A Single-Homed Ad Hoc Distributed Server

M. Szymaniak[†], G. Pierre[†], M. Simons-Nikolova[‡], M. van Steen[†]

Vrije Universiteit Amsterdam[†], Philips Research Labs Eindhoven[‡]

Technical report IR-CS-013, Vrije Universiteit

Abstract. We present ad hoc distributed servers, which are groups of unreliable nodes scattered over a wide-area network that cooperate to create the illusion of being a single reliable server. Maintaining such an illusion requires an ad hoc distributed server to provide stable contact addresses and enable its member nodes to transparently handoff client requests among each other. We propose to implement stable contact addresses and wide-area client handoffs using Mobile IPv6. We demonstrate that the overhead due to contacting the server via its stable contact address can be estimated as the latency between the server's member nodes. The client-perceived handoff time is also shown to be a linear function of the latencies between the client and the member nodes participating in the handoff.

Keywords: ad-hoc server groups, decentralized servers, transparent service handoff.

1 Introduction

Contemporary Internet applications demand vast amounts of computing and data storage resources. However, although the number of resources available on-line grows with the Internet itself, they are more and more often provided by small and unreliable nodes. Still, we believe that exploiting these resources may turn out to be more beneficial for the application users than acquiring some expensive, yet powerful and reliable resources.

The dispersion and unreliability of resources might considerably complicate the design and implementation of Internet applications. However, application development can be facilitated if one can aggregate resources and present them as if all were provided by a single, stable, and high-performance node. Much like in existing P2P systems, we envisage that certain server applications can be hosted on an ever-changing collection of unreliable nodes. We call such a collection an ad hoc distributed server, which appears to its clients as a single reliable machine. This hides the details of the server-side system architecture from the clients, and allows that architecture to evolve as necessary.

This approach has been followed by several existing systems. For example, Grids allow their participants to share CPU power by transparently spreading computations over many distributed machines. Similarly, content delivery networks host documents on many distributed Web servers that efficiently deliver these documents to Web clients while pretending to be a single Web server.

The problem of resource aggregation is more complex in the context of ad hoc distributed servers, where the resources are typically provided by unreliable, low-performance nodes. This happens, for example, when the nodes are desktop machines connected to the Internet via DSL lines and contributed voluntarily by a group of individuals. In that case, each node has a limited request-processing capacity and can withdraw from the system at any moment. This forces the system to maintain redundant resources that can be used to replace those that suddenly become unavailable.

An ad-hoc distributed server is a group of nodes that preserves its address irrespective of the changes in the group composition. It can also transparently handoff requests among its member nodes. We show how stable addresses and wide-area client handoffs can be implemented using Mobile IPv6. In essence, our solution qualifies as low-level middleware, interacting with the underlying operating system.

The main contribution of this paper is that we describe how to construct a stable Internet server from a changing and heterogeneous collection of hosts. We describe and evaluate its performance and demonstrate how a high degree of distribution transparency can be achieved for clients accessing the server.

This paper is structured as follows. Section 2 describes ad hoc distributed servers and discusses issues that must be addressed to implement them. Sections 3, 4, and 5 respectively present functions required from the network, transport and application layers. Experimental results are shown in Section 6. Section 7 explains the issues specific to our server implementation, and Section 8 discusses related projects. Finally, Section 9 concludes.

2 System Model

2.1 Motivating Example

One potential application of ad hoc distributed servers is to use them as stable servers in a collaborative content delivery network. In contrast to commercial content delivery networks, which deploy a distributed infrastructure of servers and rent their capacity to customers [1], a collaborative content delivery network is constructed from end-user machines that have been voluntarily contributed to the system [2]. In this sense, collaborative CDNs share a number of properties with peer-to-peer overlay networks.

In a collaborative CDN, an end user can replicate a Web site originally hosted on his own node over many other nodes contributed by other end users. The site's clients are transparently redirected to their proximal nodes, for example, by means of DNS redirection [3]. Just as in regular CDNs, replication increases the number of clients that the Web site can service, and improves site availability. It is also likely to reduce the client-perceived access latency, as the clients communicate with nearby nodes [4].

One problem with collaborative CDNs is that the contributed end-user nodes cannot be relied upon to service their clients efficiently. This is because such nodes are typically connected to the Internet using low to medium bandwidth connections. Also, one cannot rely on the availability of a node, as its owner may misconfigure it or decide to (temporarily) withdraw it from the collaborative CDN at any moment.

Collaborative CDNs must address these problems to service their clients as reliably and efficiently as their commercial counterparts. In particular, nodes must be made stable, as each client is redirected to a particular node for some period of time. Furthermore, nodes must balance the request-processing load at a fine granularity so that resources are used efficiently, leading to the maximum available processing performance.

A promising solution to that problem is to create the illusion that collaborative CDNs consist of a modest number of reliable and powerful servers, instead of many unreliable and low-capacity end-user nodes. Such an illusion is provided by an ad hoc distributed server, which aggregates a number of end-user nodes spread across different networks. In particular, each ad hoc distributed server has one IP address at which it can always be contacted, regardless of which nodes actually belong to the server.

