Project no. IST-033576

XtreemOS

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

On the feasibility of integration between distributed servers and virtual nodes
D3.2.10

Due date of deliverable: November 30th, 2008
Actual submission date: December 4th, 2008

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP3.2
Task number: T3.2.1, T3.2.4

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 Guillaume Pierre / December 4th, 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	27/10/2008	Jörg Domaschka	ULM	Draft of architecture section
0.3	27/10/2008	Jeffrey Napper	VUA	Draft of challenge/limitation section.
0.4	27/10/2008	Jörg Domaschka	ULM	Draft of status section
0.5	28/10/2008	Jeffrey Napper	VUA	Reorganized and edited document
1.0	31/10/2008	Guillaume Pierre	VUA	Wrap-up for internal review
1.1	7/11/2008	Jeffrey Napper	VUA	Edited according to comments from Christine Morin
1.2	4/12/2008	Guillaume Pierre	VUA	Edited according to comments from Arnaud Laprévotte

Reviewers:
Christine Morin (INRIA) and Arnaud Laprévotte (Mandriva)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.2.1	Design and implementation of distributed servers	VUA*
T3.2.4	Design and implementation of a virtual node system	ULM*

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

Executive Summary

Two different services developed within WP3.2 have seemingly complementary features. On the one hand, Virtual Nodes allow one to replicate a Java-based service for fault-tolerance. On the other hand, Distributed Servers are designed to make the distribution of a resource totally transparent to its clients. From a conceptual point of view, both services are highly complementary: virtual nodes provide fault-tolerant replication that is mostly transparent to the service developer, but lacks an access method that makes fault-tolerance transparent to the clients; distributed servers provide a solution for making the service replication transparent to the client.

An important design goal is to keep the architecture of the integrated Virtual Nodes and Distributed Servers as independent as possible from the middleware protocol used to access the replicated service. We therefore propose to encapsulate the middleware functionality on the server side in a *protocol adapter* in Virtual Nodes that handles incoming connections according to the communication protocol used by the middleware layer. The client's middleware layer remains unmodified. Distributed Servers can then be used to transparently fail-over the client's middleware layer to a different replica on the server side if the client's current replica fails.

We further discuss the integration of both services, the difficulties to be addressed, and the progress of the current work toward an integrated system merging Virtual Nodes and Distributed Servers. The structural challenges include interfacing the two services written in different languages and the different uses of membership services between Distributed Servers and Virtual Nodes. We also discuss the challenge for Virtual Nodes to recover from faults using Distributed Servers. Finally, progress on the integration is such that a wide part of the JNI integration layer has now been implemented and tested. Virtual Nodes can thus start and steer a Distributed Servers instance.

When this work is available, we plan to promote its use within XtreemOS so that mission-critical services can run in a fault-tolerant environment provided by Distributed Servers and Virtual Nodes.

Contents

1 Introduction **3**

2 Overview **4**
2.1 Distributed Server Basics 4
2.2 Virtual Nodes Basics 5

3 Challenges and Restrictions **6**
3.1 Interfacing Challenges 6
3.2 Recovering from Faults 7
3.3 Membership Views 7

4 Expected Architecture **8**
4.1 Virtual Nodes Configuration 9
4.2 Distributed Servers Configuration 9
4.3 Runtime Behavior 10

5 Current Status and Open Issues **11**

6 Conclusion **11**

1 Introduction

Two different services developed within WP3.2 have seemingly complementary features. On the one hand, Virtual Nodes allow one to replicate a Java-based service for fault-tolerance [2]. Replication is transparent to the service programmer, provided that a few guidelines are respected. Fault-tolerant replication allows the client-side programmers not to worry about possible failures of the service. However, this relies on the use of specialized software at the client side, in order to handle the switch-over from one replica to another in case of a node failure. While this is acceptable in a closed environment, this characteristic makes virtual nodes unsuitable for open systems such as Web services, where the client-side software may use any off-the-shelve implementation of the communication protocol.

