Enabling Service Adaptability with Versatile Anycast

M. Szymaniak[†], G. Pierre[†], M. Simons-Nikolova[‡], M. van Steen[†]
Vrije Universiteit Amsterdam[†], Philips Research Labs Eindhoven[‡]
{*michal,gpierre,steen*}*@cs.vu.nl, mariana.nikolova@philips.com*

Abstract

We present versatile anycast, which allows a service running on a varying collection of nodes scattered over a wide-area network to present itself to the clients as one running on a single node. Providing a single logical address enables the client-side software to preserve the traditional service access model based on single access points. At the same time, the dynamic composition of anycast groups implemented by versatile anycast enables the server-side service infrastructure to evolve and adapt to changing network conditions.

We implement versatile anycast using Mobile IPv6, which decouples the logical addresses of mobile nodes from their physical location. We exploit that decoupling to implement logical service addresses that are not bound to any physical nodes, and employ standard MIPv6 mechanisms to dynamically map each such address onto individual service nodes. Our solution enables a service to transparently hand off clients among the service nodes at the network level while preserving optimal routing between the clients and the service nodes.

We demonstrate that the overhead of versatile anycasting is very low. In particular, the client-perceived handoff time is shown to be a linear function of the latencies among the client and the service nodes participating in the handoff.

1 Introduction

Grid applications are changing. The Grid was created as a distributed super-computer hosting standalone parallel computations [1]. Such computations can often be managed by a single "coordinator" node responsible for submitting node allocation requests to the grid scheduler, distributing the application input to the allocated nodes, and retrieving the results. The coordinator then becomes the natural entry point to communicate with the grid application.

However, grid applications increasingly often adopt a different model in which they provide *services* to one or more external components [2]. By means of standard SOAP-based interfaces, grid services can now be used by other applications running either inside or outside of the grid.

An important property of such services is that they might be instantiated dynamically to provide the required quality of service. For example, a service might dynamically adapt its capacity to guarantee consistent response times to its users. This typically involves changes in the number and location of machines running service instances.

Another reason for a grid service to change its hardware base is the evolution of service implementation. For example, a regular Web service might migrate from a single node to a distributed grid infrastructure in order to benefit from advanced adaptation and capacity planning features. Such a migration is likely to result in significant design changes in the architecture of the service. However, a coordinator-based service access model restricts the type of possible architectural changes, as it relies on fixed coordinators to act as gateways to the service.

The need for service adaptability often conflicts with the traditional service access model implemented by clients. To easily integrate a grid service into their own applications, clients typically require that the service maintains a single stable access point implementing a SOAP interface. On the other hand, the service needs the ability to change its internal organization at will, preferably transparently to the clients.

To satisfy both these requirements, we propose to decouple the conceptual communication model exploited by grid clients from the actual service implementation. The idea is to identify a grid service by a single stable *logical* network address. Similar to the addresses of traditional access points, a logical address can always be used to communicate with the service. However, the difference is that logical addresses are not bound to any physical node. Instead, each such address can be dynamically mapped to any grid node while the service retains full and timely control of this mapping.

Decoupling the communication model from the service implementation has several advantages. First, it allows the service architecture to evolve without being constrained by its original design, which improves service adaptability. Second, as upgrading the service infrastructure does not affect client-side applications, they might be implemented by multiple vendors independent of the service operator. Such an approach has been followed by many successful Internet services, including the Web and BitTorrent [3, 4].

Assigning a logical address to a group of physical nodes comes close to anycasting. Anycast was proposed as a routing and addressing scheme by which traffic sent to an anycast address automatically reaches some node within the addressed group [5]. This functionality is typically implemented by means of routing algorithms, which cause Internet routers to redirect anycast traffic according to some network proximity metric.

However, efficient usage of resources available within a grid service depends on many factors other than network proximity. For example, differences in the utilization of individual grid nodes make it necessary to route client requests according to additional metrics such as network bandwidth and cpu load. Also, as the utilization and availability of service nodes changes dynamically, the routing decisions must have immediate effects to prevent client requests from being redirected to overloaded or unavailable nodes. Finally, efficient anycasting should not introduce significant overhead compared to unicast communication.

The limitations of routing-based anycast resulted in proposing many alternative anycast implementations. They provide anycast functionality using either front ends, DHTs, DNS-based redirection, or anycast-aware client-side software. However, as we discuss in a previous study, none of these implementations eliminates all the limitations entirely [6].

Our solution lies in the design of *versatile anycast*, in which each anycast group retains full and timely control over how the incoming traffic is switched among the individual nodes within that group. At the same time, our implementation does not incur any significant communication overhead compared to unicast communication. These two properties enable us to implement the logical addresses of grid services as anycast addresses provided by versatile anycast.

We implement versatile anycast by exploiting the logical separation of network addresses that Mobile IPv6 assigns to mobile nodes. In principle, each Mobile IPv6 node has a permanent “home” address, which identifies the node, and a temporary “care-of” address, which identifies the node’s current location. Mobile IPv6 ensures that network traffic sent to home addresses is transparently forwarded to their care-of counterparts. To this end, it relies on clients communicating with mobile nodes to translate between home- and care-of addresses.

This article demonstrates that the very same translation mechanisms can also be used to equip grid services with logical addresses. In that case, a service as a whole can be perceived by its clients as some fictitious mobile node, regardless of the current composition of the service infrastructure. The logical address of the service is implemented as the home address of the fictitious mobile node, whereas the addresses of individual nodes within the service infrastructure can be treated as potential care-of addresses of that node. Using traffic-switching mechanisms provided by Mobile IPv6, a grid service can transparently handoff its clients to individual nodes at which it is hosted. Note that implementing logical addresses in the network layer allows for leaving the higher layers of client-side software untouched. This means in particular that with a relatively small number of server-side modifications, our scheme can be incorporated into *any* service exploiting the traditional access model based on single access points, including those that already exist.

A preliminary discussion on the usefulness of versatile anycast has been reported in [6]. However, it only contains a high-level description of the functionality, and no performance evaluation. The contributions of the present article are threefold: (i) a full description of versatile anycast implementation; (ii) details on the integration of versatile anycast with higher-level protocols such as TCP; and (iii) an in-depth performance evaluation and discussion on various possible optimizations. None of these contributions have been reported in [6].

The remainder of this article is structured as follows. Section 2 describes the architecture of grid services equipped with logical addresses, and demonstrates how such addresses enable service adaptability. Section 3 presents related work and explains why it is hard to provide logical addresses using current techniques. Section 4 describes how grid services can exploit versatile anycast to implement logical addresses, and how versatile anycast can be implemented using Mobile IPv6. Finally, Section 5 evaluates the performance of our anycast implementation, and Section 7 concludes.

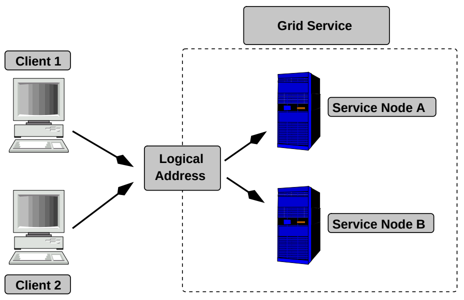


Figure 1: Accessing grid services via logical addresses

2 System Model

2.1 Overview

Introducing logical addresses enables grid services to decouple the client-side software development from the service-side infrastructure design. Figure 1 depicts the conceptual service access model. In principle, nothing changes from the perspective of the service clients, which access the service at its logical address just like they traditionally communicate with the addresses of physical service gateways.

However, the logical address is not bound to any physical node. It is therefore the responsibility of the service to ensure that all the traffic heading to that address is routed to the physical address of one or more nodes within the service infrastructure. To this end, the service transparently maps the logical address onto the physical addresses of the service nodes. As long as the mapping mechanism enables the service to dynamically switch the clients among the service nodes, each of these nodes can join and leave the service infrastructure at will, allowing for service adaptability.

2.2 Properties

The functionality of logical addresses requires them to have a number of properties. The first fundamental one is transparency, which means that using logical addresses cannot mandate any changes to the client-side software, which must be able to communicate with grid services via logical addresses just like via their physical counterparts. The second fundamental property is efficiency, which means that accessing grid services via logical addresses should not incur significant overhead compared to doing so via physical addresses. In particular, the clients should be able to communicate efficiently with a grid service even when service nodes are scattered over a wide-area network, which is often the case with massively popular network services [7].