Ad hoc distributed servers can offer an arbitrary level of availability and performance, depending on the number of nodes forming the server. The high performance level, in turn, can be achieved by balancing the client load over the nodes on a very fine granularity. To this end, nodes forming an ad hoc distributed server can handoff an individual client request among each other, not only just after the request has been received, but also while it is being serviced. Such fine-grain load distribution potentially results in optimal usage of node capacities.

Our primary reason to develop ad hoc distributed servers was to improve the reliability and efficiency of collaborative CDNs by integrating many end-user nodes into a few reliable and powerful servers. However, we believe that such servers can be useful to many large-scale decentralized applications where the system is composed of unreliable and low-capacity nodes. Typically, large-scale ubiquitous computing systems, but also next-generation grids come to mind.

2.2 Ad Hoc Distributed Servers

We define an ad hoc distributed server as a group of nodes that cooperate to create the illusion that they together form a single stable server. Such an illusion allows to provide stability guarantees despite individual node failures. It also hides the details of server-side system configuration from the clients, and decouples it from the development of client-side software. Nodes forming an ad hoc distributed server, called member nodes, may be scattered over the Internet. This happens, for example, when the server consists of desktop machines contributed by individuals who are willing to jointly run some network application. This way of building a server can be very cost-effective, as it spreads both the cost and maintenance effort over many parties.

The member nodes are likely to be very heterogeneous. In general, they may be connected to the Internet using network connections of various speed and they can run different operating systems on various hardware platforms. Furthermore, since they join the server voluntarily and without providing any guarantees, they are also allowed to leave the server at any moment, either gracefully or not.

We assume that each member node belongs to some local-area IPv6 network, which is connected to the Internet via an IPv6 router. We further assume that even if a member node leaves the server, its router can still be contacted. Note that, in some cases, member-node networks may have a very simple structure and consist of only two elements (a member node and its router). This happens, for example, when a member node is connected to the Internet via its private ADSL modem. In that case, the IPv6 router is deployed either in that modem or in the ADSL provider's switchboard [5].

The dynamic composition of an ad hoc distributed server forces the member nodes to tightly cooperate in order to ensure stable application execution. This is achieved by preserving server identity and hiding the distribution and unreliability of the member nodes from the clients.

We assume that the application running on an ad hoc distributed server follows the traditional request-response model. Whenever a client request arrives, it is routed to one of the member nodes for servicing. It may also happen that the request visits several member nodes while being serviced, so that each of them generates a subsequent part of the response.

Handing off requests among member nodes must remain transparent to the clients because clients expect responses to be generated by a single server. The handoff transparency should be preserved even if a member node leaves the server while servicing clients. Also, handoffs must remain transparent to the network routers in the sense that the protocols run by these routers must not be violated.

2.3 Design Issues

From our description so far, it is clear that numerous issues need to be addressed when developing an ad hoc distributed server. These include reliability, heterogeneity, as well as dynamics of member nodes.

In this paper, we concentrate on two specific problems that are critical to maintain the illusion that an ad hoc distributed server is actually a single machine. The first problem is how to provide each server with a stable contact address that can be shared by

the member nodes; the second problem is how to enable the member nodes to handoff requests among each other such that the handoffs are transparent to the clients and the network infrastructure. For the sake of clarity, we assume in the following sections that each server has only one contact address. However, we describe how multiple contact addresses can be supported in Section 7.3.

Our solutions are based on Mobile IPv6 (MIPv6), which is a standard protocol for communication with mobile nodes [6]. We assume that this protocol is implemented by the member nodes and their routers. We also assume that MIPv6 functions are in general supported by the client-side operating systems. Section 7.2 discusses how an ad hoc distributed server can also service clients that do not implement MIPv6.

Contact Address Stability To maintain the illusion of a single server, an ad hoc distributed server must provide a single contact address that the clients can use to access it. The contact address should have the same properties as that of a centralized, highly accessible server: it should always be reachable, and it should never change.

The requirements with respect to the contact address are hard to meet in an ad hoc distributed server because of the unreliability of individual member nodes. As they can join and leave the server at any moment, a mechanism is needed to intercept the traffic targeting a member node that has just left, and re-route this traffic to some other member node for servicing.

Intercepting traffic from the network of a former member node requires that some machine in that network agrees to forward the traffic. The problem here is how to find such a machine when the former member node can no longer be contacted.

This problem can be solved by exploiting the router in the network of the former member node. That router can tunnel the traffic targeting the former member node to some other member node. Such tunneling is a standard feature provided by Mobile IPv6 and is traditionally used when a mobile node moves out of its home network to ensure that the mobile node can still be reached via its home address. The complete mechanism to provide stable contact addresses is described in Section 3.2.

Contact Address Sharing Another problem with implementing the contact address is that a client may need to be serviced by several member nodes during a single session. However, to hide the server's distribution from the client, all these member nodes must use the same contact address as the source of their response traffic to that client.

Clusters often solve this problem by proxying all the server traffic through some designated front end, whose address is advertised as the contact address for the entire cluster [7]. In an ad hoc distributed server, however, such a solution could easily create a network bottleneck, especially if the front end uses a network connection with low upstream capacity such as ADSL.

