174 bytes buffer that had to be transferred to the acceptor. Additionally, the contact node had to forward both the HoTI and HoT.

Therefore, for every client, 6 messages have to be sent. The bandwidth required for these messages is low as the messages are sent using a single IPv6 packet without any payload, using only an option header. The handover request sent over TCP to the acceptor was also small in size, and consisted of:

- 76 bytes long handover request structure
- 96 bytes serialized TCP endpoint (sendbuffers are empty because no data has been sent to the client yet)
- 174 bytes received data buffer (applicable to 78% of the examined handovers, as discussed above)

Because of this, it is unlikely that the contact nodes capacity is limited due to insufficient bandwidth, provided it is equipped with a modern Internet connection.

The random distribution policy used in the experiment only requests the replica group view from the replica management protocol, and selects a random entry. The amount of computation involved is very low, and as a result the operations required for each client are limited to accepting it, selecting a random destination and sending and receiving nonblocking control messages. More advanced policies may require additional computation, lowering the capacity of the distributed server.

We were not able to examine the effect of communication latency on the performance of the distributed server. In case no replica failure exists, we believe it will not influence the ratio at which clients can be distributed, only the time it takes for a handover to complete. However, in the event of a timeout, the background task responsible for distributing clients needs to re-invoke the policy. Therefore, it will temporally not be able to continue handing over clients, limiting the number of handovers it can finish in time. For this reason, policies should return a target as early as possible.

We also compared the time required by a client to retrieve the document, either directly or through the proxy. In the first case, it takes on average 20.23 ms between instantiating the TCP connection and tearing it down. With our proxy distributing the client, the average time was 23.41 ms, an increase of only 3.18 ms. On average, it took the contact node about 1 ms to extract the socket, transfer it and receive the reply. Within this time, the acceptor also executes the return routability procedure. The difference between this 1 ms and added delay can be explained by the need to retransmit the request in case the contact node did not acknowledge it yet. However, the values measured on the Internet will be several times higher because the time required for a handover to complete depends mostly on the communication latency.

6.3.3 Handover Scalability

Although initial connection handoffs are unlikely to overload a distributed server's contact node, mid-connection handoffs require additional action from the contact node and the home agent. In this section we evaluate the scalability of distributed servers with respect to massive numbers of mid-connection handoffs.

To this end we run the sample application described in section 6.2, each running 20 workers threads. We wrote a separate client application that connects to the distributed server and writes a configurable number of capital ‘H’ characters, each causing a handover, followed by a ‘Q’ to close the connection.

We were able to achieve an average of 250 handovers per second per replica, which was limited by the low number of clients. However, this rate could only be sustained for several seconds, due to issues described in section 6.4. The percentage of CPU resources measured for the MIPv6 daemon was 0.5%, and for the application 4.5%. Combined, the replica required 5% of resources, and interpolating these results to 100% we estimate a replica could be able to execute 5000 handovers per second. This estimates is roughly same as the estimate in section 6.3.2 for the number of clients the contact node can distribute per second. However, as the contact node needs to relay messages for every handover, the actual number of handovers per replica depends on the total number of handovers within the whole distributed server.

Finally, we run the example application on only the contact node and configured it with a number of bogus replica addresses to simulate unresponsive replicas. The handover timeout was set to 2 seconds, after which the contact node selects an alternative target. Because the policy was instructed to prefer remote handovers, it first tried all bogus replicas before finally handing over the client to the local server application, after which it printed its own identity.

The timeout mechanism proved to work well, and a new connection attempt was issued every two seconds. The client application, being a standard telnet implementation, reported that the connection succeeded, but no data was echoed as no replica was serving the client. To simulate the effect of a frozen remote endpoint, we modified the framework to first create the request and connect afterwards. We then measured the time it took for the client application to report a broken connection, which was 21 seconds on average. During this period, the contact node made 11 attempts to handover the client. By modifying the distribution policy, we were able to return a valid replica address at the 10^{th} connection attempt. As a result, the client application received a response in time and printed the replica’s identity. Although the time for the kernel to deem a connection broken depends on the implementation and round trip times to the peer, we can conclude that our framework is able to overcome multiple successive replica failures.

6.4 Code Maturity

The experiments discussed in section 6.3 were the largest deployment of Versatile Anycast to date. They gave new insights, and revealed a number of issues within the two libraries used by our framework to perform handovers.

As the goal of this thesis was to propose an easy to use framework to build distributed servers, most issues found within the underlying libraries are not yet addressed.

Consequently, the implementation of Versatile Anycast is not fully mature, and these issues need to be addressed before one can reliable use distribute servers build using our framework. We discuss the issues encountered here.

6.4.1 IPv6 Address Changes

The Versatile Anycast implementation is based on a MIPv6 daemon running in mobile node operation. The original functionality is retained, while modifications were made to support Versatile Anycast primitives.

The most important consequence is that after execution of the `mipv6_addr_passive` and `mipv6_addr_active` function, the routing table at the host running the replica changes. Both functions add a virtual interface to the system, and update the routing table to prefer this interface over the original interfaces. Traffic sent through this virtual interface is tunneled to the home agent, which then delivers the traffic to the correspondent node. This enables a mobile node to use its fixed home address, even when away from its home network. The daemon can decide to switch to route optimization, removing the need to tunnel traffic through the home agent by revealing the host's care-off address.