Another group of properties is related to the mapping of logical addresses onto physical nodes. For example, efficient usage of service nodes requires fine-grain control over which clients are redirected to which nodes. This means that the logical address implementation must enable the service to redirect each client separately and according to any set of metrics. For example, classical load-balancing schemes route traffic based on the current load of each service node, and on the network distance between clients and service nodes [8, 9].

Another characteristic of modern grid services is that they are running on large collections of nodes that can dynamically join and leave the service infrastructure [10]. As a consequence, the service infrastructure might experience frequent changes in its hardware composition. However, such changes should not affect the service performance, and so the service should be able to quickly adapt to sudden departures of service nodes. This can be achieved by transparently redirecting clients from the departing nodes to those remaining operational, which requires that each client can be switched from one service node to another at any moment.

However, while switching traffic is relatively easy when clients communicate with grid services using connection-less protocols, it becomes far more complex when connection-oriented protocols, such as TCP, are used for this purpose. This is because these protocols require some state information to be maintained by both the clients and the service nodes. To guarantee that client connections are not broken upon switching, the logical address implementation must ensure that traffic switching is coordinated with its corresponding state transfer between service nodes.

3 Related Work

The traditional access model to grid services is described in the Open Grid Service Infrastructure (OGSI) specification [11]. In essence, OGSI defines mechanisms for communication with grid services, including a set of conventions ruling the interactions between grid services and their clients. According to OGSI, clients communicate with grid services via interfaces specified using Web Services Description Language (WSDL). The interfaces are implemented by service instances running on service nodes. Each such instance listens to client requests at its respective port.

OGSI defines a number of mechanisms that enable the clients to locate their respective service instances. These mechanisms enable each client to resolve grid service handles (GSH) into concrete network addresses and port numbers at service nodes.

GSHs can be perceived as logical service addresses. However, their implementation is relatively complex, as it relies on a number of additional components responsible for reliable and fault-tolerant resolution of GSHs. This forces the client-side software to implement all the resolution protocols on top of those necessary to communicate with grid services themselves. Also, it forces the grid service to deploy and maintain the components for GSH resolution. On the other hand, we demonstrate that logical addresses can be implemented such that the clients continue to communicate with a grid service as if it was running on a single node, and without all the complexity incurred by GSHs.

There exist several techniques that enable service adaptability by means of logical addresses in a less complex manner than OGSI. A number of them achieves that by mandating modifications to the client-side software such as those organizing clients and service nodes into P2P overlays [12, 13, 14]. However, as one of our main goals is to keep the client-side software untouched, none of these solutions is attractive in our case.

Several techniques implement logical addresses without modifying the client-side software. The classical one implements a logical address as that of a physical front end, which forwards client traffic to individual nodes hosting a grid service [15]. Such a solution offers real-time and fine-grain control over the client traffic. However, when used in wide-area setups, front ends tend to become performance bottlenecks, as they limit network bandwidth available to the service and introduce additional client access latency [16].

Another common solution is to map clients to service nodes using DNS. In that case, each service is identified by its DNS name rather than a network address. The mapping of the service DNS name onto the addresses of service nodes is performed by the DNS server responsible for that name. In the essence, this DNS server can resolve the service DNS name to the addresses of different service nodes such that the client requests are ultimately scattered over multiple service nodes [17].

DNS redirection has been successfully employed by many large-scale distributed systems, as it integrates transparently into the Internet communication model, exploits the scalability of DNS, and provides fairly good control over client redirection [18]. However, DNS caching can severely delay updating the redirection mappings, as many DNS servers are configured to ignore short TTL values. This makes DNS unattractive to adaptable grid services, which need to tolerate rapid changes in their hardware configuration. Also, since DNS names are resolved only before the actual communication is initiated, they cannot be used to switch clients between service nodes while communication is already in progress.

Logical addresses could also be implemented by means of anycasting. Anycast is a network addressing and routing scheme whereby data sent to an anycast address are routed to one of many nodes within its corresponding anycast group [5]. The chosen node is typically the "nearest" or "best" to the data sender as viewed by the network topology. The classical anycast implementation relies on routing algorithms, which cause Internet routers to redirect anycast traffic according to some network proximity metric.

Grid services could implement logical addresses as anycast addresses. In that case, all the nodes hosting a given service would form an anycast group, and the anycast implementation would naturally spread the client traffic heading to the logical address among these nodes. Anycasting would therefore implement the conceptual service access model described in Section 2.

Using anycast to implement logical addresses would preserve the traditional service access model based on a single access points, as each client would communicate with a grid service via the single anycast address of that service. At the same time, the adaptability of the service infrastructure would be preserved as well, as anycast groups are by nature supposed to change dynamically. However, our earlier study demonstrated that none of the current anycast implementations can provide all the properties required of logical addresses, such as efficiency, timely and fine-grain traffic control, or connection-aware client switching [6].

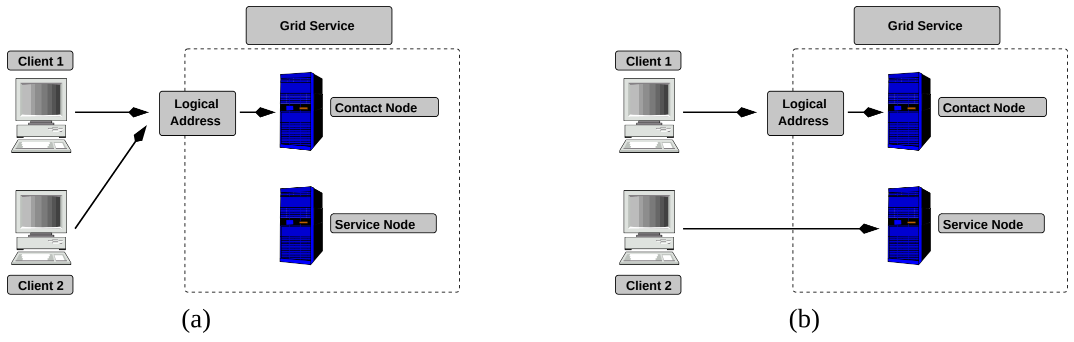


Figure 2: Versatile anycast: establishing contact (a), and client handoff (b)

The anycast-based approach has recently been followed in OASIS, which essentially provides a general-purpose anycasting functionality to Internet services [19]. OASIS integrates multi-node services into a global infrastructure in order to perform accurate network measurements, which in turn allows for mapping service clients to their proximal service nodes. The strength of OASIS lies in its advanced mapping policy. However, as OASIS relies on standard redirection mechanisms such as DNS, it also inherits their limitations discussed above. In fact, our anycast implementation proposed in this paper could be used by OASIS as yet another redirection mechanism.

The following sections discuss in detail how to implement *versatile anycast*. In contrast to the previous anycast implementations, versatile anycast provides all the properties required of logical service addresses. We first discuss the architecture details, and then demonstrate that the overhead of versatile anycasting is very low.

4 Versatile Anycast

Versatile anycast allows grid services to implement the conceptual service access model discussed in Section 2, in which the client traffic is redirected from the logical address of a grid service to the physical address of some service node. Versatile anycast achieves that by implementing the logical address as an anycast address, and by switching the traffic heading to that address among the service nodes forming its corresponding anycast group.

Versatile anycast works in two phases. First, it ensures that the client traffic sent to the anycast address reaches a designated service node called a *contact node* (see Figure 2a). This is achieved by assigning the anycast address to the contact node. However, to preserve service reachability even after the contact node becomes unavailable, versatile anycast allows the anycast address to be re-assigned to any other service node at any moment, which effectively turns that node into a new contact node.

Of course, the contact node should not service all the clients by itself. Rather, it should distribute the client-handling effort among other service nodes. This constitutes the second phase of versatile anycasting, in which the contact node transparently hands off individual clients to other service nodes, potentially causing different clients to be serviced by different service nodes (see Figure 2b). Note that once a client is handed off to some service node, the contact node is no longer involved in the communication with that client. Also, each service node can further handoff its clients to any other service node at any moment. These two features are crucial for service adaptability, as they enable each service node to share its load with new service nodes when they join the service infrastructure, and to leave the service infrastructure without disturbing its clients by handing them off before leaving.

We propose to implement versatile anycast using the address-translation capabilities provided by the Mobile IPv6 protocol. These capabilities have originally been introduced to enable communication with mobile nodes while they move among various networks. However, we demonstrate that one can also exploit these capabilities to implement versatile anycasting.