We propose to decouple the contact address from the addresses of the member nodes. The contact address can be assigned to any member, which enables the server to change this assignment on a per-client basis as necessary. Similar to the traffic interception discussed in the previous section, the address decoupling can also be implemented using Mobile IPv6. The details of this mechanism are discussed in Section 3.2.

Distributed Handoff Member nodes in an ad hoc distributed server may need to handoff client requests among each other. This happens, for example, when different response parts are generated by different member nodes, or when a member node decides to leave

the server while servicing some clients, which in that case must be taken over by some other member node.

Request handoffs must be transparent to the clients. Handing off a client therefore requires that the client status information is transferred from one member node (*donor*) to another (*acceptor*), such that the client cannot distinguish between these two. In particular, the donor and the acceptor must ensure the robustness of connections that the client might have opened to the server. This requires that the donor and the acceptor support handoffs at various levels of the communication protocol stack.

Starting from the top of the protocol stack, the donor and the acceptor must agree upon the application-level state of the client. This requires that the application supports serializing the current client state on the donor and restarting the client servicing from that serialized state on the acceptor. As we discuss in Section 5, these two operations are relatively simple for applications that deliver content upon request, such as Web servers. However, other types of applications may need more sophisticated mechanisms.

Since clients often receive application-level data using transport-level protocols such as TCP, the donor and the acceptor must also ensure that these connections are not broken upon handoff. The donor must therefore transfer its connection description to the acceptor so that the acceptor can continue sending data via the same connection while the client remains unaware that the server-side connection endpoint has moved. In TCP, this means transferring the TCP socket state, as we describe in Section 4.

Enforcing the consistency of application- and transport-level streams is still not enough, as data are transmitted in network-level datagrams which contain the network address of the client and the contact address of the server. Any mismatch in these addresses will result in breaking the communication. Maintaining network-level consistency during a handoff over a wide-area network is a complex task, as we discuss next.

3 Network-Level Handoff

We propose to implement network-level consistency such that the acceptor communicates directly with the client while appearing to the client to have the same address as the donor before the handoff. This is similar to the mobility support in IP, where mobile nodes communicate with other nodes using persistent addresses while moving among various networks.

The following section discusses some aspects of Mobile IPv6. Then, we show how an ad hoc distributed server can use it to implement reliable contact addresses and transparent network-level handoffs over a wide-area network.

3.1 MIPv6 Overview

Mobile IPv6 (MIPv6) consists of a set of extensions to the IPv6 protocol [6]. MIPv6 has been proposed to enable any *IPv6 mobile node* (MN) to be reached by any other *correspondent nodes* (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 special router capable of serving as a *home agent* (HA). Such an HA acts as a representative for the MN while it is away.

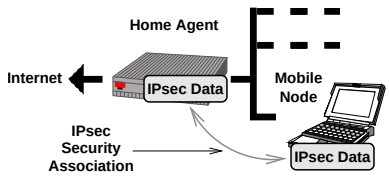


Fig. 1. Home network in Mobile IPv6

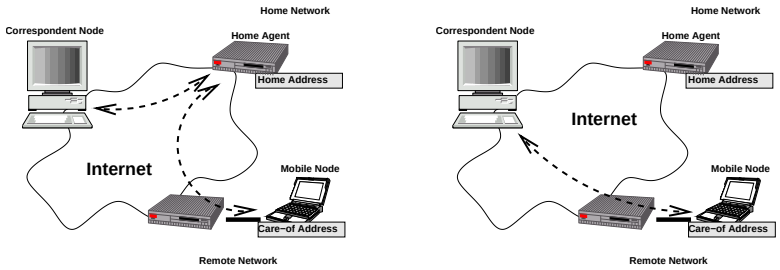


Fig. 2. Communication in MIPv6: tunneling (a), and route optimization (b)

An HA must authenticate MNs before it can start representing them [8]. To this end, each MN must establish an *IPsec security association* with its HA in its home network (see Figure 1). Such associations are established using the Internet Key Exchange [9].

To allow one to reach a 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* (HoA) identifies a MN in its home network and never changes. A MN can always be reached at its HoA. A MN can also have a *care-of address* (CoA), which is obtained from a visited network when the MN moves to that network. The CoA represents the current physical network attachment of the MN and can change as the MN moves among various networks. The MN reports all its CoAs to its HA.

The goal of MIPv6 is to ensure uninterrupted communication with MNs via their HoA 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 HoA of a MN to the CoA of that node (see Figure 2a).

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 a 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 a MN to reveal its CoA to any CN to allow direct communication (see Figure 2b).

Route optimization is prone to address spoofing. To protect itself, the CN must

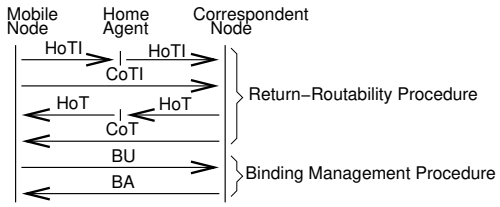


Fig. 3. Route optimization protocol

authenticate the CoA when it is being revealed using a *return-routability procedure* (RR), which is used to verify that the same MN can be reached at the HA and at the CoA.