On the other hand, Distributed Servers are designed to make the distribution of a resource totally transparent to its clients [7]. Distributed servers are agnostic regarding the nature of the distributed resource. They only make sure that a group of equivalent resources distributed arbitrarily remains perceived by its clients as a single entity, with a stable IPv6 address. Distributed servers enable the distributed resource itself to control the mapping between this single virtual IPv6 address and the actual physical resources.

From a conceptual point of view, both services are highly complementary: virtual nodes provide fault-tolerant replication that is mostly transparent to the service developer, but lacks an access method that makes fault-tolerance transparent to the clients; distributed servers provide a solution for making the service replication transparent to the client. Although each service has its own utility when used in isolation, we see that merging both systems would in principle allow one to build fault-tolerant replicated services where the complexity of replication would be transparent to both the service developer, and to the client-side application. Such a system could thus rely on open access protocols such as HTTP/JSON.

Merging these two different services into a single whole is however not a trivial task. First, virtual nodes rely on a Java implementation for simplicity reasons while distributed servers were developed in C to be able to directly access Linux kernel primitives. We therefore had to develop Java support for distributed servers. Second, although both services rely on some membership protocol to give a consistent view of currently available replicas to all parties, the two services have requirements: virtual nodes need to be conservative before declaring a node as failed, because the cost of a false positive is potentially very high; on the other hand, distributed servers should detect node failures much more aggressively to present a continuously available service to the clients. Finally, the semantic of failure recovery in virtual nodes and distributed servers is not exactly the same, which implies that certain corner cases of node failures cannot be recovered transparently to the clients.

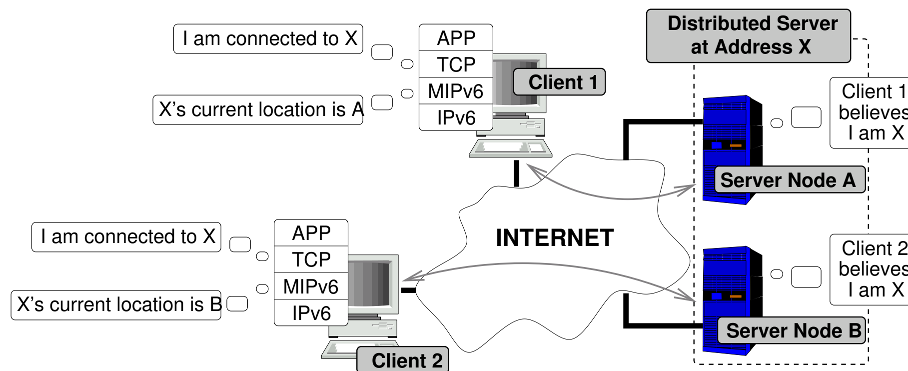


Figure 1: Communication with a Distributed Servers service.

Although we have not yet finished the integration, we believe such a merger between the two services is possible. Note that this deliverable does *not* discuss the internals of virtual nodes or distributed servers separately, since they are already covered in D3.2.6 [7] and D3.2.9 [2]. We instead focus on the integration only.

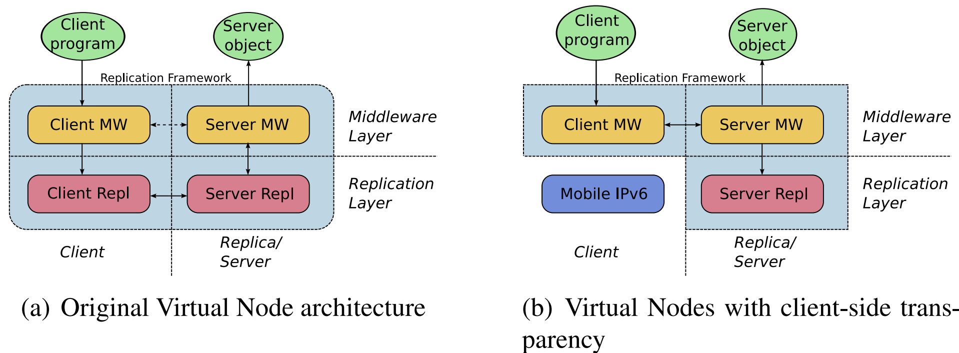
This document is organized as follows: Section 2 provides a brief overview of the two services. Section 3 discusses the scientific challenges that an integration creates with directions to address them. Section 4 describes the planned architecture of the integrated service. Section 5 presents the current status of the integration, and Section 6 concludes.