The following section discusses some basic aspects of Mobile IPv6, which is the standard protocol designed for mobile communication. Then, we show how selected functions of Mobile IPv6 can be employed to implement versatile anycast.

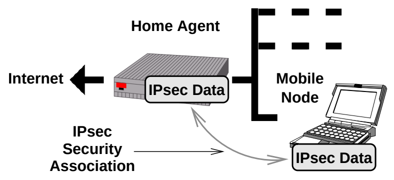
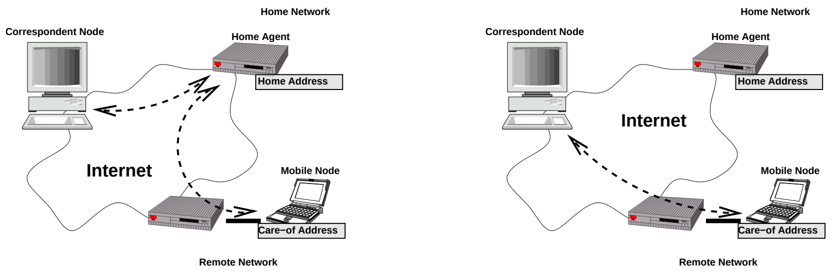


Figure 3: Home network in Mobile IPv6



(a) (b)

Figure 4: Communication in MIPv6: tunneling (a), and route optimization (b)

4.1 Mobile IPv6

Mobile IPv6 (MIPv6) consists of a set of extensions to the IPv6 protocol [20]. MIPv6 has been proposed to enable any *IPv6 mobile node* (MN) to be reached by any other *correspondent node* (CN), even if the MN is temporarily away from its usual location.

MIPv6 assumes that each MN belongs to one home network, which contains at least one MIPv6-enabled router capable of serving as a *home agent* (HA). Such an HA acts as a representative for the MN while it is away.

An HA must authenticate MNs before it can start representing them [21]. To this end, each MN must establish an *IPsec security association* with its HA in its home network (see Figure 3). Such associations are established using the Internet Key Exchange [22].

To allow one to reach an MN while it is away from home and connected to some visited network, MIPv6 distinguishes between two types of addresses that are assigned to MNs. The *home address* identifies an MN in its home network and never changes. An MN can always be reached at its home address. An MN can also have a *care-of address*, which is obtained from a visited network when the MN moves to that network. The care-of address represents the current physical network attachment of the MN and can change as the MN moves among various networks. The MN reports all its care-of addresses to its HA.

The goal of MIPv6 is to ensure uninterrupted communication with MNs via their home addresses and independently of their current network attachment. To this end, MIPv6 provides two mechanisms to communicate with MNs that are away from home. The first mechanism is *tunneling*, wherein the HA transparently tunnels the traffic targeting the home address of an MN to the care-of address of that node (see Figure 4a).

The advantage of tunneling is that it is totally transparent to the CNs. Hence, no MIPv6 support is required from any node other than the MN and its HA. However, tunneling can also lead to two problems. First, if many MNs from the same home network are away, then their shared HA can become a bottleneck. Also, if the distance between an MN and its home network is large, then tunneling can introduce significant communication latency. These two problems are addressed by the second MIPv6 communication mechanism, called *route optimization*. It enables an MN to reveal its care-of address to any CN to allow direct communication (see Figure 4b).

Route optimization is prone to address spoofing. To protect itself, the CN must authenticate the care-of address using a *return-routability procedure*, which is used to verify that the same MN can be reached at the HA and at the care-of address.

The return-routability procedure is initiated by the MN which simultaneously sends two messages to the

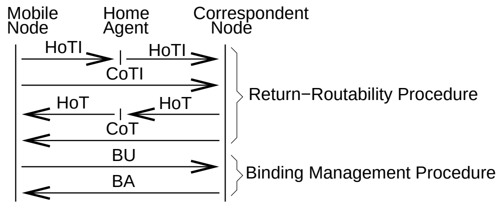


Figure 5: Route optimization protocol

CN (see Figure 5). The first message, called *Home Test Init* (HoTI), is tunneled through the HA, whereas the second message, called *Care-of Test Init* (CoTI), is sent directly. The CN retrieves the MN's home address and care-of address from the first- and second message, respectively. The CN responds with two messages, *Home Test* (HoT) and *Care-of Test* (CoT). The HoT message is tunneled to the MN through the HA, whereas the CoT message is sent directly.

The HoT and CoT messages contain home- and care-of keygen tokens, respectively, which are combined to create a *binding management key* (Kbm). The ability of the MN to create the Kbm based on the tokens received via two different paths is the proof that the MN has passed the return-routability procedure and that the home- and care-of addresses correspond to the same MN.

The MN uses the Kbm to authorize the *binding management procedure*. The goal of this procedure is to create the mapping between home- and care-of address at the CN such that it communicates directly with the MN. To this end, the MN sends the Kbm to the CN in a message called *Binding Update (BU)*. This message also contains the home address, the care-of address, the lifetime of the requested home-to-care-of address mapping, and a sequence number, which orders all the BU messages sent by a given MN to a given CN.

Upon receiving the BU message, the CN verifies that the Kbm found inside that message is valid and matches the home/care-of address pair. In this way, the CN can now be certain that the MN has passed the return-routability procedure. It therefore creates a *binding cache entry* for the MN, which is essentially a mapping between home- and care-of address. The binding cache entry allows the CN to translate between home- and care-of address in the incoming and outgoing traffic, which enables the CN to communicate with the MN directly at its care-of address. This eliminates the latency introduced by tunneling, and offloads the HA.

As the last step of route optimization, the CN confirms creating the binding cache entry by sending a *Binding Acknowledgment* (BA) message to the MN. Note that binding cache entries are deleted once their lifetime expires, and must be therefore periodically refreshed. The MN can also cause an old binding cache entry to be deleted immediately by sending a new BU message with the lifetime set to zero. Such a message can be sent without performing the return-routability procedure.

Route optimization is less transparent than tunneling, as the IP layer at the CN is aware of the current physical attachment of the MN. However, that information is confined inside the IP layer. The CN uses it to translate source and destination addresses in IP packets exchanged with MNs according to the binding cache entries created during the binding management procedures.

Translating addresses in the IP layer hides care-of addresses from higher-level protocols such as TCP and UDP. As a consequence, these protocols use only the home address of an MN and the changes in the MN's location remain transparent to applications running on CNs.

4.2 Employing Mobile IPv6 for Versatile Anycasting

Our implementation of versatile anycast exploits the fact that Mobile IPv6 decouples home- and care-of addresses, effectively allowing for the traffic directed to the former to be transparently redirected to the latter. This comes close to the anycast communication model, in which traffic sent to the anycast address of an anycast group is routed to the interface of some node within that group. We exploit our implementation of versatile anycast to transparently redirect the clients of a grid service from its logical (anycast) address to the individual service nodes.

More specifically, versatile anycast presents a grid service to its clients as a single fictitious MN. The anycast address X of that service then becomes the home address of that fictitious MN. The addresses of the service

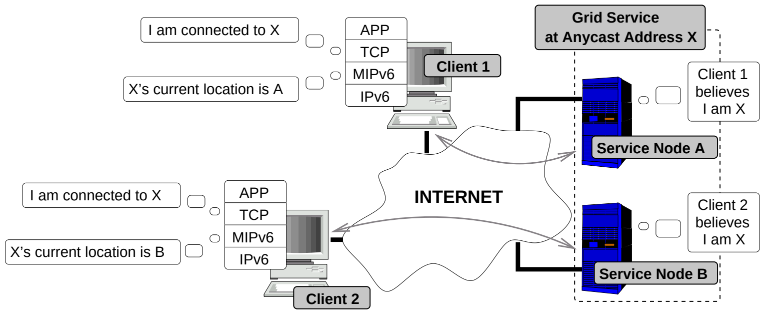


Figure 6: Communication with a grid service

nodes, in turn, act as care-of addresses to which the traffic can be redirected. By disclosing different care-of addresses to different clients, versatile anycast can convince different clients that the MN has moved to different locations (see Figure 6). Note that the client's higher (transport and application) layers retain the illusion that they communicate with the one and only node holding address X, as the translation between home- and care-of addresses is confined in the network layer. This effectively enables the service nodes to jointly service their clients via a single anycast address, which allows for preserving the traditional service access model based on single access points.

