Project no. IST-033576

XtreemOS

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

First Prototype Version of Ad Hoc Distributed Servers
D3.2.2

Due date of deliverable: November 30^{th}, 2007
Actual submission date: November 30^{th}, 2007

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

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

Responsible institution: VUA
Editor & and editor's address: Guillaume Pierre
Vrije Universiteit
Dept. of Computer Science
De Boelelaan 1083
1081HV Amsterdam
The Netherlands

Version 1.0 / Last edited by Guillaume Pierre / November 28, 2007

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
0.0	2007/10/12	Guillaume Pierre	VUA	Initial document skeleton
0.1	2007/10/30	Guillaume Pierre	VUA	More content added
0.2	2007/11/06	Guillaume Pierre	VUA	Integrate comments from internal review
0.3	2007/11/10	Guillaume Pierre	VUA	Integrate more comments from internal review
1.0	2007/11/30	Guillaume Pierre	VUA	Add some minor precisions

Reviewers:
Yvon Jégou (INRIA) and Luis Pablo Prieto (TID)

Tasks related to this deliverable:

Task No.	Task description	Partners involved
T3.2.1	Design and implementation of ad hoc distributed servers	VUA*, ULM

*task leader

Contents

Executive summary

Workpackage 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, 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 to present a group of server processes to its clients as a single entity. 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 internals of our distributed servers implementation. Three levels of implementations are being realized.

Low-level interface The low-level interface provides the finest level of control to the server application programmers. However, it requires a detailed understanding of the internals of Mobile IPv6. We do not envisage that regular programmers would use this interface, except in exceptional cases.

High-level interface The high-level interface provides a C++ interface that normal programmers can use to integrate distributed servers into their own server applications. This requires changing the server application code to a certain extent, but it is easier to use than the low-level interface.

Distributed server proxy This proxy allows to use partial functionality from the distributed servers with applications totally unaware of distributed servers. This is in the form of a TCP proxy that sits in front of each application node, and does connection redirection automatically according to some redirection policy. The proxy does not require any changes to the server application, and it also supports server applications written in any programming language. On the other hand, a few functionalities from distributed servers require active participation from the application, and are thus not available.

1 Introduction

Workpackage 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, 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 to present a group of server processes to its clients as a single entity. 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 internals of our distributed servers implementation. All code is available in the `xtreemos` Subversion repository under directory `WP3.2/distr-server`. This work has been realized at VUA.

This deliverable describes distributed servers using a bottom-up organization. Section 2 briefly summarizes Versatile Anycast, the underlying technology that distributed servers are using. Then, Section 3 describes the low-level implementation of distributed servers that is intended for advanced programmers only. Section 4 describes the high-level interface that regular programmers are expected to use. Section 5 discusses a distributed server-enabled proxy that allows one to add partial distributed servers functionality to distributed servers-unaware programs. Section 6 discusses kernel-level requirements of distributed servers technology. Finally, Section 7 concludes.

2 Essentials on Versatile Anycast

Distributed servers rely for their operation on an underlying technology called Versatile Anycast. Although it is not the purpose of this document to fully detail the internals of versatile anycast, we summarize its features and mechanisms here. For an in-depth discussion and performance evaluation of versatile anycast, we refer the reader to [9].

2.1 Features

Versatile anycast uses the Mobile IPv6 protocol [6] to present a group of physical machines as a single entity to its clients. The group can be composed of any number of server machines located anywhere around the world. Clients need not know anything about the current composition of the group. They are provided with a single, stable IPv6 address that they can use to contact the group of nodes. Similarly, no non-standard feature is required from the home agent. The only machines that need to use specialized software are the server nodes themselves.

Versatile anycast subsequently provides the following features:

- Make sure that all network traffic addressed to the versatile anycast address reaches one specific node out the targeted group. This node is referred to as the contact node.

- Allow a group of nodes to change its contact node. This may happen, for example, upon a failure of the current contact node.