The RR procedure is initiated by the MN which simultaneously sends two messages to the CN (see Figure 3). 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 HoA and CoA 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 RR procedure and that the HoA and CoA 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 HoA and CoA 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 HoA, CoA, the lifetime of the requested HoA-to-CoA 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 HoA/CoA pair. In this way, the CN can now be certain that the MN has passed the RR procedure. It therefore creates a *binding cache entry* for the MN, which is essentially a mapping between HoA and CoA. The binding cache entry allows the CN to translate between HoA and CoA in the incoming and outgoing traffic, which enables the CN to communicate with the MN directly at its CoA. 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 RR procedure.

Route optimization is less transparent than tunneling, as the IP layer at the CN

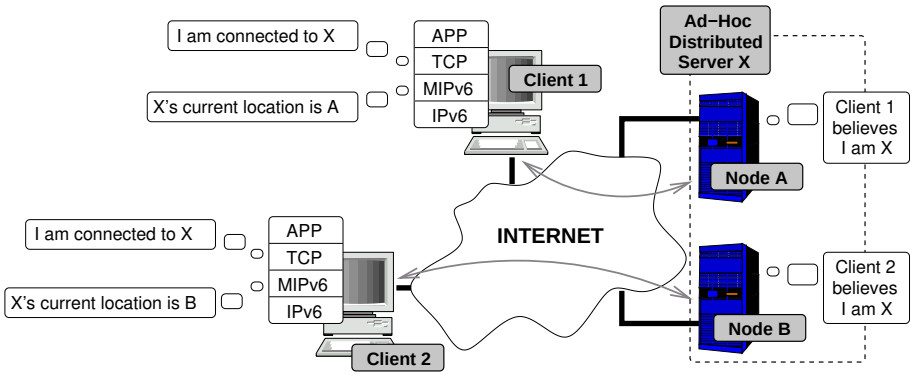


Fig. 4. Communication with an ad hoc distributed server

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 CoAs from higher-level protocols such as TCP and UDP. As a consequence, these protocols use only the HoA of a MN and the changes in the MN's location remain transparent to applications running on CNs.

3.2 Adapting Mobile IPv6

We use MIPv6 to hide the unreliability and distribution of member nodes forming an ad hoc distributed server. The essential idea is to take advantage of the address decoupling provided by Mobile IPv6, which distinguishes between the Hoa and CoA of mobile nodes.

More precisely, an ad hoc distributed server can be perceived by its clients as a mobile node. The server's contact address is then used as the home address of that (fictitious) MN. Each member node address, in turn, is considered as a potential CoA of that MN. By disclosing different CoAs to each client, the server can convince different clients that it has moved to different locations. This allows the server to service its clients via its single contact address using many member nodes, just like a MN can communicate with its CNs using its HoA at many different locations.

The general model of communication between an ad hoc distributed server and its clients is depicted in Figure 4. All clients access the server using its contact address X. The MIPv6 layers at different clients may have a different idea about the server location and therefore communicate with different member nodes. Still, the client's higher (transport and application) layers retain the illusion that they communicate with server X.

Server Contact Address When setting up a distributed server, one has to select a contact address and make sure that the traffic sent to this address effectively reaches

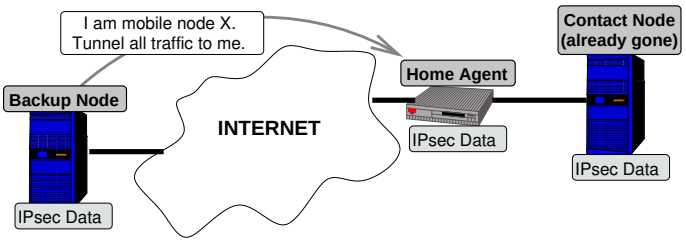


Fig. 5. Taking over the contact address

one of the server's member nodes. A simple solution could be to choose the address of an arbitrary member node as the server contact address. In that case, however, the contact address would be bound to this selected member node. Should it decide to leave the server, it would keep on receiving the server traffic.

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

To enable the server to move its contact address at will, the contact node performs a two-step procedure immediately after having created the contact 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 server. 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 system lifetime, the server 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 contact address persistent. This is achieved by enabling any backup node to take over the contact address should the contact node leave the server. Note that the contact node must trust its backup nodes that the address takeover does not take place as long as the contact node belongs to the server.

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 CoA of the contact node (see Figure 5). This results in tunneling the traffic targeting the contact node 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 contact address as all the traffic addressed to the server keeps on reaching one of the member nodes. Note that

some other backup node must take over the contact address should the new contact node leave the server.

Although the contact address is now stable, server 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 CoA of a MN is revealed to a CN. Given that CoA, the MIPv6 layer of the CN transparently translates between HoA and CoA of the mobile node.

Since an ad hoc distributed server 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 contact node using its actual address. This is likely to result in better server access performance.

