High Availability for the Lustre File System

A Dissertation
Submitted In Partial Fulfilment Of
The Requirements For The Degree Of

MASTER OF SCIENCE

In

NETWORK CENTERED COMPUTING,
HIGH PERFORMANCE COMPUTING AND COMMUNICATION

in the

FACULTY OF SCIENCE

THE UNIVERSITY OF READING

by

Matthias Weber

14 March 2007

Supervisors:
Prof. Vassil Alexandrov, University of Reading
Christian Engelmann, Oak Ridge National Laboratory

Acknowledgment

I would like to thank all of you who have given your time, assistance, and patience to make this dissertation possible.

For making my research at the Oak Ridge National Laboratory possible in the first place, I want to thank my advisor, Christian Engelmann, and Stephen L. Scott. I appreciate their invitation to write my Master thesis at such a renowned institution and the financial support.

I especially like to thank Christian for his great support and being as excited about the ongoing research as I am.

Also thanks to Hong Hoe Ong for his support in some struggle with Lustre and to Li Ou for his help with the prototype design.

Thank you Cindy Sonewald for struggling with administration and bureaucracy to keep me alive during my stay in the US.

This research is sponsored by the Mathematical, Information, and Computational Sciences Division; Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725.

Abstract

With the growing importance of high performance computing and, more importantly, the fast growing size of sophisticated high performance computing systems, research in the area of high availability is essential to meet the needs to sustain the current growth.

This Master thesis project aims to improve the availability of Lustre. Major concern of this project is the metadata server of the file system.

The metadata server of Lustre suffers from the last single point of failure in the file system. To overcome this single point of failure an active/active high availability approach is introduced.

The new file system design with multiple MDS nodes running in virtual synchrony leads to a significant increase of availability.

Two prototype implementations aim to show how the proposed system design and its new realized form of symmetric active/active high availability can be accomplished in practice.

The results of this work point out the difficulties in adapting the file system to the active/active high availability design. Tests identify not achieved functionality and show performance problems of the proposed solution.

The findings of this dissertation may be used for further work on high availability for distributed file systems.

Contents

1 Introduction

1.1 Background

1.1.1 High Performance Computing

High-performance computing (HPC) has become more and more important in the last decade. With help of this tool problems in research worldwide, such as in climate dynamics or human genomics are solved. Such real-world simulations use multi-processor parallelism and exploit even the newest HPC systems.

In general these sophisticated HPC systems suffer a lack of high availability. Thus, the HPC centres set limited runtime for jobs, forcing the application to store results. This checkpointing process wastes valuable computational time.

A desired way of producing computational results would be to use no checkpoints and to produce the result without interruption. This way, no computational time would be wasted and the result would be produced in the fastest possible way. In order to use this approach, HPC with no unforeseeable outages is required.

To make current and future HPC systems capable of these demands is the aim of ongoing research in the Oak Ridge National Laboratory (ORNL). The goal is to provide high availability (HA) for critical system components in order to eliminate single points of failure. Therefore different methods of high availability have been tested and implemented in some systems.

1.1.2 The Lustre File System

Lustre is one of many available parallel file systems. It runs on some of the fastest machines in the world. The Oak Ridge National Laboratory uses Lustre as well for their HPC Systems.