- Allow the contact node to *handoff* network traffic issued by one of its clients to another server node from the group. This operation may be realized upon the initiation of a client connection, or at any moment later on. Communication is not tunnelled through the contact node, but exploits the route optimization mechanism from MIPv6 to provide a direct network path between the client and its current server node. Handoffs are totally transparent to the client-side application.

- It is not sufficient to handoff the flow of IP packets from the client to one specific server node. To support mid-connection handoff, one must be able to extract the latest state of the TCP socket at the server donor side, and transfer it to the server acceptor side so that the TCP connection is not disturbed. Versatile anycast relies on the TCPCP Linux module for this [1].

- In the event that a server node would crash, it is possible for another node to recover the connections that the crashed machine had with its clients. This

operation is again transparent to the client application; the only consequence from the client side is a short interruption of traffic that corresponds to the necessary delay for the backup node to detect the failure of the crashed server node.

This last feature is not discussed in [9], as we developed it after the submission of that article. We therefore discuss it in more detail next.

2.2 Connection recovery upon node failure

To support connection recovery upon a server node failures, several techniques must be used together. First, a number of preparatory operations must be realized prior to node failure. Each node must have one designated backup, which must monitor its availability to decide when it should recover connections. It is also necessary that each node informs its backup each time it opens or closes a connection. To allow for TCP state reconstruction after failure, the backup node must be sent a copy of the state of the TCP connections to recover as well as the MIPv6-level biding update sequence number. The TPC state is very small (90 bytes), and need *not* be maintained up-to-date frequently. Mirroring the state of an open TCP socket may be realized for example once every few megabytes of transferred data to the client.

Once a node failure is detected, the backup node can recover the connections from the failed node as follows:

- Create a new frozen socket based on the latest state received from the failed node.
- If the failed node was having the role of contact node: contact the home agent, and take-over the IP address of the failed node (note that this requires that the failed node shared its IPsec security association [5] with its backup prior to the failure).
- Contact the client, and initiate an IP handoff toward the backup node.
- We must now recover the current state of the TCP connection. To do so, the backup node must open a raw socket and apply the following procedure:
  - Send a set of empty TCP ACK packets to the client based on the last known state. Only one such packet is really necessary, but one should send a few to account for possible packet drops in the network.
  - Wait for one ACK packet from the client. According to the TCP specification, a client receiving an old (but valid) ACK packet must reply with an up-to-date ACK.

  - From the received ACK packets, one can derive the current TCP sequence numbers.
  - Once this is done, the raw socket can be closed as it is not necessary any more.

- The backup can now update its frozen socket with the new TCP sequence numbers derived from the client's ACK, and adjust its streaming offsets accordingly.

- The backup can inform the application of the current offset within the connection. For example, a connection may have sent 12077 bytes of payload and received 1021 bytes of payload since the connection was created. It is the responsibility of the server-side application to interpret these offsets to resume the failed session. We expect that this will be possible in several cases. At worst, the application can cleanly reset the connection to the client, and expect that the client will re-issue its invocation.

- The backup can now re-activate the socket, and use it normally.

3 Low-Level API

The low-level API allows one to manipulate versatile anycast features directly. It provides two sets of functions: the first one handles the management of MIPv6 addresses and their registration with the MIPv6 home agent, while the second one handles handoffs.

Application programmers can use these functions to create a Versatile Anycast based distributed server. However, this API does not provide any support for sharing IPsec security associations, managing the membership of the distributed server, or facilitating TCP connection handoffs. In particular, it expects the programmer to interleave versatile anycast and TCPCP calls correctly as described in [9].

3.1 Address management functions

The first necessary operation at the startup of a distributed server is to create a shared IPv6 address which can be given out to the clients as a contact address. Each node from the distributed server must recognize this address as its own, and attach it to its network interface so that the server-side application can bind sockets to it. Secondly, a MIPv6 binding cache entry must be created to map the IPv6 address of each replica to that of the whole distributed server. These two actions are accomplished using the following function:

```
mip6_addr_passive(struct in6_addr *global_address,
                  struct in6_addr *local_address,
                  struct in6_addr *current_contact_node);
```

The specified `global_address` will be used by clients to connect to the system, while the `local_address` will be used when a client is handed off to this node. The address of the current contact node is required to perform handoffs, as internal messages necessary for the handoff need to be forwarded by the contact node.