A remaining question is how to enable multiple member nodes to use the same contact address simultaneously. So far, we have discussed how all the clients can directly communicate with only one member node, namely the contact node. The next section describes how the contact address is effectively shared by enabling the contact node to transparently handoff the clients to other member nodes.

MIPv6 Handoff The implementation of the server contact 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 allows it to transparently handoff the request to other member nodes, which later may themselves transparently hand it off again. We refer to the member node that handoffs a client as a *donor*, and to the member 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 protocol stack. For example, the MIPv6 layer running at correspondent nodes hides the movements of mobile nodes from the upper layers by translating home addresses into care-of addresses, and vice versa. We propose to exploit this address translation to implement distributed handoffs.

Recall that the address translation in MIPv6 is performed according to bindings created during MIPv6 route optimization. As we discussed in the previous section, an ad hoc distributed server already exploits this mechanism to establish direct communication with the client. However, since route optimizations are performed separately for each client, the server can also use them to handoff individual clients between any pair of member nodes.

The goal of a MIPv6 handoff is to cause the client traffic sent to the server to be redirected to the acceptor's address. This requires convincing the client that the server has just changed its CoA to that of the acceptor, as only then the client will update its translation bindings accordingly. To this end, the server 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 6a).

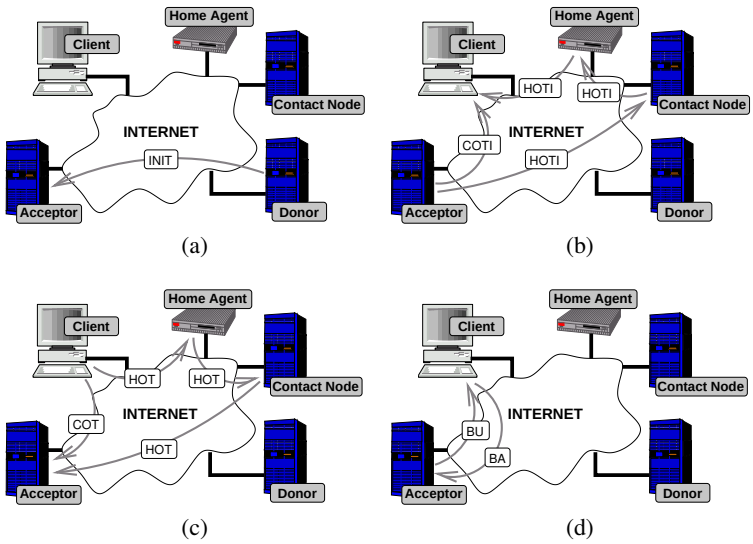


Fig. 6. MIPv6 handoff

Having received the Init message, the acceptor starts acting as a MN running the MIPv6 route optimization. It first sends the HoTI and CoTI messages (see Figure 6b). 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 a MIPv6 correspondent node during the MIPv6 handoff (see Figure 6c). 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 6d). From that moment on, the communication between the client and the server proceeds between the client and the acceptor.

The MIPv6 handoff enables the acceptor to communicate with the client on behalf of the server on the network level. However, many distributed applications communicate with the clients by means of connections provided by stateful transport-level 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 Transport-Level Handoff

Many network applications use TCP for client-server communication. 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 the TCP handoff does not touch the client-side state.

Performing a TCP handoff together with a 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 commonly employed by clusters. This makes TCP handoffs implemented by an ad hoc distributed server fundamentally different from those implemented by cluster-based systems.

This section describes how TCP handoffs are supported in an ad hoc distributed server. We first describe some basic properties of the TCP protocol, and then propose a procedure to handoff TCP connections on top of the MIPv6 handoff.

4.1 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, 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.

From the application perspective, 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. Applications use TCP sockets to send and receive data over TCP connections. Operating systems, in turn, use TCP sockets to store the states of these connections.

4.2 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 7).

We support TCP socket migration by means of the open-source TCPCP package [10]. It consists of a user-level library and a patch for the Linux kernel. TCPCP enables any

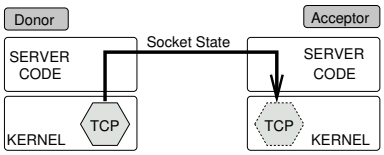
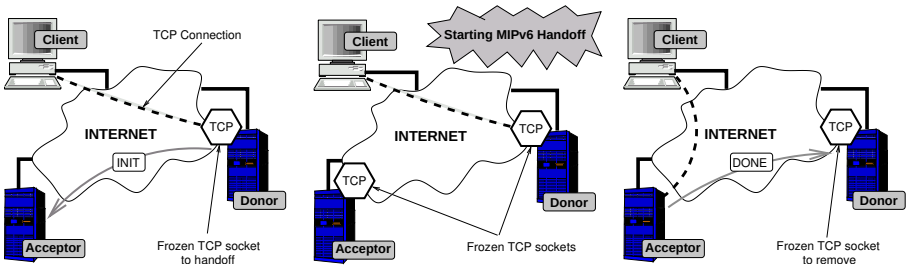


Fig. 7. Socket migration



(a) (b) (c)

Fig. 8. TCP handoff

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.