Problems arise when the node is not the current contact node, i.e. the node is not registered with the home agent. In such cases, traffic sent to the virtual interface cannot be delivered to its final destination, since the home agent will reject to forward traffic by unauthorized nodes. For that reason, one must explicitly bind the socket to the host's original IPv6 address when initiating communication. The framework currently explicitly requests the operating system to use the replica's care-off address when setting up a TCP connection for handing over a client.

But this requirement breaks the transparency, as other outgoing connections initiated by the server application itself must also be explicitly bound. Additionally, this behavior is not limited to the distributed server application. The address is not used exclusively for the distributed server, but other unsuspecting applications might unwillingly use the address when initiating outgoing connections. Starting the distributed server application may therefore disrupt other applications running on the same host. In our opinion, the Versatile Anycast implementation should be changed in this respect. Instead, one should explicitly bind to the server's contact address. This only requires the framework to bind the application listener socket to this address, since client-sockets accepted from this socket will already use the right address. Moreover, normal operation is retained for both the server application as well as other processes.

6.4.2 Versatile Anycast Errors

The framework uses a proof-of-concept Versatile Anycast implementation built upon an unstable MIPv6 implementation, that is not designed to be used on a large scale.

One serious issue is that `mip6_handoff_start` does not return in case an error is encountered. The daemon continues to retry the route optimization procedure with the client, causing the background handover thread to block indefinitely in case the client has crashed. The desired behavior would be that a maximum number of retries are made, after which the function returns an error code, indicating what went wrong. The distributed server framework can then use this information to decide upon an action.

This problem also occurs after the contact node has changed but the replica has not updated its daemon yet. Consequently, this issue has to be solved before addressing the issue of contact node management, otherwise one cannot use or test contact node failure due to hanging functions.

Information about failures can also be used to make the framework more reliable. For example in case the handover request is written to the acceptor, but no acknowledgement is received within a specified timeout interval. The framework currently closes such sockets, potentially dropping clients in mid-connection. An alternative is to retry the handover to another replica. The new acceptor the tries to take over the client using the same sequence number as the previous acceptor. In case the sequence number is accepted, the client was not yet taken over by the old acceptor. In case the client rejects the sequence number, the acceptor should report this to the donor, which then has to assume the first acceptor toke over the client (and potentially changed the TCP endpoint).

Next to not returning upon error, `mip6_handoff_start` sometimes does not return with no apparent reason. The modified daemon is controlled using UDP control datagrams, which are sent using the Versatile Anycast functions described in section 3.2.3. After such a message is sent, the function blocks on receiving a message back from the daemon, indicating success. Sometimes, no such message is sent back to the application, causing the function to block indefinitely. This behavior made it very difficult to run experiments with many handovers, as each handover could potentially disable a replica.

6.4.3 Binding Cache

Explicitly clearing the binding at the client using the `mip_handoff_clear` function does not work as expected. Only the binding at the client is cleared, but the binding at the replica itself seems to be remained intact. The same behavior is observed after handing over a client; the donor's daemon does not remove state about the client.

The Versatile Anycast API provides a function to clear any unused data structures after handoff (`mip6_handoff_finish`), but executing this function often results in segmentation faults, causing the daemon to crash and sometimes even making it necessary to even restart the operating system.

Not removing these binding entries at the contact nodes cause subsequent connections to stall for 5.5 seconds on average. Normally, a client sends a SYN packet directly to the contact address to setup a new TCP connection. This packet is tunneled to the contact node, which first completes the connection before switching to route optimization mode.

However, if the contact node still has a binding, it sends its own SYN packet directly to the client, as like route optimization was already executed. This is against the specification, and the client responds with a binding error control message and drops the packet. After 5.5 seconds on average, the contact node's MIPv6 daemon suddenly starts the route optimization, resulting in a valid binding at the client. After the binding is installed, subsequent SYN packets received can be translated correctly, and delivered to the transport layer.

Additionally, `mip_handoff_clear` only tries to clear the client's binding once. Instead of waiting for a binding acknowledgement, the function returns immediately. As a result, execution of this function does not guarantee the binding is cleared, as messages can get lost.

Apart from these issues, some smaller problems were encountered. Sometimes, `mip6_handover_mark` causes the daemon to crash with a segmentation error, although we were not able to determine the exact cause. The `mip6_address_active` function is responsible for registering the contact node with the home agent, but it does return even when the registration fails, for example because the home agent is not running. Ideally, it should check if the binding succeeded to avoid situations in which the system is not reachable on its contact address because no node has an active registration. Finally, at several occasions no data could be sent or received using the contact address at a replica because the virtual interface described at the beginning of this section disappeared without a reason.

6.4.4 TCPCP Issues

In addition to problems with the Versatile Anycast library, we encountered some issues while using the TCPCP library, which is also a proof-of-concept implementation.

TCPCP provides two functions to extract sockets: one setting a pointer to a buffer holding the serialized endpoint and one function that returns the size of this buffer. For serialized TCP endpoint with empty buffers, this size is always 96 bytes.

This serialized endpoint is then sent to the acceptor, which may have to correct an index indicating on which network interface the socket is bound. This index is altered by projecting a structure over the buffer holding the endpoint. But this structure is 100 bytes in size, with the index member located at the end. Therefore, adjusting this index results into writing in an invalid memory location. This error was discovered during testing the framework on Grid'5000, as the program crashed due to a segmentation error. The backtrace given by 'gdb' indicated that the program had crashed upon `malloc`, while allocating memory for receiving the serialized TCPCP of another client. This output was consistent throughout multiple tests, but the real cause (writing in invalid location) was ultimately determined using 'valgrind', a suite of tools for debugging and profiling programs.