This operation is entirely internal to each node of the distributed server. This means that each node is *ready* to accept traffic targeted to the distributed server, but network routes are not setup yet for read distributed server operation.

One node from the distributed server must now take the role of contact node, which means that it will receive the first packet of each newly created connection. The contact node must register itself to the home agent which controls the versatile anycast address. This registration is realized using the function:

```
mip6_addr_active(struct in6_addr *global_address,
```

```
                    struct in6_addr *local_address_for_handoffs,
                    struct in6_addr *local_address_as_contact_node,
                    struct in6_addr *home_agent);
```

The distinction between both local addresses is not absolutely necessary: the same local address can be used as both the local address of the contact node of the distributed server, and the local address of the same node in its role of a potential recipient of a handoff. Distinguishing these two addresses however allows to optimize the connection establishment procedure.

3.2 Handoff functions

A second set of functions provides means to execute an IPv6 handoff between two replicas. When a *donor* node from a distributed server initiates a handoff to an *acceptor* node from the same distributed server, the following operations must be realized:

First, the binding updates sent to the client contain a sequence number. This means that the donor needs to request the sequence number of the most recent binding update from its local MIPv6 daemon:

```
mip6_handoff_mark(struct in6_addr *client_address,
                  struct in6_addr *global_address,
                  struct in6_addr *local_address_for_handoffs,
                  struct in6_addr *acceptor_address);
```

This sequence number must then be sent to the acceptor. The acceptor can then start the handoff procedure. This procedure involves a so-called return routability check defined in the MIPv6 specification: the acceptor sends a "CoTI" message directly to the client, and a "HoTI" message to the current contact node, as specified with the `mip6_addr_passive` function[1]. The latter then forwards the "HoTI" message to the home agent, so it can be sent to the client. Upon receipt of both messages, the client replies as normal, sending the "CoT" directly, while the "HoT" is forwarded by the home agent and contact node. After the reconstruction of the binding key, the acceptor sends a binding update message to the client using the sequence number sent by the donor. These actions are executed by the acceptor using the function:

[1]HoTI, CoTI, HoT and CoT messages are defined by Mobile IPv6 [6]. They implement the so-called "return routability check", which allows the client node to verify that a proposed route optimization leads to the same server node as when using the current route.

```
mip6_handoff_start(struct in6_addr *client_address,
                   struct in6_addr *global_address,
                   struct in6_addr *local_address_for_handoffs,
                   struct in6_addr *donor_address,
                   int              sequence_number);
```

A possible optimization is, instead of executing both the return routability and binding update in sequence, to delay the latter. This allows one to execute the return routability in advance, while the donor is still in control of the client, allowing it to first finish up and empty its sending buffers. The acceptor then first executes a function that executes the return routability, after which a second function is executed for sending the binding update to the client. This optimization reduces the total handoff time because less time is used on procedures after the donor has stopped serving the client, allowing the acceptor to take over the client faster. Experiments show that the total handoff time can be reduced by roughly one third [9]. However, it requires extensive synchronization between the donor and acceptor, making it harder to use.

After the handoff has taken place, the donor can close its connection to the client, which is not needed any more.

As the binding between the global and local distributed server addresses at the client side is realized on a per-address basis, closing the connection won't affect the binding. In case the client closes its connection to the server and opens another one, it will be served by the last node it was bound to (provided that the binding has not expired). For some applications, this behavior may be undesirable. The server application can then use the following function to clear the binding at the client side after closing the connection:

```
mip6_handoff_clear(client_address, contact_handle);
```

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

3.3 TCP handoff

The procedure presented above only hands off the flow of IP packets. In case the application is connected to the client using TCP, then extra steps must be taken to extract and transfer the state of the TCP connection between the donor and the receiver. These steps are discussed at length in [9], so we only briefly summarize them here.