The following sections discuss how we implement the two phases of versatile anycast using Mobile IPv6. We first demonstrate how to implement the anycast address such that it is always reachable, yet it can also be moved between subsequent contact nodes. Then, we show how to implement transparent client handoffs between service nodes. Both these features together allow service nodes to join and leave the service infrastructure at will, thus enabling service adaptability. Note that the following sections assume that each service client supports MIPv6. We discuss how a service can handle clients that do not support MIPv6 in Section 6.1.

4.2.1 Anycast Address Implementation

To implement the first phase of versatile anycasting, one has to provide an anycast address and make sure that the traffic sent to that address ultimately reaches some node within the service infrastructure. A simple solution could be to choose the address of an arbitrary service node as the anycast address. In that case, however, the anycast address would be bound to this selected service node. This would make it impossible to contact the grid service once that node has left the service infrastructure.

To circumvent this problem, a completely new address must be issued that is then used as the anycast address. Dynamically creating an address is not difficult, as any IPv6 node can produce addresses belonging to its own network. The service node which created the new address can then make the address reachable by attaching it to its network interface, and advertise it as the service address. Later on, if the service node decides to leave the service infrastructure, all that needs to be done is *move* the anycast address to any other service node that remains in the service infrastructure. We refer to the service node that holds the anycast address at a given moment as a *contact node*.

To enable the service to move its anycast address at will, the contact node performs a two-step procedure immediately after having created the anycast address. First, it establishes an *IPsec security association* for that address with its home agent. Recall that such an association is used by MIPv6 to authenticate mobile nodes to their home agents. It then forwards the association data and the HA's address to one or more *backup nodes* within the service infrastructure. Given that any node holding the association data is considered by the home agent to be the contact node, any backup node can now impersonate the contact node when communicating with the home agent. Note that throughout the entire service lifetime, the service appears to that home agent as a regular mobile node. The home agent therefore does not need to run any specialized software in addition to MIPv6.

The contact node and all its backup nodes form a fault-tolerant group, whose goal is to keep the anycast address persistent. This is achieved by enabling any backup node to take over the anycast address should the contact node leave the service infrastructure. Note that the contact node must trust its backup nodes that the

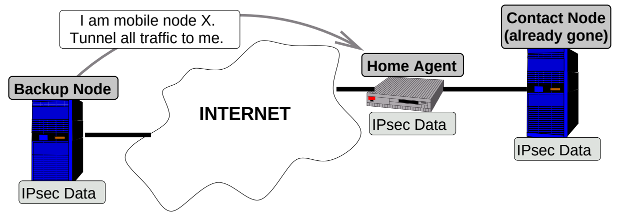


Figure 7: Taking over the anycast address

address takeover does not take place as long as the contact node remains within the service infrastructure.

To take over the contact address, one of the backup nodes convinces the home agent that it is actually the contact node that has moved to another network. To this end, that backup node authenticates itself to the home agent using the IPsec data obtained from the contact node, and reports its address as the new care-of address of the contact node (see Figure 7). This results in tunneling the traffic targeting the anycast address to the backup node through the home agent, which effectively turns the backup node into a new contact node. Doing so preserves the reachability of the anycast address as all the traffic addressed to the service keeps on reaching one of the service nodes. Note that some other backup node must take over the anycast address should the new contact node leave the service infrastructure.

Although the anycast address is now stable, service access performance might still turn out to be poor because extensive tunneling to the new contact node can overload the home agent and introduce communication latency. These limitations are addressed by route optimization wherein the care-of address of an MN is revealed to a CN. Given the care-of address, the MIPv6 layer of the CN transparently translates between home- and care-of addresses of the mobile node.

Since a grid service appears to its clients and home agents as a regular mobile node, it can also use route optimization. As a consequence, clients can communicate directly with the new contact node using its actual address. This is likely to result in better service access performance.

The remaining question is how to enable multiple service nodes to use the same anycast address simultaneously. So far, we have discussed how all the clients can directly communicate with only one service node, namely the contact node. The next section describes how the anycast address is effectively shared by enabling the contact node to transparently handoff its clients to other service nodes, which constitutes the second phase of versatile anycasting.

4.2.2 MIPv6 Handoff

The implementation of the anycast address ensures that each client request reaches the contact node. However, this node should not process all the incoming requests by itself. It therefore needs a mechanism that enables it to transparently hand off each request to another service nodes, which later may themselves transparently hand it off again. We refer to the service node that hands off a client as a *donor*, and to the service node that takes over the client as an *acceptor*.

An important observation is that while handoffs must be transparent to the client application, they need not be transparent to the underlying layers of the client-side protocol stack. For example, the MIPv6 layer running at CNs hides the movements of MNs from the upper layers by translating home addresses into care-of addresses, and vice versa. We propose to exploit this address translation to implement client handoffs between service nodes.

Recall that the address translation in MIPv6 is performed according to bindings created during MIPv6 route optimization. As we discussed in the previous section, a grid service already exploits this mechanism to establish direct communication between the contact node and the clients. However, since route optimizations are performed separately for each client, the service can also use them to hand off individual clients between any pair of service nodes.

The goal of an MIPv6 handoff is to cause the client traffic sent to the anycast address to be redirected to the acceptor's address. This requires convincing the client that the service has just changed its care-of address

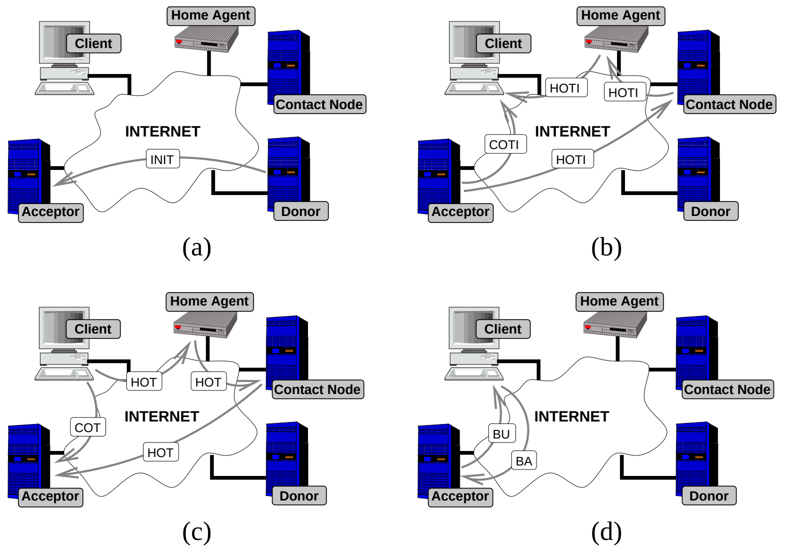


Figure 8: MIPv6 handoff

to that of the acceptor, as only then will the client update its translation bindings accordingly. To this end, the service carefully mimics the signaling of a mobile node performing route optimization.

The MIPv6 handoff signaling is coordinated by the acceptor, but initiated by the donor, which sends a special *Init* message to the acceptor. That message contains the client address and the sequence number used during the previous route optimization (see Figure 8a).

Having received the *Init* message, the acceptor starts acting as an MN running MIPv6 route optimization. It first sends the HoTI and CoTI messages (see Figure 8b). Note that the acceptor must tunnel the HoTI message to the contact node, which then tunnels it to the client through the HA.

The HoTI and CoTI messages cause their corresponding HoT and CoT messages to be sent by the client, which acts as an MIPv6 correspondent node during the MIPv6 handoff (see Figure 8c). The HoT message is also tunneled twice, by the HA and by the contact node. This requires that the contact node maintains a list of pending MIPv6 handoffs.

Having received the HoT and CoT messages, the acceptor sends a BU message to the client, which updates its binding cache entries and acknowledges the update with the BA message (see Figure 8d). From that moment on, the communication between the client and the service proceeds between the client and the acceptor.

The MIPv6 handoff enables the acceptor to communicate with the client on behalf of the service on the network level. However, grid services commonly communicate with their clients by means of stateful connection-oriented protocols such as TCP. In that case, handing off a client at the network level alone is not enough as it would break the transport-level connections. The next sections discuss how to preserve such connections during a handoff.

4.3 Transport-Level Handoff

Many grid services use the SOAP protocol for client-server communication. Messages in this protocol are typically transmitted using either HTTP or SMTP, both of which exploit TCP connections. In that case, redirecting the client's IP packets from the donor to the acceptor is not sufficient to enable the acceptor to communicate with that client. This is because maintaining a TCP connection requires that the client and the server maintain some connection state.