Another problem was encountered when activating sockets. After extraction of a socket, it is still in the kernel although it is frozen so incoming packets can be recognized and silently discarded. Despite this, `tcpcp_activate` does not unfreeze the socket at the donor, preventing the framework from re-using it. This functionality is important, because sometimes a handover to a peer may fail after extracting the socket. A policy may then decide to handover the client locally, but the framework currently has no means to re-insert or unfreeze the socket, as discussed in section 5.3.

Another issue was encountered when inserting an endpoint at the acceptor. Occasionally, `tcpcp_get` causes a segmentation error. However, running the replica under the 'gdb' utility did not replicate this behavior. Additionally, `tcpcp_create` sometimes returns without inserting the endpoint or indicating an error, so subsequent traffic sent by the client is countered with an RESET message.

6.4.5 Cluster Issues

Besides the issue of having only a single broadcast domain, as explained in section 6.3.1, deploying the application on Grid'5000 was not straightforward. This is mostly due to the dependence on the Linux-2.6.8.1 kernel.

Although one may decide to install an operating system from scratch, it is very difficult to do, if only because of the restricted Internet connectivity. Users are therefore encouraged to select one of the available existing deployment images, and customize it according to their needs. However, those existing deployment images are not suitable for running the Linux-2.6.8.1 kernel, because their environment assumed a modern kernel supporting the AMD64 processor. Fortunately, we were able to compile and run the Linux-2.6.8.1 kernel using a dated Ubuntu Linux 5.10 image for i386 capable processors.

An issue we were not able to solve is that resolving an IPv6 to a MAC address initially takes one second. By examining the output of the 'tcpdump' utility we were able to conclude that the system waits for one second before executing the neighbor discovery protocol, the IPv6 equivalent of IPv4 ARP. Therefore, starting communication with another replica takes at least one second for the first time. This initially made handovers impossible since the handover timeout was configured to be one second, after which the target replica was deemed unavailable. We worked around this problem by first ping-ing all potential communication partners before starting the experiment; hence neighbor discovery is done in advance. However, this problem did not manifest itself in our local, but smaller, server cluster.

Chapter 7

Conclusion

The use of a single server in client-server applications imposes limitations in sustainable load and availability guarantees. Within a distributed server, clients are distributed over multiple servers, preferably without support from the client application itself. The server should have fine-grain control over the distribution, and be resilient against failures. Traditional client distribution mechanisms fail to satisfy these requirements.

Versatile Anycast is an alternative mechanism that allows anycast-like functionality. The current implementation consists of two sets of functions: one set to control the binding of the server's contact address to a contact node, and a second set to switch traffic from a single client to selected replica, even in mid-connection. Although it provides the basic functionality to build distributed servers, doing so involves monitoring the contact node and electing a replacement when required. Additionally, all nodes able to handover clients must have knowledge over at least some other node, and each handover must be carefully orchestrated to avoid connection resets. It is therefore better to confine this functionality in single reusable package.

This thesis discussed the requirements of such a package, and presented a framework that allows one to build transparent distributed servers using Versatile Anycast to distribute clients. By abstracting the common parts of distributed servers, namely membership management and client distribution, the framework reduces the complexity of writing such servers.

Because different applications have different needs, we designed the framework to be versatile in that it can support a wide variety of uses. Some applications may only require random distribution of clients; others require clients to be distributed to a nearby node. Therefore, the framework separates the distribution policy from the mechanism, offering flexibility by allowing end-users to select an redirection policy that fits their needs best. Besides distributing clients upon connection, they can also be handed off in mid-connection, allowing for a wide range of new applications.

The API to the framework follows the same methodology of traditional, single-node servers, making it easier to understand and use. More advanced functionality like mid-connection handovers is offered in such a way that it is comparatively easy to use. The current implementation of the framework assumes that all nodes are reliable, but it is designed in such a way that it can be easily extended to support dynamic membership management and management of the contact address.

This thesis also presents a proxy application built using the framework, which allows one to distribute clients without adapting the original server application. We also gave a short example of a distributed server making use of mid-connection handovers. Both applications have been successfully deployed on Grid'5000, an experimental grid platform. Experiments show that our framework is able to coop with high client arrival and handover rates, while handovers can be executed within a short period of time, depending on the network latencies between the nodes involved.

Future improvements

The first problem that needs to be addressed is the dependence on the dated Linux-2.6.8.1 kernel by either porting the modified MIPv6 daemon to newer platforms or switching to a more recent project that adds MIPv6 functionality to the Linux platform. All issues discussed in section 6.4 must be addressed; the Versatile Anycast implementation must return on errors and its stability and usability should be improved. Additionally, IPsec must be used to protect the binding at the home agent, to guard against hostile takeovers.

Switching to a more recent kernel also involves porting the TCPCP package to allow TCP transport layer handoffs. One potential improvement on TCPCP would be to make the handover more transparent, by also moving options set on the socket, and the received data buffer.

Next to addressing the issues found in the libraries used by our framework (section 6.4), the framework itself should be improved, as its current implementation is not suitable for wide-area deployment. One important aspect currently not addressed by the framework is security. Communication between replicas is currently unsecured, allowing a wide range of attacks, such as Denial of Service, by issuing fake handover requests.

To support large-scale distributed servers, the framework must be extended to support dynamic membership, allowing one to add and retract replicas to or from the system without restarting it. To guard against unavailability of the server, the contact node should be monitored and backup contact nodes must decide upon a replacement if necessary.

