Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Reproducible evaluation of distributed servers

D3.2.6

Due date of deliverable: November 30^{th}, 2008
Actual submission date: November 30^{th}, 2008

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: WP3.2
Task number: T3.2.1

Responsible institution: VUA
Editor & and editor's address: Guillaume Pierre
VU University Amsterdam
Dept of Comp. Science
de Boelelaan 1081a
1081HV Amsterdam
The Netherlands

Version 1.2 / Last edited by Jeffrey Napper / December 5th, 2008

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	10/10/2008	Guillaume Pierre	VUA	Initial outline
0.2	13/10/2008	Guillaume Pierre	VUA	Membership protocol
0.3	20/10/2008	Jeffrey Napper	VUA	Overview and evaluation
1.0	28/10/2008	Guillaume Pierre	VUA	Wrap-up for internal review
1.1	4/12/2008	Jeffrey Napper	VUA	Edited according to comments from Yvon
1.2	5/12/2008	Jeffrey Napper	VUA	Edited according to comments from Gregor

Reviewers:
Yvon Jégou (INRIA/IRISA) and Gregor Pipan (XLAB).

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.2.1	Design and implementation of distributed servers	VUA*

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

Distributed Servers provide an abstraction that allows a group of server processes to appear as a single entity to its clients. This deliverable discusses the current state of development of Distributed Servers, and presents an in-depth functional and non-functional evaluation of the platform. First, we made significant progress towards porting Distributed Servers to run on recent Linux kernel versions, namely 2.6.25. Second, we ported the Gecko library interface to Distributed Servers to Java so that Java programs could make use of Distributed Servers functionality. This was important in particular for the integration of Distributed Servers with Virtual Nodes. Third, we developed a membership protocol to relieve application programmers from this difficult aspect of Distributed Servers management.

Our evaluation and work integrating Distributed Servers with Virtual Nodes [9] shows that Distributed Servers are both performant and mature. Performance evaluations show that Distributed Serverss are efficient: they allow for direct communication between a client and the server node that is currently serving it. Handoff times, as perceived by the client, can be reduced to an acceptably low value. Furthermore, we foresee no scalability bottleneck when using Distributed Servers to serve massive numbers of connections.

Finally, from our evaluation we conclude that Distributed Servers are mature enough platform to be integrated into the XtreemOS distribution (as soon as the XtreemOS kernel version is upgraded from version 2.6.20 to 2.6.25).

Contents

1 Introduction

Work package 3.2 of the XtreemOS project aims at *providing an infrastructure that can support highly available and scalable grid services and applications*, such that these can be developed independently from underlying instances of the XtreemOS operating system. When one builds a large-scale distributed service made of multiple service instances, an important issue is to give its user a simple contact address where queries can be sent. This is the goal of Distributed Servers: a distributed server is an abstraction that allows a group of server processes to appear as a single entity to its clients. Distributed Servers aim at allowing high-performance client-to-server communication, while being totally transparent to the clients. The only requirement is that the clients support the Mobile IPv6 protocol.

This deliverable describes the progress made in the last year (since [8]). First, we made significant progress towards porting Distributed Servers to run on recent Linux kernel versions, namely 2.6.25. Second, we ported the Gecko library interface [8] to Distributed Servers to Java so that Java programs could make use of Distributed Servers functionality. This was important in particular for the integration of Distributed Servers with Virtual Nodes (see deliverable D3.2.10 on this particular topic [9]). Finally, we developed a membership protocol to relieve application programmers from this difficult aspect of Distributed Servers management.

This deliverable is organized as follows. Section 2 describes the progress made in terms of implementation. Section 3 discusses the reproducible evaluation of the current system. Finally, Section 4 concludes.

2 Overview

Distributed Servers provide a location transparency for networked services [10]. Without modification, a client can connect to a single *distributed server address* for a service that may be divided among multiple locations. The distributed server address is simply an IPv6 [3] address provided for the network service by other name resolution techniques such as Domain Name System (DNS). For example using the distributed server address, a client first connects to a *contact node*. Subsequently, a client connection may be transparently *handed off*—the server endpoint of the connection may be transferred—to different servers to effect load-balancing or for client-specific processing. With the use of Mobile IPv6 (MIPv6) route optimization [4], this handoff provides a direct connection to the new server with corresponding network efficiencies—further data is not routed through the contact node. This ability to migrate client connections without modification of

the client depends on the implementation of Mobile IPv6, which typically supports client mobility. However, Distributed Servers inverts client mobility to provide the appearance of server mobility.