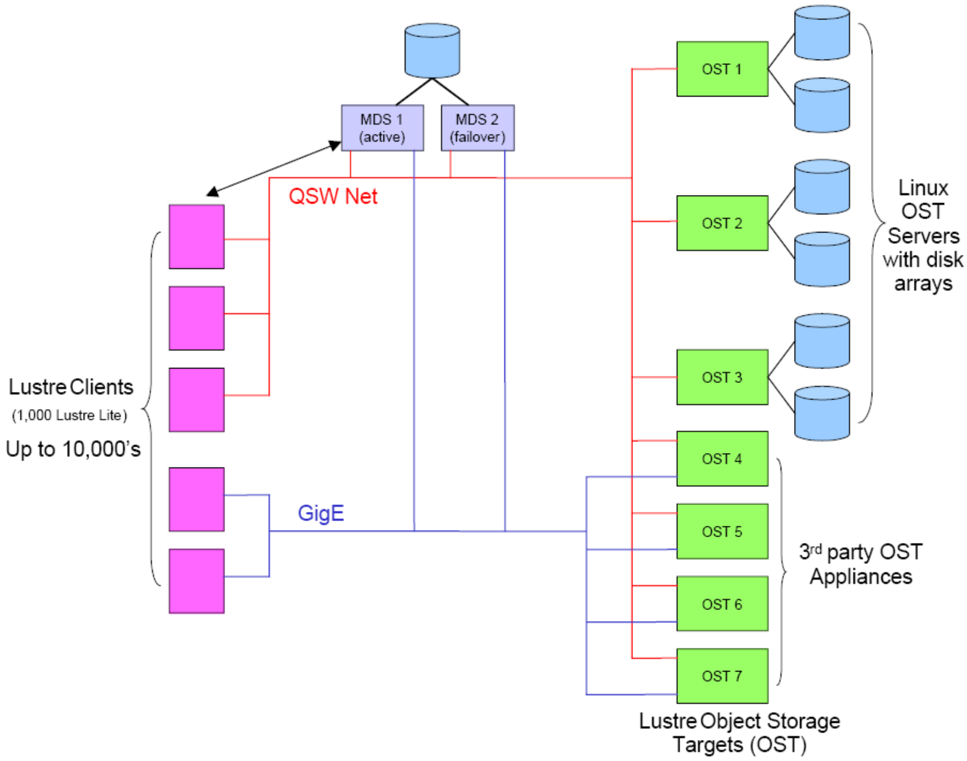

Figure 1.1.: Lustre Overview [8]

Today's network-oriented computing environments require high-performance, network-aware file systems that can satisfy both the data storage requirements of individual systems and the data sharing requirements of workgroups and clusters of cooperative systems. The Lustre File System, an open source, high-performance file system from Cluster File Systems, Inc., is a distributed file system that eliminates the performance, availability, and scalability problems that are present in many traditional distributed file systems. Lustre is a highly modular next generation storage architecture that combines established, open standards, the Linux operating system, and innovative protocols into a reliable, network-neutral data storage and retrieval solution. Lustre provides high I/O throughput in clusters and shared-data environments, and also provides independence from the location of data on the physical storage, protection from single points of failure, and fast recovery from cluster reconfiguration and server or network outages. [8, page 1]

Figure 1.1 shows the Lustre File System design. Lustre consists of three main components:

- Client
- Meta Data Server (MDS)
- Object Storage Target (OST)

Lustre supports tens of thousands of Clients. The client nodes can mount Lustre volumes and perform normal file system operations, like create, read or write.

The Meta Data Server (MDS) is used to store the metadata of the file system. Currently, Lustre supports two MDS. One is the working MDS, the other is the backup MDS for failover. The Lustre failover mechanism is illustrated in Figure 1.2. In case of a failure the backup MDS becomes active and the clients switch over to this MDS. However, these two MDS share one disk to store the Metadata. Thus, this HA approach still suffers a single point of failure.

The Object Storage Target (OST) is used to physically store the file data as objects. The data can be striped over several OSTs in a RAID pattern. Currently, Lustre supports hundreds of OSTs. Lustre automatically avoids malfunctioning OSTs.

The components of Lustre are connected together and communicate via a wide variety of networks. This is due to Lustre's use of an open Network Abstraction Layer. Lustre currently supports tcp (Ethernet), openib (Mellanox-Gold Infiniband), iib (Infinicon Infiniband), vib (Voltaire Infiniband), ra (RapidArray), elan (Quadrics Elan), gm (Myrinet).