Although the framework is designed to include a set of ready-implemented policies to choose from, only random distribution is currently available. More policies should be defined and implemented. Additionally, one should investigate the requirements of more advanced policies, and extend the framework to support common requirements such as information dissemination.

Applications requiring multiple simultaneous connections to the service are currently not supported, as the handover of one causes the other connections to break. This can be solved by allowing multiple contact addresses, requiring adaptations to the current framework and likely the Versatile Anycast implementation.

Finally, only TCP connections are supported by the framework. Although the use of UDP results a different methodology for writing server applications, the framework could be extended to support datagram based services as well.

We believe the framework allows one to build transparent distributed servers without additional complexity over traditional, single-node servers. Its implementation should mature over time, making it easier to build distributed servers and provide services to large numbers of clients.

Bibliography

1. W. Almesberger, *Tcp connection passing*, in *Ottawa Linux Symposium*. 2004: Ottowa, United States.
2. J. Arkko, V. Devarapalli, and F. Dupont, *Using ipsec to protect mobile ipv6 signaling between mobile nodes and home agents*, in *RFC 3776*. 2004.
3. V. Cardellini, E. Casalicchio, and M. Colajanni, *The state of the art in locally distributed web-server systems* ACM Computing Surveys, 2002. **34**(2): p. 263-311.
4. M. Colajanni, P.S. Yu, and D.M. Dias, *Analysis of task assignment policies in scalable distributed web-server systems.* IEEE Transactions on Parallel and Distributed Systems, 1998. **9**(6): p. 585-597.
5. J. Dilley, et al., *Globally distributed content delivery.* IEEE Internet Computing, 2002. **6**(5): p. 50-58.
6. M. Freedman, K. Lakshminarayanan, and D. Mazières. *Oasis: Anycast for any service.* in *Proc. of the Symposion on Networked Systems Design and Implementations*. 2006. San Jose, United States.
7. D. Johnson, C. Perkins, and J. Arkko, *Mobility support in ipv6*, in *RFC 3775*. 2004.
8. M. Szymaniak, et al., *A single-homed ad hoc distributed server*, in *Technical Report IR-CS-013*. 2005, Vrije Universiteit: Amsterdam, the Netherlands.
9. M. Szymaniak, et al., *Enabling service adaptability with versatile anycast.* Concurrency and Computation: Practice and Experience, 2006. **13**(12): p. 1837-1863.
10. M. Szymaniak, G. Pierre, and M. van Steen. *Netairt: A dns-based redirection system for apache.* in *Proc. of the IADIS International*. 2003. Algarve, Portugal.
11. M. Szymaniak, G. Pierre, and M. van Steen. *Versatile anycasting with mobile ipv6.* in *Proc. of the 2nd International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications*. 2006. Pisa, Italy: ACM Press.
12. S. Voulgaris, D. Davidia, and M. van Steen, *Cyclon: Inexpensive membership management for unstructured p2p overlays* Journal of Network and Systems Management, 2005. **13**(2): p. 197-217.
13. Web: *Tcpcp library.* 2005 - accessed July 2007; Location: http://tcpcp.sourceforge.net.
14. Web: *Fmipv6.* 2007 - accessed November 2007; Location: http://www.fmipv6.org/.
15. Web: *Grid5000.* 2007 - accessed Januari 2008; Location: https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home.
16. Web: *Implementing mobile ipv6.* 2007 - accessed November 2007; Location: http://www.cisco.com/univercd/cc/td/doc/product/software/ios123/123cgcr/ipv6_c/sa_mobv6.htm.
17. Web: *Ist-anemone project.* 2007 - accessed 2007 januari 2007 januari; Location: http://www.ist-anemone.eu/index.php/Home_Page.
18. Web: *Tcpcp2.* 2007 - accessed November 2007; Location: http://tcpcp2.sourceforge.net/.
19. Web: *The cable guy – september 2004: Introduction to mobile ipv6.* 2004 - accessed July 2007; Location: http://www.microsoft.com/technet/community/columns/cableguy/cg0904.mspx.

20. Web: *Survey of mobile ipv6 implementations for freebsd and linux*. 2006 - accessed July 2007; Location: http://www.eurescom.de/~public-web-deliverables/P1100-series/P1113/D1/41_mip.html.
21. Web: *Mobile ipv6 for linux*. 2007 - accessed July 2007; Location: http://www.mobile-ipv6.org/.

Appendix

A Proxy Application

Discussed in Chapter 5.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <ctype.h>

#include <netdb.h>
#include <ifaddrs.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

// Include the GeckoFramework
#include "libgecko/GeckoFramework.h"
#include "libgecko/policies/GeckoRandomPolicy.h"
#include "libgecko/GeckoSelect.h"

extern int errno;

#define READ_BUFFER        1024

#define NIP6ADDR(addr) \
        ntohs((addr)->s6_addr16[0]), \
        ntohs((addr)->s6_addr16[1]), \
        ntohs((addr)->s6_addr16[2]), \
        ntohs((addr)->s6_addr16[3]), \
        ntohs((addr)->s6_addr16[4]), \
        ntohs((addr)->s6_addr16[5]), \
        ntohs((addr)->s6_addr16[6]), \
        ntohs((addr)->s6_addr16[7])


GeckoFramework * geckoframework = NULL;
GeckoServerSocket * geckoserversocket = NULL;
pthread_mutex_t worker_sync_mutex;

// Configuration
struct config_s {
    struct    in6_addr local_addr;      // Local address to use
    char * ifname;                      // Interface to use
    char * geckoconf;                   // Path to configuration file for GeckoLib

    struct in6_addr application_addr;   // IP address of application server host
    int application_port;               // Port number of (proxied) application server
```