1. At the donor: freeze the socket, extract its state from the kernel;

2. Send the TCP state to the acceptor;

3. At the acceptor: re-create a socket from the received state, keep it frozen;

4. At the acceptor: initiate the return-routability procedure and the binding update;

5. At the acceptor: initiate the binding update to the client;

6. At the acceptor: send an acknowledgement to the donor to notify that the handoff was successful;

7. At the acceptor: re-activate the socket;

8. At the donor: close the frozen socket.

3.4 Test cases

As part of our dissemination activities, we built and videotaped a demonstration of versatile anycast based on the low-level interface. The demo showcases a group of two servers in different network segments using versatile anycast, and that can stream video over HTTP. These servers are queried by one client, which is using an unmodified standard video client to view the video. The demonstration videos are available online at [3], and are organized as follows:

1. **Presentation of the experiment:** presentation of the testbed and the streaming video server application.

2. **Simple TCP handoff:** the client connection is handed-off between the two servers once every 15 seconds. The only way to make the handoff visible at the client side was to organize servers such that one would deliver a colorful movie while the other would deliver a monochrome version of the same movie.

3. **Handoffs can take place at any moment:** there is no need for complex preparation procedures before a handoff can take place. Here, handoffs are manually triggered by the operator.

4. **Connection recovery upon server crash:** we simulate the crash of one server by physically disconnecting its network cable. From the client side, we notice a 1-second interruption in the movie, then the streaming resumes as normal.

5. **Connection recovery with client-side buffering:** in this particular application, one can even hide the 1-second streaming interruption due to a node failure by using standard client-side data buffering techniques.

4 High-Level API

The low-level API presented in Section 3 provides all the necessary functionality to build a distributed server application. However, doing so is quite difficult, and many important details must be handled by the application programmer: request IPv6 addresses from the home agent, manage the membership of the distributed server, monitor each node's availability, select a contact node, share the IPsec security association with potential contact node backups, define policies for deciding which connection should be handoff to which server node, implement the actual handoff procedure, etc. Many of these requirements are in fact common to all distributed server applications. We therefore built a higher-level interface that implements these functionalities, and presents a simpler API to the programmer. The high-level API is implemented in C++.

Note that this API is tentative, and likely to be changed in future versions according to discussions with WP3.1.

4.1 Features

The high-level API allows one to define the configuration of a distributed server thanks to a simple configuration file. This file contains information about the global address of the distributed server, the address of the home agent, the identity of the contact node and its authorized backups, and various other internal parameters. A programmer can then build a distributed server using the following functionalities:

- All distributed server functionality is available through a `GeckoFramework` object that each node of the distributed server must instantiate. Upon instantiation, this object creates a background thread which is used to handle the communication and handoff between the distributed server nodes.

- In the simplest instance of distributed servers, one can simply define a handoff policy which is invoked each time a new client contacts the distributed server. The current implementation only contains a "random" policy, but in future versions we will provide more policies to choose from, such as a round-robin policy, or more advanced ones such as a proximity-based policy which could redirect a client to the closest distributed server node. An application programmer can also implement custom policies by simply deriving a base class and implementing a selection method.

- The distributed server framework provides a "socket" class which is a wrapper around a normal TCP socket. This is necessary to provide a simple handoff interface, and programmers are requested to use this wrapper rather

than the actual socket. In particular, this class contains a `handoff` method which allows to explicitly handoff a connection (either upon connection establishment or while data are being exchanged) to another member of the distributed server.

The current framework focuses on the most essential functionalities. Additional functionalities may be added in the future, according to actual needs and available workforce:

- Dynamically add/remove nodes to/from a distributed server
- Proactively detect node failures, and recover the failed connections
- Support recovery upon failure of the contact node

4.2 API

4.2.1 GeckoFramework

The main class of the high-level API is `GeckoFramework`. An object of this class must be instantiated by every member node of the distributed server.

```
class GeckoFramework {
public:
  GeckoFramework          (struct in6_addr *local_address,
                           char *config_filename,
                           char *network_interface);
  virtual ~GeckoFramework();
  int SetPolicy           (GeckoPolicy *policy);
  int GetServerSocket     (GeckoServerSocket **serversock);
  int JoinReplicaNetwork  ();
  int LeaveReplicaNetwork();
};
```

Using this object, a programmer can now carry two more operations: (optionally) setup a redirection policy, and create a server socket to receive client connections. Note that, if a redirection policy is defined, the redirection will be realized automatically by the background thread, transparently to the application running at that node.