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 can be unfrozen after the IP-level traffic has been switched.

The TCP handoff procedure is depicted in Figure 8. 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 6) to the acceptor, which re-creates the socket in its own kernel (Figure 8a). Having re-created the socket, the acceptor conveys the MIPv6 handoff to switch the client traffic from the donor to the acceptor (Figure 8b).

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 8c).

Combining the TCP and MIPv6 handoffs allows an ad hoc distributed server to migrate server-side TCP sockets among its member nodes without breaking the associated TCP connections. To maintain the handoff transparency, however, the server 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.

5 Application-Level Handoff

Migrating the server-side TCP socket enables the acceptor to send response 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 response data into two parts, depending on which member 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 server. 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 application-level protocol.

Generating subsequent response parts without violating the application-level 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 6 and 8. 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 application from dealing with handoffs at different levels, the construction of Init messages can be implemented in a separate library. The core function of that library is:

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

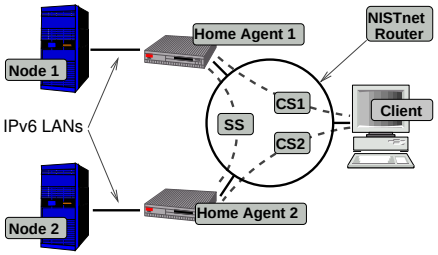


Fig. 9. Testbed topology

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 state to the acceptor. Once the call returns, the donor can close 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 state. The acceptor application simply needs to unserialize the received application state and determine what data should be sent to the client next. Once this is done, these data are transmitted using the client socket returned by the `client_receive()`. From that moment on, the application can communicate over that socket just as over any other client socket created using traditional methods. However, the 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.

6 Evaluation

We evaluate the performance of our ad hoc distributed server using a simple testbed (see Figure 9). The core of that testbed is a NISTnet router, which connects the client machine to our server prototype [11]. The server prototype consists of two member 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 support 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 [12]. All

the machines are equipped with PIII processors, with clocks varying from 450 to 700 MHz.

6.1 Server Access Latency

The stable address implementation in an ad hoc distributed server based on MIPv6 with tunneling causes the client packets to be routed through the home agent, which then tunnels them to the contact node. The server 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 server, 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 server. Both configurations use the contact addresses created by Node 1. However, whereas Node 1 belongs to the server 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 server 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 are 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 by all the machines visited by the UDP packets.

Recall that the clients of an ad hoc distributed server can use route optimization to communicate directly with 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.

6.2 Handoff Time Decomposition

An ad hoc distributed server allows its member nodes to handoff a client TCP connection among each other. In this experiment, we investigate how much time is necessary to handoff a connection, and what operations consume most of that time.

Handoffs are performed by a simple application 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 handoffs 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 packet, which we time-stamp

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	RR Procedure	2.5 ms	3.7 ms	4.9 ms	8.9 ms
5	BU Construction	2.7 ms	2.7 ms	2.7 ms	2.7 ms
6	BM 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

to mark the boundary between subsequent phases. To detect events, we monitor all the packets exchanged in the testbed using tcpdump on the NISTnet router, and configured to listen on all the network interfaces of this router.

Table 1 reports the average values over 100 download sessions. We have emulated various speeds of the upstream DSL connection 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.

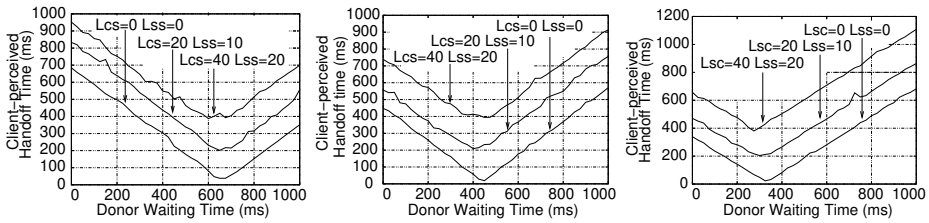
As can be observed, extracting the socket at the donor apparently takes between 0.8 and 11.8 ms depending on the network bandwidth (Phase 1). However, since this operation is entirely local, it should not depend on the bandwidth at all. We have therefore verified these results by measuring the actual 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 packet delays introduced by the bandwidth-shaping mechanism.

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 RR 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 network is empty by the time that phase starts which allows the shaping mechanism to transmit the two packets 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



(a) 1 Mbps (b) 1.5 Mbps (c) 2 Mbps

Fig. 10. Client-perceived handoff times for various upstream node connection bandwidths

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 corresponds to the latencies of network paths followed by the messages exchanged during the respective phases.

6.3 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.

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 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 10.

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 just after receiving the last acknowledgement 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

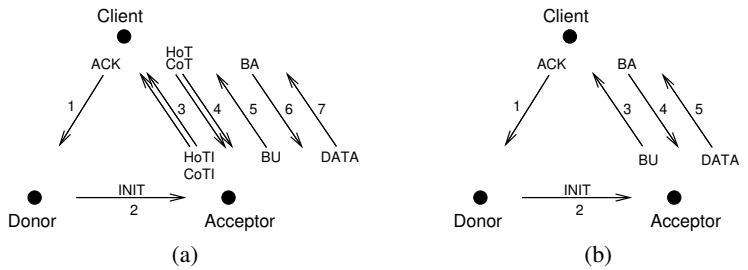


Fig. 11. Optimizing wide-area latencies

the donor should always empty its output TCP buffers before freezing the socket and starting the handoff.