```
    int nr_workers;                     // Number of worker threads
    bool verbose;                       // Print verbose output
    char local_name[128];               // Local hostname
} config;


void perror_exit(char * msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}


void tellusage()
{
    fprintf(stderr, "Usage:\n");
    fprintf(stderr, " Compulsory:\n");
    fprintf(stderr, "   -g <string>           Path to the GeckoFramework configuration file\n");
    fprintf(stderr, "   -a <integer>          Port number of local application server\n");
    fprintf(stderr, " Optional:\n");
    fprintf(stderr, "   -l <IPv6 address>     Local address to use (default: first encountered
                                          IPv6 address)\n");
    fprintf(stderr, "   -i <string>           Interface to use for clients (must be able to bind
                                          local address to it  (default: eth0)\n");
    fprintf(stderr, "   -s <IPv6 address | hostname>  Address of application server
                                          (default: localhost)\n");
    exit(EXIT_FAILURE);
}


int getconfig(int argc, char *argv[])
{
    int i;
    bool g_set, a_set, l_set;

    // default config
    config.local_addr = in6addr_loopback;
    config.ifname = "eth0";
    config.application_addr = in6addr_loopback;
    config.nr_workers = 5;
    config.verbose = false;

    // get specifics from argv
    g_set = a_set = l_set = false;
    struct hostent *hp;

    for (i=1 ; i<argc ; i++) {
        if ( (argv[i][0] == '-') && (argv[i][1] != '\0') ) {

            switch(argv[i][1]) {
                case 'g':    // -g : config.geckoconf
                    if (i>=argc) return -1;
                    config.geckoconf = argv[++i];
                    g_set = true;
                    break;

                case 'a':    // -a config.application_port
                    if (i>=argc) return -1;
                    if ( (config.application_port = strtol(argv[++i], NULL, 10)) < 1)
```

```
                perror_exit("Invalid argument for application port (-f)\n");
            if (config.application_port < 1)
                perror_exit("Invalid application port number (-f)");
            a_set = true;
            break;

        case 'l':    // -l : config.local_addr
            if (i>=argc) return -1;
            if (!inet_pton(AF_INET6, argv[++i], &(config.local_addr)) )
                perror_exit("Invalid IPv6 address given as local address (-l)");
            l_set = true;
            break;

        case 'i':    // -i : config.ifname
            if (i>=argc) return -1;
            config.ifname = argv[++i];
            break;

        case 's':    // -i : config.application_host
            if (i>=argc) return -1;
            hp = gethostbyname2(argv[++i], AF_INET6);
            if (hp == NULL || hp->h_addrtype != AF_INET6)
                perror_exit("No IPv6 addresses available for given appication hostname (-s)");
            config.application_addr = *((struct in6_addr*)hp->h_addr_list[0]);
            break;

        case 'n':    // -n : config.nr_workers
            if (i>=argc) return -1;
            if ( (config.nr_workers = strtol(argv[++i], NULL, 10)) < 1)
                perror_exit("Invalid argument for number of workers (-n)\n");
            if (config.nr_workers < 1) perror_exit("Invalid number of workers (-n)\n");
            break;

        case 'v':    // -n : verbose mode
            config.verbose = true;
            break;

        default:
            return -1;
            break;
    }

    } else return -1;    // malformed argument (no -x)
}

if (!l_set) {
    // take first IPv6 address found for this host
    struct ifaddrs *ifap, *ifa;
    struct sockaddr_in6 *sa;
    getifaddrs(&ifap);
    for (ifa=ifap; ifa; ifa = ifa->ifa_next) {
        if(ifa->ifa_name == NULL) continue;                    // on the right interface
        if(strcmp(ifa->ifa_name, config.ifname) != 0) continue;

        if (ifa->ifa_addr->sa_family != AF_INET6) continue;    // only interested in ipv6
        sa = (struct sockaddr_in6 *) ifa->ifa_addr;

        if (sa->sin6_family != AF_INET6) continue;             // we need ipv6
        if (ifa->ifa_addr == NULL)    continue;
```

```
            config.local_addr = (struct in6_addr) sa->sin6_addr;
        }
    }

    if (! (g_set && a_set)) return -1;

    return 0;
}


void tellconfig()
{
    printf("Initializing GeckoProxy server\n");
    printf(" GeckoConfig:\n");
    printf("    %s\n", config.geckoconf);
    printf(" Listening:\n");
    printf("    address=%x:%x:%x:%x:%x:%x:%x:%x, ifname=%s\n", NIP6ADDR(&config.local_addr),
          config.ifname );
    printf(" Application server:\n");
    printf("    address=%x:%x:%x:%x:%x:%x:%x:%x, port=%d\n", NIP6ADDR(&config.application_addr),
         config.application_port );
    printf("\n");
}


struct geckoproxy_sockstate {
    GeckoSocket * partner;
    char to_write[READ_BUFFER];
    int to_write_size;
};


int main(int argc, char *argv[])
{
    if (getconfig(argc, argv) <0) {
        tellusage();
        return 0;
    }
    tellconfig();

    // create framework / server
    geckoframework = new GeckoFramework(&(config.local_addr), config.geckoconf, config.ifname);
    geckoframework->GetServerSocket(&geckoserversocket);

    // create and bind policy
    GeckoRandomPolicy * policy = new GeckoRandomPolicy(geckoframework);
    policy->OnlyContactNodeDistributes();
    geckoframework->SetPolicy(policy);