Preserving handoff transparency requires that apart from switching the client's IP traffic, this server-side connection state is also transferred from the donor to the acceptor. Transferring the TCP connection state from one node to another is commonly referred to as *TCP handoff*. Note that TCP handoff does not affect client-side state.

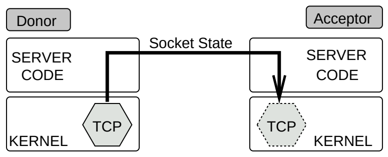


Figure 9: Socket migration

Performing a TCP handoff together with an MIPv6 handoff results in transparent switching of the complete TCP connection from the donor to the acceptor. As a result, the client and the acceptor communicate directly with each other, which eliminates the need for shared front ends often employed by clusters. This makes TCP handoffs implemented by an *adaptable* grid service fundamentally different from those implemented by their cluster-based counterparts.

This section describes how TCP handoffs are supported in an adaptable grid service. We first describe some basic properties of the TCP protocol, and then propose a procedure to hand off TCP connections on top of MIPv6 handoff.

4.3.1 TCP Properties

TCP is a reliable communication protocol based on IP. Reliability of communication is ensured by means of acknowledgments and retransmissions. In TCP, each transmitted packet is numbered and must be acknowledged by the receiver. Should that not happen within some period of time, then the packet is assumed to be lost and therefore periodically retransmitted until its acknowledgment arrives, or a timeout occurs.

TCP requires the communicating parties to maintain some state. This state mainly consists of identifiers used for recent acknowledgments and (re)transmissions, along with buffers containing the data that have not yet been sent or acknowledged. The total size of a TCP connection state depends on the buffer sizes, and varies from 90 bytes to around 90 kB.

The control states maintained by both ends of a TCP connection must remain consistent for the protocol to function properly. If one party receives a message proving that the other end is not in a legal control state, then it *resets* the connection.

Each end of a TCP connection is attached to a TCP socket. Sockets are an abstraction of various communication mechanisms provided by the operating system. Client applications and service implementations use TCP sockets to send and receive data over TCP connections. Operating systems, in turn, use TCP sockets to store the state of these connections.

4.3.2 TCP Handoff

Transferring the state of a TCP connection effectively means that the server-side TCP socket is migrated from the donor to the acceptor. To this end, the donor must extract the socket state from the operating system's kernel and send it to the acceptor. The acceptor, in turn, re-creates the socket in its own kernel based on the received state (see Figure 9).

We support TCP socket migration by means of the open-source TCPCP package [23]. It consists of a user-level library and a patch for the Linux kernel. TCPCP enables any donor application to extract an open TCP socket from the kernel in a serialized form. Given that serialized form, TCPCP re-creates the TCP socket in the acceptor application, possibly running on another node. The IP-level traffic associated with the TCP socket is assumed to be switched by some other mechanism.

The problem here is that while the socket is being migrated, the client may send data or acknowledgments to the server. We must therefore ensure that packets issued by the client during the migration can never reach a node that does not hold the corresponding TCP socket. Otherwise, the receiving member node would issue TCP control messages back reporting a missing socket, which would cause the connection to be reset. TCPCP solves this problem by maintaining two separate instances of the server-side socket during the period when it is unclear whether client-issued packets will reach the donor or the acceptor. In this way, the client traffic sent during the handoff always reaches some socket instance and can never trigger the connection reset.

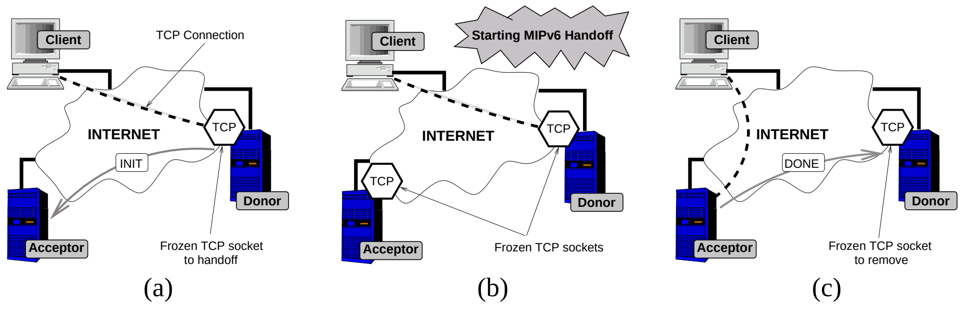


Figure 10: TCP handoff

Maintaining two server-side socket instances forces TCPCP to keep their states consistent with each other, and with the connection state held by the client. TCPCP achieves that by simply disallowing the TCP connection state to change during the migration. To this end, it freezes the socket right before extracting it from the kernel. A frozen socket does not send any data nor acknowledgments, and it silently drops all the incoming data or acknowledgments without processing them. Note that the dropped data and acknowledgments will be retransmitted by the client. The socket is unfrozen after the IP-level traffic has been switched.

The TCP handoff procedure is depicted in Figure 10. The donor first freezes and extracts the TCP socket from the kernel. The socket is then sent in the *Init* message (also used for MIPv6 handoff in Figure 8) to the acceptor, which re-creates the socket in its own kernel (Figure 10a). Having re-created the socket, the acceptor conveys the MIPv6 handoff to switch the client traffic from the donor to the acceptor (Figure 10b). Note that the two server-side socket instances are kept frozen during the MIPv6 handoff. Once the MIPv6 handoff has been completed, the acceptor unfreezes its socket, which can immediately be used to communicate with the client. The acceptor also notifies the donor about the handoff completion with a *Done* message, so that the donor can safely remove its frozen socket instance (Figure 10c).

Combining the TCP and MIPv6 handoffs allows a grid service to migrate server-side TCP sockets among nodes within its infrastructure without breaking the associated TCP connections. To maintain the handoff transparency, however, the service must also ensure that the data sent over this connection by the acceptor are consistent at the application level with those sent by the donor before the handoff. We discuss this issue next.

4.4 Application-Level Handoff

Migrating the server-side TCP socket enables the acceptor to send service data to the client over the same TCP connection that was used by the donor before the migration. As a consequence, each socket migration logically divides the service response data sent to the client into two parts, depending on which service node actually sent the data.

Preserving the handoff transparency requires that this logical division remains invisible to the client, which expects all the response data to be sent by a single service node. The part sent after the handoff must therefore seamlessly match the part sent before the handoff, and all the parts together must form a response that is valid in terms of the service protocol.

Generating subsequent response parts without violating the service protocol requires that the donor passes the application-level state of the connection to the acceptor. Given that state, the acceptor generates and sends its response part as if it had generated all the previous response parts as well.

Passing the application-level state requires it to be serialized. The serialization method is typically application-specific. In HTTP, for example, a response is generated after receiving an HTTP request, and consists of a header and the actual requested content. In that case, the serialized application-level state consists of the HTTP request being serviced, an indicator saying whether the HTTP header has already been sent, and the description of the content part that has been sent so far. If the content is a static document, then such a description can simply be the document name and the offset at which the previous content part ends.

The donor sends the serialized application-level state to the acceptor together with the *Init* message depicted in Figures 8 and 10. Recall that this message also contains all the data necessary to perform handoffs at the transport and network layers. Constructing such a message therefore requires that the donor concatenates the sequence number from the local MIPv6 implementation, the TCP socket state, and the application-level state.

To relieve the service implementation from dealing with handoffs at different levels, the construction of *Init* messages can be implemented in a separate library. We discuss the syntax of the library calls in C, but the library itself can also be implemented in Java or in any other language. The core function of that library is:

```
client_handoff(client_socket_fd, acceptor_IP, app_state)
```

which constructs the *Init* message, sends it to the acceptor, and waits for the *Done* message that signals the handoff completion. The donor would call this function to migrate a given client socket along with the application-level state to the acceptor. Once the call returns, the donor closes the client socket using the standard `close()` call.