In the last year, we have further developed Distributed Servers by 1) porting Distributed Servers to a recent Linux kernel, 2) adding an interface to the Java language, and 3) implementing a membership protocol to manage both fault-tolerance of the contact node (see also [9]) and the larger set of nodes that receive client handoffs.

2.1 Port to 2.6.25 Linux kernel

The Distributed Servers system was originally built on Linux kernel 2.6.8 (released 2004), which had relatively unstable support for Mobile IPv6. Time marches on, and newer kernels such as 2.6.25 now have relatively stable support for Mobile IPv6 along with support for new hardware likely to be found in computational grids. With this in mind, we began a port of Distributed Servers to the kernel release 2.6.25 purported to be the focus of the release of XtreemOS that will include Distributed Servers. The port is largely a reimplementation as the Linux kernel network stack has changed significantly with the incorporation of IPv6 support.

Porting Distributed Servers requires porting some support software as well. The TCP connection passing system (TCPCP) provides migration of open sockets by serializing the network stack state, which can then be transferred to a different node [1]. TCPCP was written for Linux kernel 2.6.11. In porting Distributed Servers, we decided to also migrate to TCPCP2 [2] written for Linux kernel 2.6.15. This successor to TCPCP provides better support the for IPv6 sockets that we migrate in the implementation of Distributed Servers. Our migration consisted of modifying the Distributed Servers API to use the new API provided by TCPCP2. We have successfully migrated to the new API provided by TCPCP2 and upgraded support for TCPCP2 to the TCP/IP implementation of Linux kernel 2.6.25. We are planning to submit our port to the maintainers of the TCPCP2 package, to be integrated into the official TCPCP2 distribution.

The remainder of our ongoing porting effort is directed to migration of the changes to Mobile IPv6 required for Distributed Servers. Distributed Servers uses the Mobile IPv6 implementation to hide migration of the server-side of a connection. The client-side requires no modifications, but the server-side must incorporate changes to the Mobile IPv6 layer to support migration of single connections rather than migration of the entire host across networks. We are currently implementing these changes in the user-level daemon corresponding to Mobile IPv6 for the Linux kernel 2.6.25. Due to the significant changes from the old version for kernel 2.6.8, the port entails significant reimplementation work.

2.2 Interface using Java

Previously, we developed a library interface to Distributed Servers in C++ called Gecko [8]. The Gecko library API provides an object-oriented interface to Distributed Servers that implements client handoff and server management (for example, changing the contact node), but is not callable directly from Java. In support of integration with Virtual Nodes [9], we have extended this interface using the Java Native Interface (JNI) [5] to provide a similar API within the Java programming language. JNI is both necessary and efficient to our purposes. Using JNI allows us to leverage the Gecko library API directly from Java, eliminating significant reimplementation of the API in the Java language. Further, the Java API must be able to interface efficiently with TCPCP2, which provides an API in the C language. Hence, although we are somewhat forced to use JNI, it is actually to our long-term benefit.

Our Java interface to the Gecko library is focused primarily on fault-tolerance integration with Virtual Nodes, providing support for client handoff and recovery in the presence of failures of the contact node and clients' connections. To this end, there are two main objects in the Java framework: *GeckoFramework,* which controls a similar object in the Gecko library that manages the contact node and *GSocket,* which manages a client connection including failover of the connection to a backup (primary-backup management is discussed in the next subsection).

2.3 Membership protocol

Any application using Distributed Servers needs a good control of its own membership. There are two reasons for this: first, to be operational, a distributed server must have at all times one of its nodes designated as the "contact node." This is the node that receives connection requests from new incoming clients, and which selects a node from the distributed server to treat this client. Should the current contact node fail, it is important that another node is quickly elected as the new contact node to take over this important responsibility. Second, to be able to handoff connections to other nodes of the distributed server, the contact node needs to have an up-to-date view of the list of nodes currently present in the distributed server.

The management of membership in any distributed system is a difficult problem. Instead of letting each application programmer be responsible for building a new (suboptimal) implementation each time, we decided to add a standard membership protocol inside Distributed Servers.

In this protocol, nodes of a distributed server are categorized into three categories:

- One "contact node",

- A small (typically less than 5) set of "backups" ready to take over the contact node's responsibilities in case this one should fail, and
- Any number of server nodes (or "snodes") that can simply provide the required service but are not allowed to become contact nodes themselves.