    // we're using a single thread to serve all clients using the GeckoSelect
    GeckoSelect * geckoselect = new GeckoSelect(geckoframework);
    GeckoSocketList * read_list;
    GeckoSocketList * write_list;
    int retc;

    while (1) {
        read_list = new GeckoSocketList();
        write_list = new GeckoSocketList();
        GeckoSocketList::iterator itr;
```

```
if (config.verbose) printf("Select..");
retc = geckoselect->Select(read_list, write_list, true, NULL);
if (retc < 0) perror_exit("Error selecting\n");
if (config.verbose) printf(".. returned\n");

// Retc = 1 means that there is a new client waiting to get accepted
if (retc == 1) {
    // accept new client
    GeckoSocket * clientconn = NULL;
    geckoserversocket->Accept(&clientconn);

    struct sockaddr_in6 csin;
    clientconn->GetPeerName(&csin);
    printf("New client accepted: %x:%x:%x:%x:%x:%x:%x:%x\n", NIP6ADDR(&csin.sin6_addr));

    // Create connection to application server (high-performance proxy should
    // do this nonblocking ;)
    GeckoSocket * serverconn = new GeckoSocket(geckoframework);
    serverconn->Connect(&(config.application_addr), config.application_port);

    // Make both nonblocking
    serverconn->SetNonBlocking();
    clientconn->SetNonBlocking();

    // Each GeckoSocket can hold a  pointer to a application-specific
    // structure. We will use this to specify to which GeckoSocket read data must
    // be written
    struct geckoproxy_sockstate * clientState = new struct geckoproxy_sockstate;
    clientState->partner = serverconn;
    clientconn->SetState(clientState, sizeof(struct geckoproxy_sockstate));

    struct geckoproxy_sockstate * serverState  = new struct geckoproxy_sockstate;
    serverState->partner = clientconn;
    serverconn->SetState(serverState, sizeof(struct geckoproxy_sockstate));

    // Add both GeckoSockets to the geckoselect set for read activity
    if (geckoselect->MonitorSocketForRead(serverconn) <0) return -1;
    if (geckoselect->MonitorSocketForRead(clientconn) <0) return -1;
}

// Read list: connection endpoints that want to write data to the other side
for (itr=read_list->begin() ; itr!=read_list->end() ; itr++) {
    GeckoSocket * from;
    GeckoSocket * to;
    struct geckoproxy_sockstate * from_state;
    struct geckoproxy_sockstate * to_state;
    struct sockaddr_in6 tosin, fromsin;

    from = (GeckoSocket *) (*itr);
    from->GetState((void **)&from_state);

    to = from_state->partner;
    to->GetState((void **)&to_state);

    to->GetPeerName(&tosin);
    from->GetPeerName(&fromsin);

    char buff[READ_BUFFER];
```

```
    ssize_t read_size;
    read_size = from->Read(&buff, READ_BUFFER);

    if (read_size == 0) {
        // Connection closed
        close(to->socket.sd);
        close(from->socket.sd);
        geckoselect->StopMonitoringSocketForRead(to);
        geckoselect->StopMonitoringSocketForRead(from);

        printf("Node %x:%x:%x:%x:%x:%x:%x:%x closed connnection to %x:%x:%x:%x:%x:%x:%x:%x\n",
              NIP6ADDR(&fromsin.sin6_addr), NIP6ADDR(&tosin.sin6_addr));
    }
    else {
        // Data ready to be written..
        ssize_t write_size;
        if (config.verbose) printf("Writing data from %x:%x:%x:%x:%x:%x:%x:%x to
              %x:%x:%x:%x:%x:%x:%x:%x\n", NIP6ADDR(&fromsin.sin6_addr),
              NIP6ADDR(&tosin.sin6_addr));

        write_size = to->Write(&buff, read_size);

        // All data written?
        if (write_size < read_size) {
            // stop monitoring for read: first empty write buffer
            if (geckoselect->StopMonitoringSocketForRead(from) <0) return -1;

            if (config.verbose) printf("Not all data could be written, saving %d
                (read) - %d (written) bytes for later\n", read_size, write_size);

            // add to buffer
            memcpy(to_state->to_write, (buff+write_size), (read_size - write_size));
            to_state->to_write_size = (read_size - write_size);

            // start monitoring for write
            if (geckoselect->MonitorSocketForWrite(to) <0) return -1;
        }
    }

    // end of processing read-ready socket
}

// Write list: connection endpoints for which data is buffered by proxy
for (itr=write_list->begin() ; itr!=write_list->end() ; itr++) {
    GeckoSocket * from;
    GeckoSocket * to;
    struct geckoproxy_sockstate * from_state;
    struct geckoproxy_sockstate * to_state;
    struct sockaddr_in6 tosin, fromsin;

    to = (GeckoSocket *) (*itr);
    to->GetState((void **)&to_state);

    from = to_state->partner;
    from->GetState((void **)&from_state);

    to->GetPeerName(&tosin);
    from->GetPeerName(&fromsin);
    if (config.verbose) printf("Writing saved data from %x:%x:%x:%x:%x:%x:%x:%x
```

```
                to %x:%x:%x:%x:%x:%x:%x:%x\n",
                NIP6ADDR(&fromsin.sin6_addr), NIP6ADDR(&tosin.sin6_addr));

            // write buffer
            ssize_t write_size;
            write_size = to->Write(&(to_state->to_write), to_state->to_write_size);