2 Overview

Before we discuss the technical challenges of integrating Distributed Servers and Virtual Nodes, we briefly review the important aspects of each service.

2.1 Distributed Server Basics

Distributed Servers are an infrastructure that exploits mobile IPv6 [4] in order to achieve location transparency on all layers that are above IP [8]. In mobile IPv6 each mobile node has two addresses: a home address and a remote address. A router in the node's home network (called the home agent) is responsible for routing messages destined for the home address to the remote address. Distributed Servers turn this approach upside-down: multiple nodes share a single mobile IP home address as seen in Figure 1. One of the nodes registers its physical IP address with the home agent. All packets addressed to the public home address



(a) Original Virtual Node architecture

(b) Virtual Nodes with client-side transparency

Figure 2: Integrating Virtual Nodes with Distributed Servers.

will then be relayed to this node that we will refer to as *contact node*. By pretending that the mobile node has changed its location, another node of the group can take over connections on a per client basis—transparently to the client application. The contact node can similarly change its location. Distributed Servers can recover client connections to failing nodes but do not offer any fault-tolerance features at the application level. More details on Distributed Servers can be found in [7].

2.2 Virtual Nodes Basics

Virtual Nodes is a Java-based replication framework for request-response based services [3]. It comes with support for multiple replication protocols and provides various configuration options to optimize the overall system performance. For the remainder of this document we assume the use of active replication protocol. In this replication protocol, all nodes have the same function and run the same algorithm. All replicas are able to receive client requests. A replica receiving a request is called the *contact replica* for that request. The contact replica forwards requests to all other replicas using total order multicast. This order on messages defines their execution order. Assuming that the service implementation is deterministic, the replica state changes in a deterministic way. The reply that results from the execution of a request is sent back to the contact replica that in turn sends the first reply back to the client.

Virtual Nodes do not provide location transparency. However, they decouple the application from the replication framework by allowing plug-in support for arbitrary middleware systems. Yet, the middleware layer at client-side must currently be modified as it needs to be linked to the replication logic. Figure 2(a) shows the four components that encompass virtual nodes and the dependencies

among them. The service is implemented with client and server applications communicating with a generic middleware (MW) component that must be modified to invoke the Virtual Nodes replication layer below. However, the replication layer is also generic, and Virtual Nodes can be integrated into different middleware implementations and thus support different services.

In order to further decouple the user of the service from the replication framework, we propose a system composition like the one shown in Figure 2(b). Here the client uses only unmodified components and thus can rely on off-the-shelf software. The replication layer specific to Virtual Nodes appears only on the server side implementation. The communication middleware must still be modified to invoke the replication layer below, but the client and server applications specific to the service remain unmodified.

3 Challenges and Restrictions

Integrating Virtual Nodes and Distributed Servers presents several challenges, and imposes a number of restrictions. First, we discuss structural challenges such as interfacing the two services written in different languages, then the different uses of membership services between Distributed Servers and Virtual Nodes. Finally, we discuss the challenge for Virtual Nodes to recover from faults using Distributed Servers.

3.1 Interfacing Challenges

Virtual Nodes provide transparent replication for Java server applications. The system semantics of a running system is determined by the service and the Virtual Nodes framework. These entities should trigger changes to the Distributed Servers layer that operates on a lower system level. The API to Distributed Servers, called Gecko, is written in C++ [6, 9]. The Gecko API provides an object-oriented interface to Distributed Servers that provides for client handoff and server management (for example, changing the contact node), but it is not callable directly from Java. Further, Gecko is designed primarily for scalable load distribution of clients rather than fault-tolerance. For example, the Gecko API does not currently support the recovery of a failed client connection.