4.2.2 GeckoPolicy

A redirection policy is implemented by deriving the `GeckoPolicy` base class.

```
class GeckoPolicy {
public:
  GeckoPolicy                       (GeckoFramework *framework);
  virtual ~GeckoPolicy              ();
  virtual int SelectTarget          (GeckoSocket *client,
                                     struct in6_addr *target);
  virtual void NotifyFailedHandoff(struct int6_addr *target);
};
```

To instantiate a new policy, one must overload the `SelectTarget()` method, which fills in the `target` variable with the address where the handoff should take place, given the identity of the client socket. One must also overload the `NotifyFailedHandoff`, which is called by the framework in the case a handoff failed, and which gives the redirection policy an opportunity to take note the failure and remove the concerned node from potential redirection targets.

The current implementation contains a single policy, which selects a target randomly among all server nodes.

4.2.3 GeckoServerSocket

A server socket object simply allows the application to call an `accept()` method, which blocks the caller until a client connection is received. Note again that, if a redirection policy is defined, only the server node that should actually process the connection will accept a new socket; the operation is transparent to the application running at the contact node (unless the contact node is selected to process the connection, of course).

```
class GeckoServerSocket {
  GeckoServerSocket (GeckoFramework *framework);
  virtual ~GeckoServerSocket();
  int Close          ();
  int Accept         (GeckoSocket **client_socket);
};
```

Upon an established connection, the application receives a `GeckoSocket` object that can be used to communicate with the client. This can happen in different situations:

- When a client initiates a new connection, and the redirection policy decides that the contact node should handle the connection;

- When a client initiates a new connection, after being handed-off to another server node;
- During an existing connection, after a server node hand-off the connection to another node.

4.2.4 GeckoSocket

```
class GeckoSocket {
  GeckoSocket            (GeckoFramework *framework);
  virtual ~GeckoSocket   ();
  int     Bind           (int sd);
  int     Connect        (const struct in6_addr *target_address,
                          int target_port);
  int     SetNonBlocking();
  int     SetBlocking    ();
  bool    IsBlocking     ();
  ssize_t Write          (const void *buf, size_t nbytes);
  ssize_t Read           (void *buf, size_t nbytes);
  void    Close          ();
  void    GetPeerName    (struct sockaddr_in6 *in);
  int     Handoff        (bool nonblocking);
  int     Handoff        (struct in6_addr *target,
                          bool nonblocking);
};
```

The `GeckoSocket` contains the expected methods to read/write data, set into blocking or non-blocking mode, etc. Additionally, it contains two extra methods `Handoff`. The first one will handoff the connection according to the current redirection policy, while the other allows to programmer to specify a handoff target explicitly.

The `nonblocking` parameter allows the programmer to decide whether the application should wait until handoff is complete, or if the handoff should take place in the background. In the second case, a handoff failure would lead to an automatic selection of another handoff target according to the local redirection policy.

4.3 Test cases

Although the high-level API implementation is still relatively experimental, it is already possible to implement applications that use it. For testing purpose, we built a tiny "echo" server application, which automatically hands off connections

using the “random” policy upon connection establishment. It also hands off the connection again each time it reads a ’H’ character in its socket input. We successfully deployed this application in the MIPv6 testbed at VUA, and also within one server cluster from Grid’5000 in Rennes. We could not deploy this application over more locations from Grid’5000 because IPv6 connectivity is not (yet) available between Grid’5000 sites.

5 Distributed Server Proxy

Programmers can easily integrate distributed server functionality into their server applications using the high-level API presented in Section 4. However, using this API can also have drawbacks. First, it is currently available only for C++ programs. Programs written, for example, in Java currently cannot benefit from distributed servers functionality. Also, although the high-level API should be fairly easy to understand and to use, it does require that programmers adapt the source code of their applications.