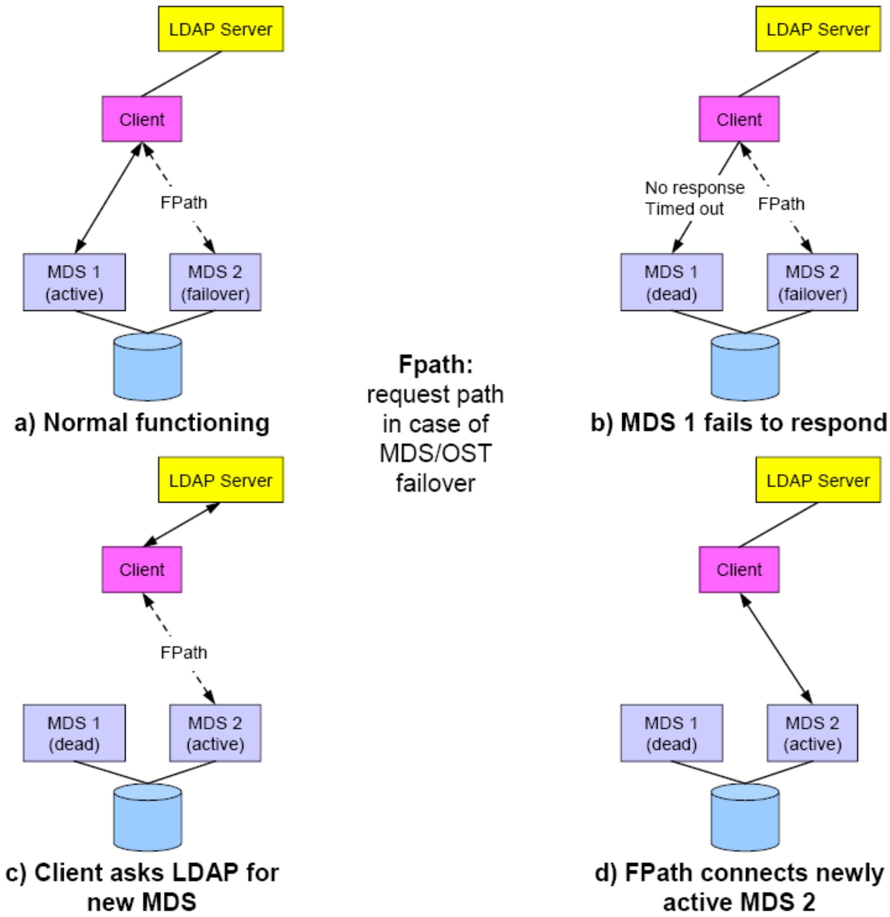

Figure 1.2.: Lustre Failover Mechanism [8]

1.2 Previous Work

1.2.1 High Availability Computing

HA of a system is its ability to mask errors from the user. This is achieved with redundancy of critical system components and thus elimination of single points of failure. If a component fails the redundant component takes over. This functionality prevents system outages and possible loss of data.

The degree of transparency in which this replacement occurs can lead to a wide variation of configurations. Warm and hot standby are active/standby configurations commonly used in high availability computing. Asymmetric and symmetric active/active configurations are commonly used in mission critical applications.[1]

- **Warm Standby** requires some service state replication and an automatic fail-over. The service is interrupted and some state is lost. Service state is regularly replicated to the redundant service. In case of a failure, it replaces the failed one and continues to operate based on the previous replication. Only those state changes are lost that occurred between the last replication and the failure.[1]

- **Hot Standby** requires full service state replication and an automatic fail-over. The service is interrupted, but no state is lost. Service state is replicated to the redundant service on any change, *i.e.*, it is always up-to-date. In case of a failure, it replaces the failed one and continues to operate based on the current state.[1]

- **Asymmetric Active/Active** Asymmetric active/active requires two or more active services that offer the same capabilities at tandem without coordination, while optional standby services may replace failing active services ($n + 1$ and $n + m$). Asymmetric active/active provides improved throughput performance, but it has limited use cases due to the missing coordination between active services.[1]

- **Symmetric active/active** requires two or more active services that offer the same capabilities and maintain a common global service state using virtual synchrony. There is no interruption of service and no loss of state, since active services run in virtual synchrony without the need to fail-over.[1]

These redundancy strategies are entirely based on the fail-stop model, which assumes that system components, such as individual services, nodes, and communication links, fail by simply stopping. They do not guarantee correctness if a failing system component violates this assumption by producing false output.[1]

Previous and related research in the area of symmetric active/active HA encompasses the two following described projects. Goal of these projects were prototype implementations as proof-of-concept.

[1] Towards High Availability for High-Performance Computing System Services [12]

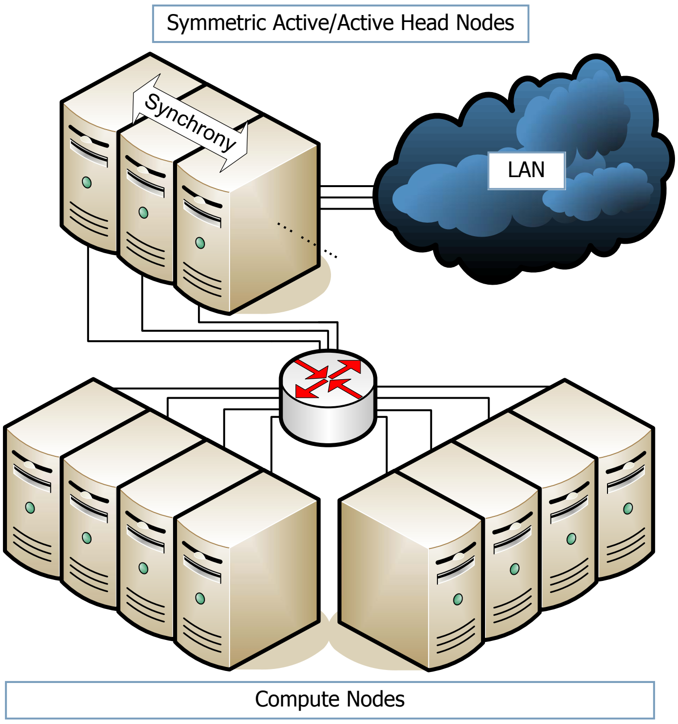

Figure 1.3.: Advanced Beowulf Cluster Architecture with Symmetric Active/Active High Availability for Head Node System Services [21]

JOSHUA

The emergence of cluster computing in the late 90s made low to mid-end scientific computing affordable to everyone, while it introduced the Beowulf cluster system architecture with its single head node controlling a set of dedicated compute nodes. The impact of a head node failure is severe as it not only causes significant system downtime, but also interrupts the entire system. One way to improve the availability of HPC systems is to deploy multiple head nodes.[19]

The JOSHUA project offers symmetric active/active HA for HPC job and resource management services. It represents a virtually synchronous environment using external replication providing HA without any interruption of service and without any loss of state.[21]

Figure 1.3 shows the system layout of the prototype solution in the JOSHUA project.