All that is needed at the acceptor's side is to create a special socket bound to a well-known port, which is used to receive *Init* messages from donors. Whenever a message arrives at this socket, the acceptor can call another library function to accept an incoming handoff:

```
client_receive(special_socket_fd, &client_socket_fd, &app_state)
```

This function reads the *Init* message from the special socket, performs the MIPv6 and TCP handoff, and sends the *Done* message to the donor once these handoffs are complete. Finally, the function *returns* the client socket re-created by the TCP handoff and the application-level state. The service instance running on the acceptor simply needs to unserialize the received application-level state and determine what data should be sent to the client next. Once this is done, these data are transmitted over the re-created client socket just as over any other client socket created using traditional methods. However, the re-created socket must be closed using a special library function `client_close(client_socket_fd)`, which ensures that the MIPv6 binding cache entry on the client side is deleted.

4.5 Summary

Versatile anycast provides grid services with logical addresses, and enables each such service to redirect client traffic from its logical addresses to the physical addresses of the service nodes for load balancing. The service nodes can also handoff individual each client among themselves, which enables service nodes to join and leave the service infrastructure as necessary. The resulting decoupling between the logical service address and the service infrastructure contributes to the improved service adaptability and comes at the expense of upgrading the service nodes such that they support versatile anycast functions.

However, as versatile anycast is just a routing mechanism, it cannot make a grid service adaptable by itself. To this end, the service needs to implement a number of additional functions such as membership management and load balancing. These functions enable the service to react to the changes in the composition of its infrastructure, and to select service nodes to which clients should be redirected or handed off. Both membership management and load balancing can be implemented using standard techniques [24, 25].

5 Evaluation

We evaluate the performance of versatile anycast using a simple testbed (see Figure 11). The core of that testbed is a NISTnet router, which connects the client machine to a service infrastructure [26]. The infrastructure consists of two service nodes located in different networks, which are connected to the NISTnet core via their home agents.

We use the NISTnet router to emulate wide-area latencies. However, since NISTnet is not IPv6-enabled, we established three IP6-in-IP4 tunnels: *SS* to control packet transmission between the member nodes, and *CS1* and *CS2* to control packet transmission between these member nodes and the client.

The NISTnet router runs Linux 2.4.20. All the remaining machines run Linux 2.6.8.1 and MIPL-2.0-RC1, which is an open-source MIPv6 implementation for Linux [27]. All the machines are equipped with PIII processors, with clocks varying from 450 to 700 MHz.

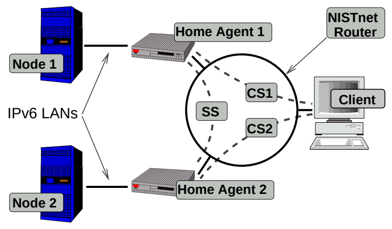


Figure 11: Testbed topology

5.1 Server Access Latency

The anycast address implementation based on tunneling provided by MIPv6 causes the client packets to be routed through the home agent, which then tunnels them to the contact node. The service access latency therefore consists of two parts: the latency between the client and the home agent, and the latency between the home agent and the contact node.

To verify this claim, we developed a simple UDP-echo application. A UDP-echo client sends a 128-byte UDP packet to the service, which sends that packet back. The client measures the round-trip time as the delay between sending and receiving the packet.

We used two different configurations of the UDP-echo service. Both configurations use the anycast addresses created by Node 1. However, whereas Node 1 belongs to the service in the first configuration, it does not in the second one. In that case, the packets are tunneled between Home Agent 1 and Node 2.

For each service configuration, we have configured NISTnet with several combinations of latency values. Packets transmitted through the SS tunnel were delayed by various latencies Lat_{SS}. Packets transmitted through the CS1 tunnel, in turn, were delayed by various latencies Lat_{CS1}. For each pair of latencies, we iteratively ran the UDP-echo client 100 times and calculated the average over the reported round-trip times.

The results were very consistent. The average reported round-trip time was $2 * Lat_{CS1} + X$ for configuration 1, and $2 * Lat_{CS1} + 2 * Lat_{SS} + Y$ for configuration 2, where X and Y are small additional delays (on average 2.13 ms and 3.61 ms, respectively). We attribute the X and Y delays to the latency of Ethernet links and the time of local processing at all the machines visited by the UDP packets.

Recall that the grid service can use route optimization to enable direct communication between its clients and the contact node. However, since route optimization takes place in parallel to the application-level communication, we do not consider it in this experiment, and analyze it only when evaluating the handoff times below.

5.2 Handoff Time Decomposition

Versatile anycast enables the service nodes to hand off a client TCP connection among each other. In this experiment, we investigate how much time is necessary to hand off a TCP connection, and what operations consume most of that time.

Handoffs are performed by a simple service that delivers 1 MB of content upon request. The client first opens a TCP connection to Node 1 acting as the contact node. Node 1 transfers 500 kB of data, and hands off the connection to Node 2 immediately after the last `send()` call returns. Node 2 sends another 500 kB of data and closes the connection.

The total handoff time can be divided into seven phases (see Table 1). The phases are delimited by the event of sending or receiving some specific packets, which we time-stamp to mark the boundary between subsequent phases. To detect events, we monitor all the packets exchanged in the testbed using *tcpdump* listening on all the network interfaces of the NISTnet router.

Table 1 reports the delays averaged over 100 download sessions. We have emulated various speeds of the upstream DSL connections by shaping the traffic sent from the home agents to the NISTnet router using the standard *cbq* queuing discipline available in the Linux kernel. The results for unshaped 100 Mbps Ethernet are included for completion.

Table 1: Handoff time decomposition (without NISTnet delays)

No.	Operation Name	Inter-node Bandwidth			
		100 Mbps	2 Mbps	1.5 Mbps	1 Mbps
1	Socket Extraction	0.8 ms	5.8 ms	6.9 ms	11.8 ms
2	State Transfer	6.5 ms	319.1 ms	434.1 ms	648.2 ms
3	Socket Re-creation	2.2 ms	2.1 ms	2.1 ms	2.2 ms
4	Return-Routability Procedure	2.5 ms	3.7 ms	4.9 ms	8.9 ms
5	BU-Message Construction	2.7 ms	2.7 ms	2.7 ms	2.7 ms
6	Binding-Management Procedure	2.6 ms	2.6 ms	2.6 ms	2.6 ms
7	Socket Activation	1.1 ms	1.1 ms	1.1 ms	1.1 ms
Total Time:		18.4 ms	337.1 ms	454.4 ms	677.5 ms

As can be observed, extracting the socket at the donor apparently takes between 0.8 and 11.8 ms depending on the network bandwidth (Phase 1). However, since this operation is entirely local, it should not depend on the bandwidth at all. We have therefore verified these results by measuring the actual time spent in the socket-extracting call, which turned out to be 0.8 ms on average. We believe that the higher values obtained using packet monitoring result from transmission delays introduced by bandwidth shaping.

Most of the total handoff time is spent on transferring the socket state (Phase 2). The duration of this phase is proportional to the network bandwidth, as each time the donor transfers the 90 kB of the socket state to the acceptor. This time accounts for up to 95% of the total handoff time when emulating 1 Mbps DSL lines.

Local phases such as re-creating the socket, constructing the BU message, and activating the socket correspond turn out to be relatively fast and independent of the bandwidth (Phases 3, 5, and 7). The return-routability procedure, in turn, demonstrated some dependency on the bandwidth (Phase 4). However, since the packets transmitted during this phase are very small, we believe that this dependency is artificial, and results from delaying packets by the shaping mechanism previously observed for Phase 1.

Interestingly, the artificial delays introduced by traffic shaping cannot be observed for the binding management procedure, where the BU and BA messages are exchanged between the acceptor and the client (Phase 6). This is probably because the low network activity during Phases 3-5 causes the state of the shaping mechanism to be reset by the time Phase 6 starts, which enables the two packets to be transmitted without any delay.

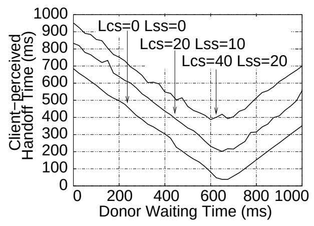
We also performed the same experiment for various combinations of L_{SS}, L_{CS1}, and L_{CS2} latencies emulated by NISTnet (we used $L_{CS1} = L_{CS2}$). The results are similar to those presented in Table 1, except that the time spent in some phases varies proportionally to the NISTnet latencies. In particular, phase 2 varies by L_{SS}, phase 4 varies by $2 * L_{SS} + 2 * L_{CS1}$, and phase 6 varies by $2 * L_{CS2}$. The additional delays correspond to the latencies of network paths followed by the messages exchanged during the respective phases. Note that should any of the MIPv6 packets be lost, it will be automatically retransmitted; in that case, the overall handoff time will obviously be extended by the MIPv6 retransmission timeout of 1 second.