Note that this distinction is orthogonal to the actual query processing capabilities of the respective classes of nodes. In Distributed Servers, any node (including the contact node and the backup nodes) can be selected to process incoming requests.

The membership protocol must provide two functionalities:

- Make sure that at most one node is elected as the contact node. There may be transient periods during which the distributed server will not have any contact node (right after a failure of the current contact node), but these periods should be as short as possible. It is essential that the contact node and its backups have a perfectly consistent view of their mutual membership. The protocol therefore relies on two-phase commit. We consider the associated cost as acceptable, since the two-phase commit need only be executed among the contact node and its backups.
- Make sure that the contact node and all its backups have a reasonably consistent view of the current list of snodes. Snodes do not need to have any view of the group membership. They must only know who the current contact node is. Determining the identity of the contact node is easy: any node may simply send a message to the home address of the distributed server. By definition, the current contact node will receive it and may reply with its own identity.

  It is not vital that the view of snodes membership is perfectly consistent between all backup nodes. We can therefore use a much more lightweight and scalable protocol there.

2.3.1 Contact node election protocol

Joining the set of backups: to join the set of backups, one must send a 'join' message to the current contact node that carries some form of authentication. The contact node assigns a unique identifier to this node by incrementing a counter. It runs a 2-phase commit protocol with all its backup nodes to inform them of the new backup. If one backup node does not respond after a given timeout, then the contact node considers it to be failed and retries a 2-phase commit that contains the information about the new backup and about the failed one. Once a 2-phase commit has succeeded, the new backup is informed that joining was successful.

Backup node failure detection: the contact node does not need to proactively detect the failure of backup nodes. These failures can be discovered later on, upon a join operation of another backup node.

Contact node failure detection: backup nodes are ordered according to their unique IDs. The contact node is always selected as the backup node with the lowest ID. Each backup node monitors the liveness of its predecessor in the list using heartbeat messages. Whenever it detects a failure, it runs a 2-phase commit across all backup nodes to inform them of the change. Note that this also informs the current contact node of such failures. If the failed node happens to be the contact node, then its successor takes over the responsibility as the new contact node.

False positive failure detection: it may happen that a backup node is wrongly accused of having failed. In such a case, it will try to monitor its predecessor in the list. The predecessor can then send a reply message "we consider you to have failed, please join again."

2.3.2 Membership protocol of the snodes

As discussed previously, we need to make sure that the contact node has a reasonably up-to-date view of the snode membership, to be able to handoff incoming connections to them. However, the correctness of the distributed server is not compromised by slight inconsistencies there. Snodes do not need any view of the group membership at all. They only need to know the identity of the current contact node.

Each snode periodically sends a registration message to the contact node (containing authentication information). If the snode was already known by the contact node, nothing happens. Otherwise, the contact node adds it to its list, and sends the update to its backups. A 2-phase commit is *not* necessary.

Note that periodic re-registration is not strictly necessary, and may overload the contact node. Instead, it may be more efficient to re-register only if a snode suspects that it has been forgotten (e.g., it has not been contacted for a given period)

The contact node can detect the failure of snodes when a handoff to them fails. In this case, it can simply remove the failing snode from its list and send the update to its backups (again, no 2-phase commit is necessary).

2.3.3 Current status

The membership protocol has been fully implemented, but has not yet been integrated into the Gecko library framework. This is of course in our immediate agenda.

Authentication of newly joined nodes is currently realized using a simple password known to the whole distributed server. This is obviously not very secure, as a disclosure of this secret may open the distributed server to accept rogue nodes in its membership. Future versions will rely on XtreemOS certificates as a better form of authentication.

3 Reproducible Evaluation

We evaluate Distributed Servers along two main directions: a functional evaluation of the properties provided by Distributed Servers, and a performance evaluation of the different components of Distributed Servers. In the rest of this section, we describe the functional evaluation as a sequence of both advantages and disadvantages to using Distributed Servers. Finally, we end with an experimental performance evaluation that provides quantitative breakdown of the overheads incurred in using Distributed Servers.

3.1 Functional evaluation

The functional evaluation of Distributed Servers concerns the different benefits and limitations of using Distributed Servers.

3.1.1 Client-side Mobile IPv6 Support

Distributed Servers assume that client-side operating systems support the functionality of a Mobile IPv6 (MIPv6) correspondent node. This is already true for many popular operating systems, including Linux and Windows. However, it might still happen that some potential service clients do not support MIPv6: for example, when services are hosted on the grid for clients external to the grid. While grid service running Distributed Servers can support a very large group of MIPv6-enabled clients, the service should also be able to support a small number of MIPv6-disabled clients.