To integrate Virtual Nodes and Distributed Servers, we have extended the Gecko API and ported it to Java using the Java Native Interface (JNI) [5]. JNI is both necessary and efficient to our purposes. Further, the Java API must be able to interface efficiently with TCPCP2 [1], which provides an API in the C language to save and restore the state of a TCP/IP connection, and is essential for recovering connections to failing nodes. Using JNI also allows us to leverage the Gecko

API directly from Java, eliminating significant reimplementation of Gecko in the Java language.

Our Java interface to Gecko 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 allows to manage the contact node; and *GSocket,* which manages a client connection including fail-over of a connection to a backup node.

3.2 Recovering from Faults

Although Distributed Servers are designed primarily for building a location transparent scalable service, the ability to handoff a client connection can help a server replica recover a client connection from another failed server. A handoff requires creating a new socket to the client at the receiving server side, which can be kept updated with the current state of the connection to the client. In case of failure, the socket can be bound to the client by Distributed Servers. To this end we have created recoverable sockets in Java [7]. Recovery at the application level is handled by Virtual Nodes.

Virtual Nodes provide fault recovery at the application level using replication [2]. Although Distributed Servers can transparently recover a connection to the client, any data sent by the client and acknowledged by the failed server is unrecoverable unless it has already been stored at a replica. There thus exists a tradeoff between acknowledging data to the client quickly to improve latency observed by the client and acknowledging data slowly enough to replicate that data at other server replicas. Deciding this tradeoff belongs to the application level. Currently, our integration efforts acknowledge data quickly, and recover transparently only from failures that occur between consecutive client requests. In the case of a server node failure while processing a request, a backup server node can recover the connection but lacks sufficient information about the request to restart it transparently. In this case a proper exception reply can be sent to the client (instead of letting the dangling connection reach a timeout value if we would not recover the connection). We assume that on receipt of an exception, the request can be quickly retried by the client.

3.3 Membership Views

As shown in Figure 3, the system has two different membership services. This is due to the contradicting goals of high availability and performant operation. On the one hand, the service should be reachable by clients as much as possible. This implies that the contact node should be available at any time to parse and forward

client requests. In case of a suspected failure of the contact node, another node should quickly overtake the contact node responsibility. In other terms, from the point of view of Distributed Servers, continuous availability requirements promote the use of aggressive, false-positive-prone membership management. On the other hand, creating new service-level replicas is expensive, as normal operation has to be suspended and the replica state has to be transferred to the new replica. In consequence, Virtual Nodes prefer to minimize the number of false positives in the case of detecting failed replicas, and therefore use a more conservative membership mechanism.

To satisfy both requirements we use two failure detectors, one for each layer of the system. The membership component in Distributed Servers only cares about the availability of the contact node. All other nodes should monitor the contact node aggressively so that its failure will be detected quickly with little downtime. As soon as the current contact node is suspected to have failed, it is replaced within the Distributed Servers infrastructure. It is not necessary to notify the Virtual Nodes layer about this operation. Assuming the use of active replication, all nodes of the Virtual Nodes are symmetric so any one of them is eligible to receive requests. The benefit of only changing the contact node is that this strategy is resilient to false positives. If the former contact node was suspected by mistake, existing client connections will not be affected. Clients that have already connected to the former contact node will keep those connections. Only newly opened connections will reach the new contact node.

While changing the contact node is a lightweight operation, we want to avoid false positives when deciding about replica failures. Thus, the failure detector of the Virtual Nodes framework is conservative using long timeouts. Once it has decided that a node has failed, it informs the connection manager that starts moving connections of the failed node to other replicas. In addition it also calls the Distributed Servers layer to remove the failed node from its membership list.