6.4 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 11(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 RR 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 11(b)).

To allow the acceptor to run the RR 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 RR 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 RR 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 RR procedure upon receiving the Prepare message, and waits for the Init message before sending the BU message to the client.

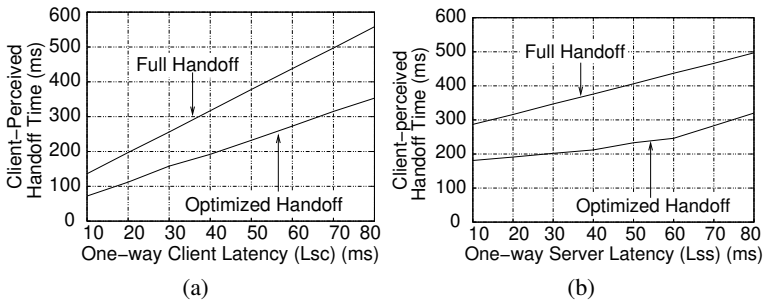


Fig. 12. The impact of performing return-routability procedures in advance

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 12(a) shows the results obtained for L_{SS} fixed to 20 ms and L_{CS} varying from 10 to 80 ms, whereas Figure 12(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 RR 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 RR 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 12(b) for L_{SS} equal or greater than 60 ms.

7 Discussion

7.1 Trusted Member Nodes

Developing an ad hoc distributed server requires that a number of trust-based relationships are formed among the member nodes. For instance, all member nodes must trust the contact node to handoff requests and assist in forwarding messages during MIPv6 handoffs. However, since the contact node anyway represents the server upon its first contact with a client, it must be trusted by nature.

Another type of trust is required to make the contact address stable. Recall that an ad hoc distributed server requires that the contact node forwards its IPsec data negotiated with the home agent to a number of backup nodes. The contact node must fully trust these nodes, since holding the IPsec data enables any backup node to impersonate the contact node against the home agent. However, the IPsec data are only related to a specific contact address, which is different from the physical address of the contact node. This ensures that the contact node can still protect its identity and communication that is unrelated to participating in the server.

7.2 Client-side MIPv6 Support

Our proposed mechanisms assume that the client-side operating systems support the functionality of a MIPv6 correspondent node. Although this is true for many popular operating systems, some potential clients may still be MIPv6-disabled [12, 13].

An ad hoc distributed server can support a small number of MIPv6-disabled clients. Recall that the client-side MIPv6 support is necessary to handoff 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 a contact node cannot be too large to prevent the contact node from becoming a bottleneck.

7.3 Multiple Contact Addresses

Although ad hoc distributed servers 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 member nodes. In that case, each contact node has its own contact address, which is advertised along with the other contact 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, an ad hoc distributed server may register them in the DNS. This also allows one to update the set of contact nodes from time to time so that the server can stop using the home agents of former contact nodes after their provided addresses have been removed from the DNS.

7.4 Multiple Client Connections

Applications sometimes allow a client to simultaneously open multiple TCP connections to the same server. In that case, the client can download several content pieces in parallel, which might reduce the total download time.

Opening multiple TCP connections to an ad hoc distributed server holding a single contact address can lead to problems when the server decides to handoff 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's address and the server's contact 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 server has multiple contact addresses. As each translation binding is associated with only one contact address, it does not affect the traffic sent to other addresses. Provided that the client-side application opens simultaneous connections to different contact addresses, the server can handoff each of them

just like non-parallel connections. Note that handing off parallel connections to different member nodes effectively implements a parallel download from a distributed group of nodes, which has been shown to dramatically improve the client experience [14].

8 Related Work

The problem of transparent aggregation of many nodes into a single logical one has been investigated for many years in the context of clusters and grid computing [15]. To this end, a plethora of techniques have been developed. Most approaches are built around a special front-end node that schedules requests among the member nodes. Other systems do not have a front end, but they require some special client-side software.

8.1 Front-end Solutions

In front-end systems, clients always contact the front end, which then forwards requests to actual member nodes [7]. Depending on the system, this forwarding can take place at different levels of the protocol stack. Responses can be returned to the clients directly if member nodes pretend to be the front end and spoof the source addresses in their responses [16, 17]. This allows one to bypass the front end, which reduces the front end's load. However, spoofing is generally prohibited in the Internet. This limits such solution to clusters situated in a single location, as the spoofing only then remains transparent to the Internet infrastructure.

In other systems, the front end is also used to forward responses to the clients [18]. This architecture potentially allows to distribute member nodes among many locations [19]. On the other hand, the front end can become the system bottleneck. This risk can be reduced by implementing the forwarding functions in the operating systems's kernel [20], or even in the hardware [21].