Although client-side MIPv6 support is necessary to hand off clients among member nodes using route optimization, 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.

3.1.2 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 Distributed Servers via a single distributed server address might lead to problems when the server decides to hand off any of these connections. 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 distributed server address, either all the connections of a given client must be handed off simultaneously to the same acceptor, or none at all. This can be a disadvantage if the service needs to handoff different connections from the client to different server nodes.

This limitation can be alleviated if the service uses multiple distributed server addresses. As each translation binding is associated with only one distributed server address, it does not affect the traffic sent to other addresses. Provided that the client-side application opens simultaneous connections to different distributed server addresses, the service can hand off each of them just like non-parallel connections. A good rule of thumb is thus that each service running Distributed Servers should have a different distributed server address even if the services are hosted initially at the same contact node.

3.1.3 Fault-tolerance

In a large-scale grid service deployment, any node can fail. In that case, it is the responsibility of the application to transfer the application-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 such application-level state information. Without this state information, the service may be forced to close the connection after recovering it. Such unexpected connection closing may result in the service appearing to be unreliable.

We address this problem with the use of another XtreemOS technology—Virtual Nodes. The use of Distributed Servers and Virtual Nodes is described in more detail in deliverable D.3.2.10 [9]. In brief, Virtual Nodes provide a method to replicate the state of a server's connections across a small number of other service nodes. Should a server fail, a replica 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 data that has been received and acknowledged by a server node's TCP layer after the replicated connection state was updated for the last time. This is because the server cannot transparently force the client to retransmit the already-acknowledged data. On the other hand, ensuring that all data is replicated before sending an acknowledgment to the client can dramatically increase the latency seen by the client. Defining the tradeoff tends to be application-specific, as applications themselves might provide some degree of resilience to sudden service outages.

3.1.4 Scalability

The scalability of any XtreemOS service is great concern for very large grids. Distributed Servers supports client handoff across the wide-area to support an extremely large number of clients possibly with each client experiencing multiple handoffs to access better server locations. Unlike most server-side implementations of load-balancing, Distributed Servers can distribute load across the wide-area while still maintaining client application transparency. The contact node appears to be the only bottleneck in the system. However, a service can combine Distributed Servers with other load-balancing approaches such as round robin DNS to provide scalable support for many contact nodes. Round robin DNS provides different addresses (that is, contact nodes) for the service to different clients to effect load-balancing. With many contact nodes, we do not foresee any current bottlenecks in the Distributed Servers service. Implementing the application logic for the service remains a concern that is not in the scope of this document.

3.1.5 Gecko Library and Java API

Both of the supported libraries for using Distributed Servers—the Gecko library interface and the new Java interface—provide a convenient API although the Gecko C++ library is more complete. Client connections can be managed with flexible distribution policies. The Gecko library API includes an object-oriented interface to client handoff and server management and provides a proxy to allow client distribution without modifying server. The Java API provides for recovery of client connections between a primary and a backup. For a complete description of the Gecko library API, please see Deliverable D3.2.2 [8].

3.2 Performance Evaluation

We evaluate the performance of Distributed Servers using a simple testbed (see Figure 1). The core of that testbed is a NISTnet router, which connects the client

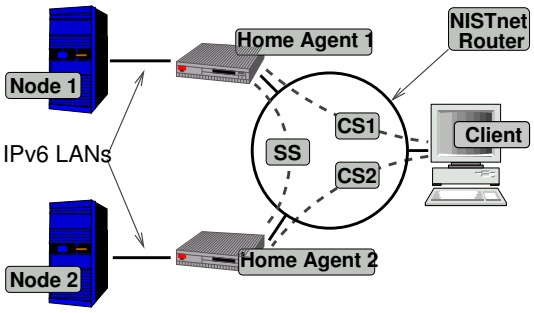


Figure 1: Testbed topology

machine to a service infrastructure [7]. 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 (as seen in Fig. 1): the *SS* tunnel controls packet transmission between the member nodes, and the *CS1* and *CS2* tunnels 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 [6]. All the machines are equipped with PIII processors, with clocks varying from 450 to 700 MHz.

3.2.1 Server Access Latency

The distributed server 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 distributed server 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 Distributed Servers uses route optimization to enable direct communication between the service's 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.