4 Expected Architecture

An important design goal is to keep the architecture of the integrated Virtual Nodes and Distributed Servers as independent as possible from the middleware protocol used to access the replicated service. We therefore propose to encapsulate the middleware functionality on the server side in a *protocol adapter* that handles incoming connections according to the communication protocol used by the middleware layer. After a single message is received, the protocol adapter encapsulates it in a middleware-independent message used by the inter-replica communication protocol. The client's middleware layer remains unmodified.

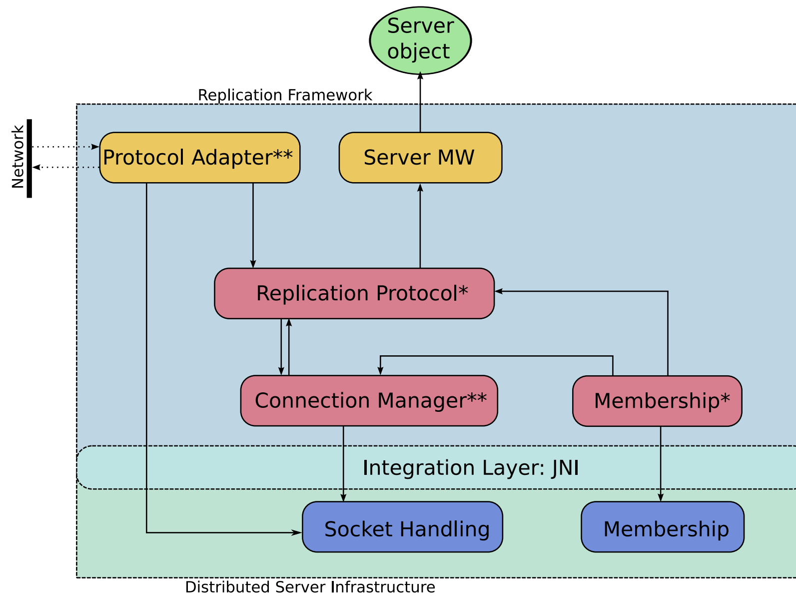


Figure 3: Integrated Virtual Nodes and Distributed Servers architecture.

Figure 3 is an overview of the architecture at server-side and the interactions between the individual components. Entities that are new compared to the stand-alone version of Virtual Nodes are marked with **; those that have to be modified are marked with *. We discuss the new components and the changes to the new components in the next section.

4.1 Virtual Nodes Configuration

Virtual nodes support a wide variety of configuration options. We think that in the integrated system only active replication should be used. Symmetry is a critical issue for such a system, as it allows for switching the contact node fast, which in turn is crucial for the system's availability as we discuss in Section 3. In an asymmetric replication scheme such as passive replication, switching the contact node would require also switching the primary replica, as only the primary replica may receive and process requests.

4.2 Distributed Servers Configuration

The Distributed Servers framework should be configured so that all of the replicas of Virtual Nodes are available for both handoffs of clients (to support recovery of the client connections after a replica failure) and for taking over as contact

node (to replace a failed contact node). As discussed in Section 3.3, Distributed Servers can be configured to change the contact node aggressively to maintain high availability of the contact node without negative impact on the Virtual Nodes framework.

4.3 Runtime Behavior

During normal operation there are a few differences to the standard Virtual Nodes operation. When a request is received, the protocol adapter takes a snapshot of the associated socket. This snapshot is appended to the request message that is relayed to other replicas and allows replicas to take over the client connection should the contact node fail. When a replica receives such a request, it extracts the socket snapshot from the message and passes the snapshot, the request, and the id of the contact replica to a *connection manager*. The connection manager can thus maintain a mapping between socket snapshots, the requests received through that socket, and the contact replica. There are no changes to the way requests are processed compared to normal Virtual Nodes operation. The connection manager is also responsible for bringing up the Gecko framework at system startup to initialize and manage Distributed Servers.