Consequently, we built an even higher-level version of distributed server technology, in the form of a distributed server-enabled proxy. This implementation provides functionality limited to handoff at connection establishment time according to one of a few pre-defined policies. No mid-connection handoff nor connection recovery upon failure is supported. On the other hand, this implementation does not impose any change whatsoever to a pre-existing server application. It also works with programs written in any language.

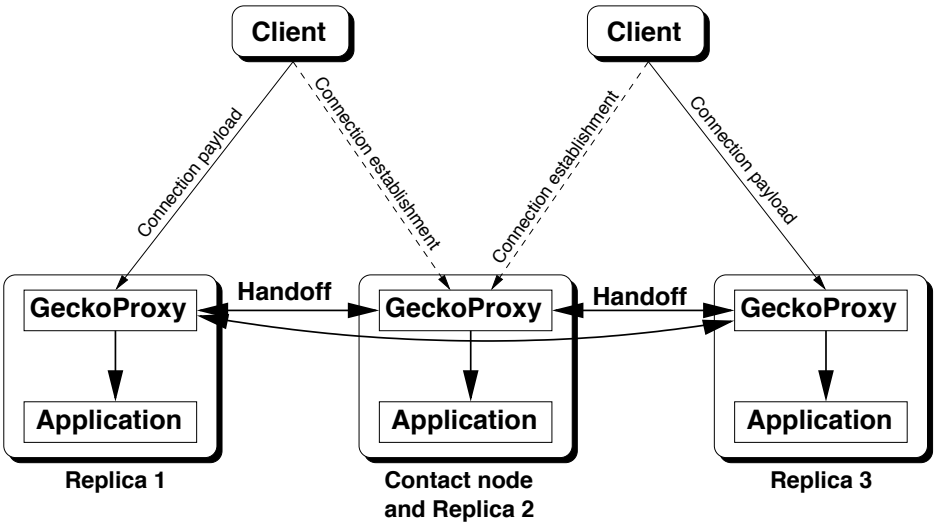

Figure 1: Design of the Distributed Server Proxy

The design of the Distributed Server Proxy as shown in Figure 1 was inspired by Stunnel, a proxy that "allows you to secure non-SSL aware daemons and protocols (like POP, IMAP, LDAP, etc) by having Stunnel provide the encryption, requiring no changes to the daemon's code [8]." One instance of the proxy sits in front of each server application. The proxies collectively implement distributed server functionality: one of them takes the role of contact node, and hands off incoming connections to other proxies (or possibly itself), according to a pre-defined redirection policy. The acceptor proxy can then open a regular TCP connection

to its local application, and forward all client traffic between the server application and the client. This way, connections can be handed off between distributed server-unaware server nodes.

The Distributed Server proxy has an extremely simple command-line interface:

```
geckoproxy -g <config_file> -a <local_application_port_number>
```

The `config_file` parameter specifies a configuration file for the distributed server, as discussed in section 4. The `local_application_port_number` parameter specifies the port number that the local distributed server-unaware application is using.

Note that this implementation reduces the functionality available thee the application: it does not allow to initiate content-aware handoffs, mid-connection handoffs, nor connection recovery upon node failure.

6 Operating System Requirements

Distributed servers require special support from the MIPv6 implementation running on the replica nodes. The current system relies on the MIPv6 version from [7]. This implementation puts only the absolutely necessary infrastructural support in the kernel, moving most functionality to a daemon running in user space. The daemon is then responsible for installing and removing packet translation mappings required to handle the special MIPv6 routing headers. This MIPv6 implementation therefore consists of two parts: first, a number of patches add the minimum number of necessary hooks into the Linux kernel. The largest part of the implementation resides in a user-level MIPv6 daemon, which in turn makes use of the kernel hooks. This kernel patch is however tightly linked to a specific version of the kernel. The latest available version provides hooks for the Linux-2.6.16 kernel. Newer kernel versions are not supported for the moment.