3.2.2 Handoff Time Decomposition

Distributed Servers enables a service's nodes to hand off transparently 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 (see Fig. 1). Node 1 transfers 500 kB of data, and hands off the connection to Node 2—changing the server endpoint of the client's connection—immediately after the last `send()` call returns. Node 2 then sends another 500 kB of data to the client and closes the connection. In this scenario, we call Node 1, where the server endpoint lies before handoff, the *donor* and Node 2, where the server endpoint lies after handoff, the *acceptor*.

The total handoff time can be divided into seven phases (see Table 1) that are discussed in more detail in [10]. 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 disci-

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

Table 1: Handoff time decomposition (without NISTnet delays). Phases are discussed in detail in [10].

pline 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 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 Binding Update message, and activating the socket 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 Binding Update and Binding Acknowledgment 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.

3.2.3 State Transfer Optimization

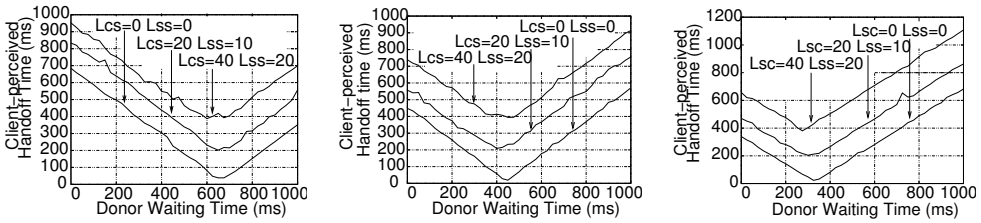
The previous experiment shows that most of the handoff time is spent transferring the socket state from the donor to the acceptor. The reason why that transfer takes so long is that in this experiment the donor extracts the socket immediately after the last `send()` call returns. This means that the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

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 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 2.

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.



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

Figure 2: Client-perceived handoff times for various upstream node connection bandwidths

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 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. The conclusion holds even as buffer sizes increase for higher bandwidth networks between the client and donor because the client-perceived handoff will still be limited to the one-way latency between the donor and acceptor.

4 Conclusion

As we have seen, the Distributed Servers platform is now mature enough to support transparent access to distributed groups of server nodes. Performance evaluations show that Distributed Serverss are efficient: they allow for direct communication between a client and the server node that is currently serving it. Handoff times, as perceived by the client, can be reduced to an acceptably low value. Furthermore, we foresee no scalability bottleneck when using Distributed Servers to serve massive numbers of connections.

The major limitations of Distributed Servers are known: first, Distributed Servers rely on the assumption that a large fraction of client machines have native support for MIPv6. Software support for MIPv6 is not a major issue, as MIPv6 has been implemented in all major operating systems. However, to be able to use MIPv6 easily, client machines ought to be connected to an IPv6-enabled network. This risk is well-identified, and systematically reported in the work package quarterly reports. Second, porting Distributed Servers to a new version of the Linux kernel requires a non-trivial amount of work. We are currently busy porting Distributed Servers to Linux kernel version 2.6.25; simultaneously, WP2.1 is working on the porting of XtreemOS foundations from 2.6.20 to 2.6.25. As soon as this

upgrade is completed, we should be able to integrate Distributed Servers into the XtreemOS distribution.

References

[1] Werner Almesberger. TCP connection passing. In *Ottawa Linux Symposium*, July 2004.

[2] NTT Corporation. TCP connection passing 2. Available on WWW, 2006. `http://tcpcp2.sourceforge.net/`.

[3] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6). RFC 2460, December 1998.

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

[5] Sheng Liang. *The Java™ Native Interface: Programmer's Guide and Specification*. Addison-Wesley, June 1999.

[6] MIPL: mobile ipv6 for linux. Available on the WWW, July 2006. `http://www.mobile-ipv6.org/`.

[7] The NIST net network emulator. Available on the WWW, July 2006. `http://www-x.antd.nist.gov/nistnet/`.

[8] Guillaume Pierre. First prototype version of ad hoc distributed servers. XtreemOS deliverable D3.2.2, November 2007.

[9] Guillaume Pierre, Jeff Napper, and Jörg Domaschka. On the feasibility of integration between distributed servers and virtual nodes. XtreemOS deliverable D3.2.10, November 2008.

[10] Michał Szymaniak, Guillaume Pierre, Mariana Simons-Nikolova, and Maarten van Steen. Enabling service adaptability with versatile anycast. *Concurrency and Computation: Practice and Experience*, 19(13):1837–1863, September 2007. `http://www.globule.org/publi/ESAVA_ccpe2007.html`.