5.3 State Transfer Optimization

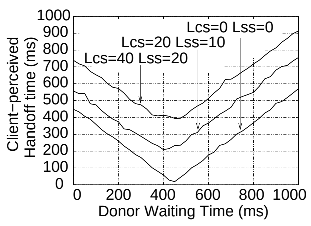
The previous experiment shows that most of the handoff time is spent transferring the socket state from the donor to the acceptor. The reason why that transfer takes so long is that in this experiment the donor extracts the socket immediately after the last `send()` call returns. This means that the socket buffers are nearly full, which results in the socket size taking about 90 kB.

One way of reducing this size is to simply wait for some time as the donor gradually sends the data stored in the socket buffers and removes the data acknowledged by the client from the buffers. This would allow the client to receive and acknowledge at least some of the data, which in turn would reduce the socket state. In this experiment, we investigate how such waiting affects the handoff time.

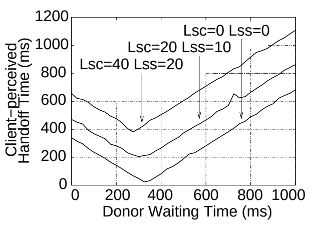
We modified our server so that it would wait for a given period of time between passing the last data to the socket and starting the actual handoff procedure. We also modified the client such that it measures its perceived handoff time. We define the client-perceived handoff time as the delay between receiving the last packet from the donor and the first packet from the acceptor.



(a) 1 Mbps

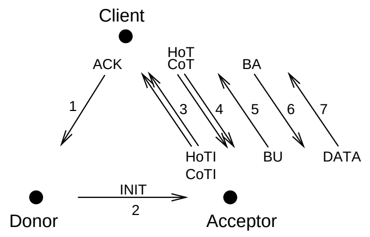


(b) 1.5 Mbps

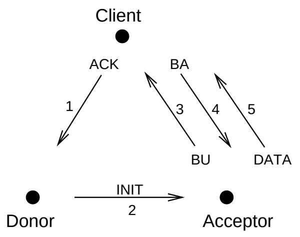


(c) 2 Mbps

Figure 12: Client-perceived handoff times for various upstream node connection bandwidths



(a)



(b)

Figure 13: Optimizing wide-area latencies

Given the modified application, we repeatedly ran 100 download sessions for 1 MB of content and waiting times varying from 0 to 1000 ms with a step of 25 ms. Similar to the previous experiments, we emulated three different DSL connection bandwidths and various combinations of wide-area latencies. The results are presented in Figure 12.

Increasing the donor's waiting time causes the client-perceived handoff time to decrease to some minimum value. Having reached that value, the client-perceived handoff time starts increasing. We verified that the minimum value corresponds to the situation when the socket was extracted right after receiving the last acknowledgment from the client, which removes the last packet from the socket buffers. As a consequence, the socket state has only 90 bytes, which can be transferred in the time of the one-way latency between the donor and the acceptor. This eliminates the delay resulting from transferring a large socket state over a low-bandwidth connection. We conclude that the donor should always empty its output TCP buffers before freezing the socket and starting the handoff.

5.4 Handoff Time Optimization

Now that the socket state is reduced to sending a single packet from the donor to the acceptor, and given that the local processing times are negligible, the actual handoff time depends only on the latencies of the paths followed by the messages exchanged during the handoff. In this experiment, we investigate whether this time can be reduced even further.

Recall that the client-perceived handoff time is the delay between receiving the last packet from the donor and the first packet from the acceptor. The beginning of the client-perceived handoff time corresponds to sending the last acknowledgment to the donor (message 1 in Figure 13(a)). Upon receiving that acknowledgment, the donor sends the *Init* message to the acceptor, which then runs the complete MIPv6 handoff. Once the MIPv6 handoff is complete, the acceptor sends the first packet containing the application data to the client. The client-perceived handoff time ends once that packet is received by the client.

In fact, all these steps do not need to be performed sequentially. In particular, the acceptor can run the return-routability procedure in advance while the donor is still busy with transferring data to the client, as no data other than the client address is needed for that. Performing the return-routability procedure in advance eliminates its time from the client-perceived handoff time, and allows the donor to send the BU message immediately after the *Init* message arrives (see Figure 13(b)).

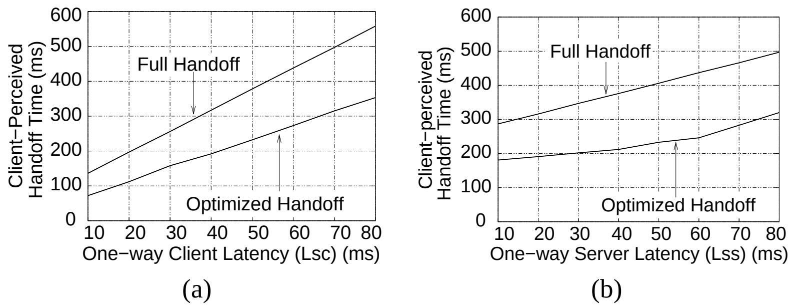


Figure 14: The impact of performing return-routability procedures in advance

To allow the acceptor to run the return-routability procedure in advance, the donor must notify the acceptor about the upcoming handoff by sending a *Prepare* message containing the client address. This message is sent when the donor has passed all its data to the socket and is about to start waiting for the last acknowledgment from the client. Having received the *Prepare* message, the acceptor conveys the return-routability procedure and keeps the resulting Kbm so that it can be sent in the BU message once the *Init* message arrives.

To investigate the impact of performing the return-routability procedure in advance, we modified our test application once again. In the new version, the donor sends the *Prepare* message immediately after returning from the last `send()` call, and then waits for the socket to become empty. The acceptor performs the return-routability procedure upon receiving the *Prepare* message, and waits for the *Init* message before sending the BU message to the client.

Similar to the previous experiment, we measured the average client-observed handoff times for 100 download sessions with the network bandwidth shaped to 2 Mbps and various combinations of L_{CS} and L_{SS} latencies. Figure 14(a) shows the results obtained for L_{SS} fixed to 20 ms and L_{CS} varying from 10 to 80 ms, whereas Figure 14(b) shows the results obtained for L_{CS} fixed to 40 ms and L_{SS} varying from 10 to 80 ms.

As can be observed, performing return-routability procedures in advance results in the reduction of client-observed handoff times. The reduction is proportional both to the latency between the client and the server and to the latency between the member nodes. This is because our optimization effectively reduces the client-observed handoff time from approximately $6 * L_{CS} + 3 * L_{SS}$ to approximately $4 * L_{CS} + L_{SS}$ since the time of tunneling the HoT/HoTI messages between the acceptor and the contact node is about L_{SS}. Note that the gain is lower if the donor's waiting time is too short to allow the acceptor to complete the return-routability procedure before the *Init* message is sent. This can sometimes be observed for large L_{SS} values, which results in an increased slope in Figure 14(b) for L_{SS} equal or greater than 60 ms.

6 Discussion

6.1 Client-side MIPv6 Support

Our proposed mechanisms assume that client-side operating systems support the functionality of an MIPv6 correspondent node. This is already true for many popular operating systems, including Linux and Windows [27, 28]. Besides, we expect that grid machines running client applications can be configured to implement MIPv6 [29]. However, it might still happen that some potential service clients do not support MIPv6.

A grid service running versatile anycast can support a small number of MIPv6-disabled clients. Recall that the client-side MIPv6 support is necessary to hand off clients among member nodes using route optimization, but it is not required to access the contact node. MIPv6-disabled clients can therefore be supported by tunneling all their traffic through the contact node. However, the number of MIPv6-disabled clients that are serviced simultaneously by the contact node should not be too large to prevent the contact node from becoming a bottleneck.

6.2 Multiple Contact Addresses

Although grid services running versatile anycast normally have a single contact node, they can also create multiple contact nodes so that the effort of forwarding requests and handling non-MIPv6 clients is spread over several service nodes. In that case, each contact node has its own anycast address, which is advertised along with the other anycast addresses. Similar to what happens in the single-address scenario, each contact node in the multiple-address scenario must have a number of backup nodes. To keep the number of trusted nodes in the server low, each contact node may decide to select the remaining contact nodes as backup nodes. As a result, all the contact nodes form a fault-tolerant group wherein all the nodes can impersonate each other as necessary.