The front end is also potentially a single point of failure. Many systems address this issue by creating redundant front ends that run on high-quality hardware, which makes a front-end failure a rare exception. On the other hand, we have demonstrated that highly-available front ends are not necessary to build a highly-available server, as long as the server can implement a stable contact address using unreliable nodes.

8.2 Other Solutions

Connection handoffs that do not utilize any front end have been implemented in Migratory TCP (M-TCP) [22]. It allows for continuous data delivery over a TCP connection even if the server that sends the data must be removed. To this end, M-TCP enables TCP connections to be migrated among a distributed group of cooperating servers, which is achieved by extending the client-side TCP implementation. The extensions implement the migration protocol that takes care of switching TCP connections among the servers, and ensure that the connection migration is transparent to the client applications. However, in contrast to ad hoc distributed servers that use standard protocols, M-TCP requires the client-side TCP implementation to be customized.

9 Conclusion

We have presented an ad hoc distributed server, which is a group of unreliable nodes scattered over a wide-area network that cooperate to create the illusion of being a single reliable server. Maintaining such an illusion requires the ad hoc distributed server to provide stable contact addresses and enable its member nodes to transparently handoff client requests among each other.

We have proposed to implement stable contact addresses and wide-area client handoffs using Mobile IPv6. In our solution, an ad hoc distributed server pretends to be a mobile node, which allows the server to decouple its contact address from the addresses of its member nodes. Such a decoupling allows the server to dynamically map its contact address to any member node address while keeping the contact address reachable. Changing the mapping on a per-client basis, in turn, enables the server to transparently handoff clients among its member nodes at the network level while preserving optimal routing between the clients and the member nodes.

We have demonstrated that the overhead of contacting the server via its stable contact 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 member nodes participating in the handoff.

We plan to use ad hoc distributed servers in Globule, a peer-to-peer content delivery network that our group is developing [2]. We believe that they will facilitate CDN management by organizing unreliable end-user machines into stable Web hosting facilities.

References

1. J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, "Globally Distributed Content Delivery," *IEEE Internet Computing*, vol. 6, no. 5, Sept. 2002.
2. G. Pierre and M. van Steen, "Design and Implementation of a User-Centered Content Delivery Network," in *Proc. 3rd IEEE Workshop on Internet Applications*, June 2003.
3. M. Szymaniak, G. Pierre, and M. van Steen, "Netairt: A DNS-based Redirection System for Apache," in *Proc. IADIS International Conference on WWW/Internet*, Nov. 2003.
4. M. Zari, H. Saiedian, and M. Naeem, "Understanding and Reducing Web Delays," *Computer*, vol. 34, no. 12, Dec. 2001.
5. C. K. Summers, *ADSL – Standards, Implementation, and Architecture*, CRC Press, 1999.
6. D. Johnson, C. Perkins, and J. Arkko, "Mobility Support in IPv6," RFC 3775, June 2004.
7. 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, June 2002.
8. J. Arkko, V. Devarapalli, and F. Dupont, "Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents," RFC 3776, June 2004.
9. D. Harkins and D. Carrel, "The Internet Key Exchange (IKE)," RFC 2409, Nov. 1998.
10. W. Almesberger, "TCP Connection Passing," in *Proc. Linux Symposium*, July 2004.
11. "The NIST Net Network Emulator," `http://www-x.antd.nist.gov/nistnet/`.
12. "MIPL – Mobile IPv6 for Linux," `http://www.mobile-ipv6.org/`.
13. "Mobile IPv6 Systems Research Lab," `http://www.mobileipv6.net/`.
14. P. Rodriguez, A. Kirpal, and E. Biersack, "Parallel-Access for Mirror Sites in the Internet," in *Proc. 19th INFOCOM Conference*, Mar. 2000.

15. T. Anderson, D. Culler, D. Patterson, and the NOW Team, "The Case for Networks of Workstations," in *IEEE Micro*, Feb. 1995.
16. M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel, "Scalable Content-aware Request Distribution in Cluster-based Network Servers," in *Proc. USENIX Annual Technical Conference*, June 2000.
17. A. Bestavros and M. Crovella and J. Liu and D. Martin, "Packet Rewriting and its Application to Scalable Web Server Architectures," in *The 6th IEEE International Conference on Network Protocols*, Oct. 1998.
18. O. Spatscheck and J. Hansen and J. Hartman and L. Peterson, "Optimizing TCP Forwarder Performance," *IEEE/ACM Transactions on Networking*, vol. 8, no. 2, 2000.
19. W. Zhang, "Linux Virtual Server for Scalable Network Services," in *Proc. of the Linux Symposium*, July 2000.
20. A. Cohen, S. Rangarajan, and H. Slye, "On the Performance of TCP Splicing for URL-aware Redirection," in *Proc. USENIX Annual Technical Conference*, Oct. 1999.
21. G. Apostolopoulos and D. Aubespin and V. Peris and P. Pradhan and D. Saha, "Design, Implementation and Performance of a Content-Based Switch," Mar. 2000.
22. F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode, "Migratory TCP: Highly Available Internet Services using Connection Migration," in *Proc. 22nd IEEE International Conference on Distributed Computing Systems*, July 2002.