            // partial buffer? -> not yet supported, drop data :)
            if (write_size < to_state->to_write_size) {
                // todo
                printf("Partial write not implemented (written %d of %d)", write_size,
                     to_state->to_write_size);
            } else
                if (config.verbose) printf("All data written - continueing monitoring
                     corresponding read-socket");

            // stop monitoring for write, start waiting for data from other side
            geckoselect->MonitorSocketForRead(from);
            geckoselect->StopMonitoringSocketForWrite(to);
        }

        // Next select loop...
    }

    return 0;
}
```

B Example Application

Discussed in Chapter 5.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "libgecko/GeckoFramework.h"
#include "libgecko/policies/GeckoRandomPolicy.h"

#define MAXREAD       1024

/* Configuraton struct    */
struct config_s {
    struct    in6_addr local_addr;
    char * configfile;
    char * nodename;
    char * ifname;
    bool   silent;
    bool   distribute_remote;
    int    nr_workers;
} config;


/* Application specific client-state structure    */
struct client_state {
    char readbuf[MAXREAD];        /* data sent by the client */
    int  bytesread;               /* number of bytes received */
    int  start_echo;              /* number of bytes to skip for echoing most recent line */
};


/* Structure given to each worker */
struct worker_params {
    GeckoServerSocket * serversock;    /* pointer to the GeckoServerSocket */
    int  workerid;                     /* id of the worker for printing purposes */
    pthread_mutex_t * mutex;           /* mutex for exclusive access to GeckoServerSocket->Accept() */
    char * nodename;                   /* name of the machine */
};



/**
*  Print message and abort
*/
void printAbort(char * message)
{
    fprintf(stderr, "Program abort: %s", message);
```

```
    abort();
}


/**
*  Print help on command line arguments
*/
int tellUsage() {
    fprintf(stderr, "Usage:\n");
    fprintf(stderr, " Compulsory:\n");
    fprintf(stderr, "   -c     Configurationfile to use for VA framework\n");
    fprintf(stderr, "   -l     Local address to use as care-off address\n");
    fprintf(stderr, "   -i     Interface to use for handovers \n");
    fprintf(stderr, "          (care-off address must be bound to it)\n");
    fprintf(stderr, "   -n     Identifying name of the replica\n");
    fprintf(stderr, "   -w     Number of worker threads\n");
    fprintf(stderr, " Optional:\n");
    fprintf(stderr, "   -s     Silent: only echo 'Q' upon close (test) \n");
    fprintf(stderr, "   -d     Force distribution to peer replica\n");
    fprintf(stderr, "Use ^c to quit server\n");
    return 1;
}


/**
*  Parse command line arguments, check if all configuration is set
*/
bool getConfig(int argc, char *argv[])
{
    int i;
    bool l_set = false, c_set = false, i_set = false, n_set = false, w_set = false;

    config.nodename = "Noname";
    config.silent = false;
    config.distribute_remote = false;

    // get params from argv
    for (i=1 ; i<argc ; i++) {
        if ( (argv[i][0] == '-') && (argv[i][1] != '\0') ) {
            switch(argv[i][1]) {
                case 'c':    /* -c   Configurationfile to use for VA framework */
                    if (i>=argc) return false;
                    config.configfile = argv[++i];
                    c_set = true;
                    break;

                case 'l':    /* -l   Local address to use as care-off address */
                    if (i>=argc) return false;
                    if (!inet_pton(AF_INET6, argv[++i], &(config.local_addr)) )
                        printAbort("Invalid IPv6 address given as local address (-l)");
                    l_set = true;
                    break;

                case 'i':    /* -i   Interface to use for handovers */
                    if (i>=argc) return false;
                    config.ifname = argv[++i];
                    i_set = true;
                    break;
```

```
                case 'n':    /* -n   Identifying name of the replica */
                    if (i>=argc) return false;
                    config.nodename = argv[++i];
                    n_set = true;
                    break;

                case 'w':   /* -w   Number of worker threads */
                    if (i>=argc) return false;
                    if ( (config.nr_workers = strtol(argv[++i], NULL, 10)) < 1)
                        printAbort("Invalid argument for number of workers\n");
                    w_set = true;
                    break;

                case 's':   /* -s   Silent: only echo 'Q' upon close (test) */
                    config.silent = true;
                    break;

                case 'd':  /* -d    Force distribution to peer replica */
                    config.distribute_remote = true;
                    break;

                default : return false;
            }
        } else return false;
    }
    if (! (l_set && c_set && i_set && n_set && w_set)) return false;

    return true;
}


/**
*  Function executed by every worker thread
*/
void * workerThread(void * params_vp)
{
    struct worker_params * params = (struct worker_params *) params_vp;

    /* Try to accept and process a client */
    GeckoSocket * client = NULL;
    struct sockaddr_in6 client_sin;

    while(true) {
        // gain exclusive access to GeckoServerSocket->Accept()
        if (pthread_mutex_lock(params->mutex) < 0)
            printAbort("Worker failed to pthread_mutex_lock");

        // accept a client
        params->serversock->Accept(&client);

        // release exclusive access to GeckoServerSocket->Accept()
        if (pthread_mutex_unlock(params->mutex) < 0)
            printAbort("Worker failed to pthread_mutex_unlock");

        // print client's IPv6 address
        client->GetPeerName(&client_sin);
        printf("Worker %d: Accepted client %x:%x:%x:%x:%x:%x:%x:%x\n",
                params->workerid,