The multiple contact addresses must somehow be advertised to the clients. To this end, a grid service may register them in the DNS. Note that this solution is significantly different from simple DNS redirection, as the DNS entries referring to logical addresses remain extremely stable even though the composition of the service infrastructure changes dynamically. Also, registering logical addresses in the DNS enables the grid service to occassionaly update the set of contact nodes so that it can stop using the home agents of former contact nodes soon after their provided addresses have been removed from the DNS.

6.3 Multiple Client Connections

Certain services might allow a client to simultaneously open multiple TCP connections to the same service, for example, to retrieve different parts of the service response in parallel. However, opening multiple TCP connections to a grid service running versatile anycast via a single anycast address can lead to problems when the server decides to hand off any of these connections. Recall that the MIPv6 handoff updates the translation bindings maintained by the client's MIPv6 layer. However, since MIPv6 translation affects *all* the traffic between the client and the anycast address, either all the connections of a given client must be handed off simultaneously to the same acceptor, or none at all.

This limitation can be alleviated if the service has multiple anycast addresses. As each translation binding is associated with only one anycast address, it does not affect the traffic sent to other addresses. Provided that the client-side application opens simultaneous connections to different anycast addresses, the service can hand off each of them just like non-parallel connections. Note that handing off parallel connections to different service nodes effectively implements a parallel download from a distributed group of nodes, which has been shown to dramatically improve the client experience [30].

6.4 Ungraceful Node Departures

In a large-scale service deployment, any node can leave the service infrastructure ungracefully, for example, because of a hardware failure. In that case, it is too late to transfer the application- and transport-level state of client connections serviced by that node to some other node. Although MIPv6 enables another node in the server to intercept the client traffic related to these connections, they can no longer be serviced without the state information, and the service is forced to close them. Such unexpected connection closing may result in the service appearing to be unreliable.

This problem can be alleviated by instructing each service node to replicate the state of all its connections across a small number of other service nodes. Should a service node leave ungracefully, the service can try to recover the connections based on the replicated state. Note that the service's ability to continue servicing a given connection greatly depends on the state of that connection. For example, it might be impossible to recover the data that have been received and acknowledged by the service node's TCP layer after the replicated connection state was updated for the last time. This is because the service cannot force the client to retransmit the already-acknowledged data. Also, recovering from ungraceful departures tends to be application-specific, as applications themselves might provide some degree of resilience to sudden service outages. We investigate various aspects of recovering client TCP connections after ungraceful departures in our present research.

7 Conclusion

We have presented versatile anycast, which allows a service running on a varying collection of nodes scattered over a wide-area network to present itself to the clients as one running on a single node. Providing a single logical address enables the client-side software to preserve the traditional service access model based on single access points. At the same time, the dynamic composition of anycast groups implemented by versatile anycast enables the service infrastructure to evolve and adapt to changing network conditions.

We propose to implement logical addresses using versatile anycast, which presents a grid service to its clients as a mobile node. This enables the service to decouple its logical address from the addresses of the nodes forming the service infrastructure. Such a decoupling allows the service to dynamically map its logical address to any service node while preserving the service reachability. Changing the mapping on a per-client basis, in turn, enables the service to transparently hand off clients among the service nodes at the network level while preserving optimal routing between the clients and the service nodes.

We have demonstrated that the overhead of contacting a service via its anycast address can be estimated as the latency between the contact node and the home agent responsible for the contact address. The client-perceived handoff time has also been shown to be a linear function of the latencies among the client and the service nodes participating in the handoff.

As a routing mechanism, versatile cannot make grid services fully adaptable by itself. To this end, a grid service must combine versatile anycast with some techniques for membership management and load balancing. These techniques enable the service to decide when to use each of the functions provided by versatile anycast.

We believe that decoupling the development of the client-side software from the evolution of the server-side service infrastructure is crucial for the successful deployment of large-scale grid services. Our implementation of logical addresses will enable a great variety of services to evolve and adapt to the increasing client demand without requiring any modifications to the client-side software.

References

[1] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid, Enabling Scalable Virtual Organizations," *International Journal on High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, 2001.

[2] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," in *Open Grid Service Infrastructure WG, Global Grid Forum*, June 2002.

[3] S. Sivasubramanian, M. Szymaniak, G. Pierre, and M. van Steen, "Replication for Web Hosting Systems," *ACM Computing Surveys*, vol. 36, no. 3, pp. 291–334, 2004.

[4] "BitTorrent, Inc.," `http://www.bittorrent.com/`.

[5] C. Partridge, T. Medez, and W. Milliken, "Host Anycasting Service," RFC 1546, Nov. 1993.

[6] M. Szymaniak, G. Pierre, and M. van Steen, "Versatile Anycasting with Mobile IPv6," in *Submitted for publication*, May 2006.

[7] L. A. Barroso, J. Dean, and U. Holzle, "Web Search for a Planet: The Google Cluster Architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[8] M. Rabinovich and A. Aggarwal, "RaDaR: A Scalable Architecture for a Global Web Hosting Service," *Computer Networks*, vol. 31, no. 11–16, pp. 1545–1561, 1999.

[9] V. Cardellini, M. Colajanni, and P. S. Yu, "Request Redirection Algorithms for Distributed Web Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 4, pp. 355–368, Apr. 2003.

[10] I. Foster and A. Iamnitchi, "On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing," in *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[11] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling, "Open Grid Services Infrastructure (OGSI) Version 1.0," Global Grid Forum Draft Recommendation, July 2003.

[12] A. Ganguly, A. Agrawal, P. Boykin, and R. Figueiredo, "WOW: Self-organizing Wide Area Overlay Networks of Virtual Workstations," in *International Symposium on High Performance Distributed Computing*, June 2006.

[13] B. Chun, P. Wu, H. Weatherspoon, and J. Kubiatowicz, "ChunkCast: An Anycast Service for Large Content Distribution," in *5th International Workshop on Peer-to-Peer Systems*, Feb. 2006.

[14] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "Scalable Application-Level Anycast for Highly Dynamic Groups," in *International Workshop on Networked Group Communication*, Sept. 2003.

[15] V. Cardellini, E. Casalicchio, M. Colajanni, and P.S. Yu, "The State of the Art in Locally Distributed Web-Server Systems," *ACM Computing Surveys*, vol. 34, no. 2, pp. 263–311, June 2002.

[16] E. Brewer, "Lessons from Giant-Scale Services," *IEEE Internet Computing*, vol. 5, no. 4, pp. 46–55, 2001.

[17] T. Brisco, "DNS Support for Load Balancing," RFC1794, Apr. 1995.

[18] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, "Globally Distributed Content Delivery," *IEEE Internet Computing*, vol. 6, no. 5, pp. 50–58, Sept. 2002.

[19] M. J. Freedman, K. Lakshminarayanan, and D. Mazieres, "OASIS: Anycast for Any Service," in *3rd Symposium on Networked Systems Design and Implementation*, May 2006.

[20] D. Johnson, C. Perkins, and J. Arkko, "Mobility Support in IPv6," RFC 3775, June 2004.

[21] J. Arkko, V. Devarapalli, and F. Dupont, "Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents," RFC 3776, June 2004.

[22] D. Harkins and D. Carrel, "The Internet Key Exchange (IKE)," RFC 2409, Nov. 1998.

[23] W. Almesberger, "TCP Connection Passing," in *Ottawa Linux Symposium*, July 2004.

[24] S. Voulgaris, D. Gavidia, and M. van Steen, "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, June 2005.

[25] V. Cardellini, M. Colajanni, and P.S. Yu, "Redirection Algorithms for Load Sharing in Distributed Web-Server Systems," in *19th International Conference on Distributed Computing Systems*, June 1999.

[26] "The NIST Net Network Emulator," `http://www-x.antd.nist.gov/nistnet/`.

[27] "MIPL – Mobile IPv6 for Linux," `http://www.mobile-ipv6.org/`.

[28] "Mobile IPv6 Systems Research Lab," `http://www.mobileipv6.net/`.

[29] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jegou, S. Lanteri, N. Melab, R. Namyst, P. Primet, O. Richard, E. Caron, J. Leduc, and G. Mornet, "Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform," in *6th International Workshop on Grid Computing*, Nov. 2005.

[30] P. Rodriguez, A. Kirpal, and E. Biersack, "Parallel-Access for Mirror Sites in the Internet," in *19th INFOCOM Conference*, Mar. 2000.