When a replica failure is detected by the Virtual Nodes membership manager, the connection manager checks whether there are socket snapshots where the failed replica is the contact. If so, the connection manager must choose a new contact replica for the corresponding requests. As the connection manager is not integrated with Distributed Servers, it cannot chose the current contact node. Furthermore, even if the manager could retrieve the information from Distributed Servers, the decision of which node should be the contact node is not deterministic. Different replicas may request the information at different points in their execution with different results. However, view changes due to changes in group membership directly influence the rank a replica has in the group. Thus, within one view there is always exactly one oldest replica, and its identity is known to the membership manager. Hence, the connection manager replaces the failed replica's information with the oldest replica obtained from the membership manager, assigning the oldest replica as the new contact replica.

After assigning a new contact replica, the connection manager then forwards any requests for the failed contact replica to the replica protocol pretending that it was sent by the new contact replica. The new contact replica ignores duplicate requests as the replication protocol supports at-most-once semantics of invocations. A reply is also sent to the new contact replica if it is available.

The new contact replica reactivates the sockets from the snapshots that it receives, takes over the corresponding connection, and waits for replies from the other replicas to come in. Once the first reply has arrived from other replicas, the

new contact replica can construct the message that is to be sent to the client. Activating the socket returns the number of bytes that have been sent and received by the failed replica since the time the snapshot was taken. The new replica can thus detect whether a reply has already been sent or partially sent. For deterministic services all replies are identical so the new contact replica can send any remaining bytes of the reply back to the client.

Finally, to avoid stale snapshots in the connection manager, we require that a replica broadcasts a message as soon as the client acknowledges receiving its reply. This message might either contain the information that the socket was closed or a new snapshot. It also triggers the removal of the respective entry in the cache used by the replication protocol for enabling at most once semantics.

5 Current Status and Open Issues

A wide part of the JNI integration layer has now been implemented and tested. Virtual Nodes can thus start and steer a Distributed Servers instance. We have all mechanisms available to serialize a socket, copy it to another machine, and reactivate it there. The connection manager has been fully implemented, but is not yet integrated in the Virtual Nodes framework. The same holds for the membership service of Distributed Servers. Consequently, the interaction between the Distributed Servers and Virtual Nodes membership layers has not been tackled so far. Furthermore, the system lacks a fully functional protocol adapter. An adaptor for SOAP/JSON is in progress.

There exist currently two versions of Virtual Nodes: one for stand-alone usage and a stripped-down version to be used with Distributed Servers. In the long run, a seamless integration of the extended functionality in Virtual Nodes is critical in order to avoid maintaining two code bases.

Finally, the current approach does not support clients that multiplex sockets. For now we assume a client cannot send two concurrent requests over the same connection. It is unclear whether this is a critical issue though. The relevance for such protocols for the domain we target should be subject to further investigations.

6 Conclusion

Although Virtual Nodes and Distributed Servers were originally designed as two independent technologies, we found a great interest in combining them together to build transparently-replicated highly available Grid services. Such integration is however far from trivial. First, the two services were originally written in dif-

ferent languages. Second, and more importantly, the two layers have seemingly conflicting goals regarding the way to handle group membership.

We showed how an integrated system could be constructed, and made significant progress toward its implementation. When this work is available, we plan to promote its use within XtreemOS so that mission-critical services can run in a fault-tolerant environment provided by Distributed Servers and Virtual Nodes.

References

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

[2] Jörg Domaschka. Reproducible evaluation of a virtual node system. XtreemOS deliverable D3.2.9, November 2008.

[3] Jörg Domaschka, Thomas Bestfleisch, Franz J. Hauck, Hans P. Reiser, and Rüdiger Kapitza. Multithreading strategies for replicated objects. In *Proceedings of the 9th International Middleware Conference*, December 2008.

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

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

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

[7] Guillaume Pierre. Reproducible evaluation of distributed servers. XtreemOS deliverable D3.2.6, November 2008.

[8] 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.

[9] Willem van Duijn. A versatile anycast framework for distributed servers. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, February 2008.