                ntohs(client_sin.sin6_addr.s6_addr16[0]), ntohs(client_sin.sin6_addr.s6_addr16[1]),
```

```
        ntohs(client_sin.sin6_addr.s6_addr16[2]), ntohs(client_sin.sin6_addr.s6_addr16[3]),
        ntohs(client_sin.sin6_addr.s6_addr16[4]), ntohs(client_sin.sin6_addr.s6_addr16[5]),
        ntohs(client_sin.sin6_addr.s6_addr16[6]), ntohs(client_sin.sin6_addr.s6_addr16[7]));

// initialize client application state
struct client_state * client_state;

if ( client->GetState((void **)&client_state) == 0) {
    // create and initialize new application state for client
    printf("Worker %d: Client is not served before\n", params->workerid);
    client_state = new struct client_state;
    memset(client_state, 0, sizeof(struct client_state));
    client_state->bytesread  = 0;
    client_state->start_echo = 0;
    client->SetState(client_state, sizeof(struct client_state));

} else {
    // client is handed off by peer
    printf("Worker %d: Client is handed off\n", params->workerid);
}

// announce identity (for demonstration purposes)
if (!config.silent) {
    char announcement[MAXREAD];
    sprintf(announcement, "\nHi, I'm worker %d on %s and I will continue echoing your data!\n",
            params->workerid, params->nodename);

        client->Write(announcement, strlen(announcement) );
}

// read one byte at a time:
// - print line upon enter
// - print all upon !
// - handoff upon H
// - close upon Q
bool client_finished = false;
while ( client->Read( &(client_state->readbuf[client_state->bytesread]), 1) == 1 ) {

    switch(client_state->readbuf[client_state->bytesread]) {

        case '\n':    // print line
            if (!config.silent) {
                printf("Worker %d: Received EOL, echoing data\n", params->workerid);
                client_state->readbuf[client_state->bytesread+1] = '\0';
                client->Write("\necho > ", strlen("\necho >") );
                client->Write(&(client_state->readbuf[client_state->start_echo + 1]),
                        strlen(&(client_state->readbuf[client_state->start_echo])) );
                client->Write("\n", 1 );

                client_state->readbuf[client_state->bytesread] = '/';
                client_state->start_echo = client_state->bytesread;
            }
            break;

        case '\r':
            client_state->bytesread--; // ignore
            break;

        case '!':  // print all
```

```
                printf("Worker %d: Printing all data\n", params->workerid);
                client->Write("\nWritten so far: ", strlen("\nWritten so far: ") );
                client->Write(client_state->readbuf, client_state->bytesread );
                client->Write("\n", 1 );
                break;

            case 'H':  // Handle 'H' (handoff)
                printf("Worker %d: Nonblocking handing off client..\n", params->workerid);
                client->Handoff(true);
                client = NULL;

                printf("Worker %d: .. nonblocking handoff completed\n", params->workerid);
                client_finished = true;
                break;

            case 'Q':  // Handle 'Q' (quit)
                printf("Worker %d: Client connection closed upon request\n", params->workerid);
                if (!config.silent) {
                    client->Write(client_state->readbuf, client_state->bytesread );
                    client->Write("\n\n", 2 );
                } else {
                    client->Write("Q", 1);
                }
                client->Close();
                client_finished = true;
                delete client;
                break;
        }

        if (client_finished) break;
        else client_state->bytesread++; // next byte
    }

    if (!client_finished) {
        // error reading data
        printf("Worker %d: Error reading data, closing\n", params->workerid);
        client->Close();
        delete client;
    } else {
        // client already deleted or done by framework
    }
}
return NULL;
}


/**
*  Start number of workers to accept and process clients
*/
void startWorkers(int number, GeckoServerSocket * serversock, char * nodename)
{
    // initialize a mutex for exclusive access to GeckoServerSocket->Accept()
    pthread_mutex_t     * sync_mutex = new pthread_mutex_t;

    if (pthread_mutex_init(sync_mutex, NULL) < 0)
        printAbort("Failed to initialize sync_mutex");

    // worker threads will be created detached, to avoid zombies
    pthread_attr_t attr;
```

```
    if (pthread_attr_init(&attr) < 0)
        printAbort("Failed to pthread_attr_init");

    if (pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) < 0)
        printAbort("Failed to pthread_attr_setdetachstate");

    // create workers
    for (int i=1 ; i<(number+1) ; i++) {
        pthread_t pt_worker_id;
        struct worker_params * params = new struct worker_params;
        params->serversock = serversock;
        params->workerid = i;
        params->mutex = sync_mutex;
        params->nodename = nodename;
        if (pthread_create(&pt_worker_id, &attr, &(workerThread), (void *)params) < 0)
            printAbort("Failed to create worker");
    }

}


/**
*  Main: read config, start framework, start workers, sleep forever
*/
int main(int argc, char *argv[])
{
    if (!getConfig(argc, argv)) return tellUsage();

    // Initialize framework
    GeckoFramework * framework = new GeckoFramework(
        &(config.local_addr),
        config.configfile,
        config.ifname   );

    // Create policy, configure and bind to framework
    GeckoRandomPolicy * policy = new GeckoRandomPolicy(framework);
    if (config.distribute_remote) policy->DoNotDistributeLocally();

    // Set random policy
    framework->SetPolicy(policy);

    // Join the network
    framework->JoinDistributedServer();

    // Get serversocket
    GeckoServerSocket * serversock = NULL;
    framework->GetServerSocket(&serversock);

    // Start workers
    startWorkers(config.nr_workers, serversock, config.nodename);

    // User should ^c to exit server ungracefully
    while (true) {
        sleep(5);
    }

    return 0;
}
```