Another pitfall of the current MIPv6 implementation is that it does not support IPsec security associations to prevent hostile address takeovers by unauthorized parties [2].

Distributed servers also need to be able to freeze, unfreeze, extract and inject TCP socket states from/into the kernel. For this, we use the TCPCP Linux extension, that we modified to support IPv6 connections. TCPCP only runs on Linux-2.6.11, which is the kernel version we are currently using in our experiments.

Clearly, being stuck with the old linux-2.6.11 kernel is not a desirable situation. One partial solution could be to use TCPCP2, a fork from TCPCP which natively supports IPv6 sockets and the Linux-2.6.15 kernel [10]. Similarly, it might be wise to change our basic MIPv6 implementation for example to FMIPv6, which supports the current Linux-2.6.23 kernel [4].

7 Conclusion

We believe that distributed servers have the potential to provide XtreemOS with a unique feature not found in any other system: allow to make the internal composition of a distributed service transparent to its clients, even if the service is deployed across multiple locations. The handoff mechanisms provides high-performance client-to-server communication, and fine-grained control of the redirection to the servers themselves.

Next to the powerful but cumbersome low-level interface of distributed servers, we have built a high-level interface that allows programmers to easily integrate distributed server features into their services. We also built a proxy that allows distributed server-unaware programs to benefit from basic distributed servers functionality.

To allow distributed servers to reach their full potential within XtreemOS, it is now time to work in collaboration with other partners and integrate this technology with other XtreemOS components:

Within WP3.2 Another innovative feature developed within WP3.2 is *virtual nodes* which, as described in deliverable D3.2.5, offer applications with automatic replication for fault-tolerance. The features from distributed servers and virtual nodes nicely complement each other to provide transparent fault-tolerant Grid applications: virtual nodes make sure that redundant processes are always ready to takeover each other's tasks in a highly-available process group, while distributed servers may allow one to completely hide node failures and the subsequent request redirection to the clients. This integration must however be considered as a difficult research topic, and will require significant research and development efforts.

With WP3.1 Now that functionality from distributed servers is getting stabilized, it is time to interact with WP3.1 to define an API consistent with the rest of XtreemOS. We plan to use the API presented in Section 4 as a starting point, and to refine it to match the common coding style of XtreemOS.

With WP2.1 and WP2.2 Distributed servers create requirements with respect to the Linux kernel, as discussed in Section 6. Clearly, the current requirement of a (modified) 2.6.11 Linux kernel is not a viable option. We plan to work in collaboration with WP2.2 to define reasonable targets, and to port the existing system to newer kernels.

With WP2.3 WP2.3 works on the XtreemOS version for mobile devices. As such, this work package has expressed interest in support of mobile IPv6

and distributed servers. We plan to discuss extensively with WP2.3, in particular to define which implementation of MIPv6 on Linux should be used by our two workpackages.

With WP3.3 and WP3.5 WP3.3 and WP3.5 are two potential users of distributed servers and virtual nodes technology, to build highly available job submission and security services. We however believe that it is still too early to directly integrate virtual node and distributed servers technology from WP3.2 with these services. Such integration is planned in the longer term.

References

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

[2] J. Arkko, V. Devarapalli, and F. Dupont. Using IPsec to protect mobile IPv6 signaling between mobile nodes and home agents. RFC 3776, June 2004.

[3] Distributed servers demonstration. `http://www.cs.vu.nl/~gpierre/mipv6/` or `http://www.xtreemos.eu/science-and-research/plonearticlemultipage.2007-05-03.8487028755/distributed-servers-for-transparently-distributed-services`.

[4] FMIPv6. `http://www.fmipv6.org/`.

[5] IPsec. `http://en.wikipedia.org/wiki/IPsec`.

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

[7] MIPL mobile IPv6 for linux. `http://mobile-ipv6.org/`.

[8] Stunnel – a universal ssl wrapper. `http://www.stunnel.org/`.

[9] 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.

[10] TCPCP2. `http://tcpcp2.sourceforge.net/`.