The prototype uses the external way to replicate the system service head nodes. Transis is used as group communication facility. The prototype design of the JOSHUA project is in its basic technologies very close to the intended solution of this project. The performance test results of the JOSHUA prototype, shown in Table 1.1, are an excellent example of the latency time imposed by the use of external replication. These times can be used to compare and judge the performance of the prototype of this project.

System	#	Latency	Overhead
TORQUE	1	98 ms	
JOSHUA/TORQUE	1	134 ms	36 ms / 37%
JOSHUA/TORQUE	2	265 ms	158 ms / 161%
JOSHUA/TORQUE	3	304 ms	206 ms / 210%
JOSHUA/TORQUE	4	349 ms	251 ms / 256%

Table 1.1.: Job Submission Latency Comparison of Single vs. Multiple Head Node HPC Job and Resource Management [21]

Metadata Service for Highly Available Cluster Storage Systems

The "Metadata Service for Highly Available Cluster Storage Systems" project targets the symmetric active/active replication model using multiple redundant service nodes running in virtual synchrony. In this model, service node failures do not cause a fail-over to a backup and there is no disruption of service or loss of service state. The prototype implementation shows that high availability of metadata servers can be achieved with an acceptable performance trade-off using the active/active metadata server solution.[2]

Goal of the project was the replication the metadata server of the Parallel Virtual File

	number of clients					
System	1	2	4	8	16	32
PVFS 1 server	11	23	52	105	229	470
Active/Active 1 server	13	27	54	109	234	475
Active/Active 2 servers	14	29	56	110	237	480
Active/Active 4 servers	17	33	67	131	256	490

Table 1.2.: Write Request Latency (ms) Comparison of Single vs. Multiple Metadata Servers [18]

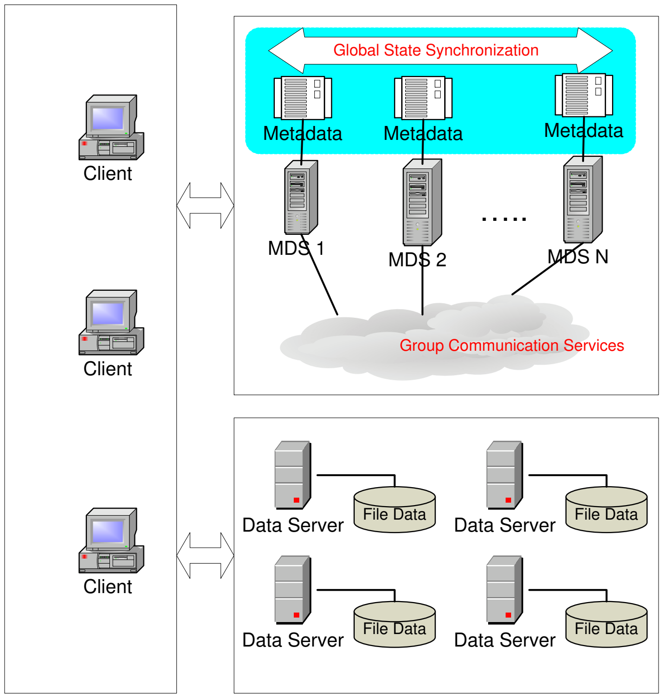

Figure 1.4.: Active/Active Metadata Servers in a Distributed Storage System [18]

System (PVFS). The replication was realised using the internal method. The group communication functionality was implemented with help of Transis. Since this Master thesis targets the same goal like the "Metadata Service for Highly Available Cluster Storage Systems" project, except with Lustre instead of PVFS, the acquired performance tests results are exceptionally valuable for comparison and judgement. Table 1.2 shows the write latency time caused by multiple metadata servers. Figures 1.5 and 1.6 show the read and write throughput of the attained prototype solution of the project.

1.2.2 Virtual Synchrony

In order to design a HA architecture, important system components must be replicated. As a result the former single component builds a group of redundant components. This group behaves like a single component to the rest of the system. If one component in this group fails a redundant component can take over. In case of an active/active architecture, the components in the group have to be in virtual synchrony. This means

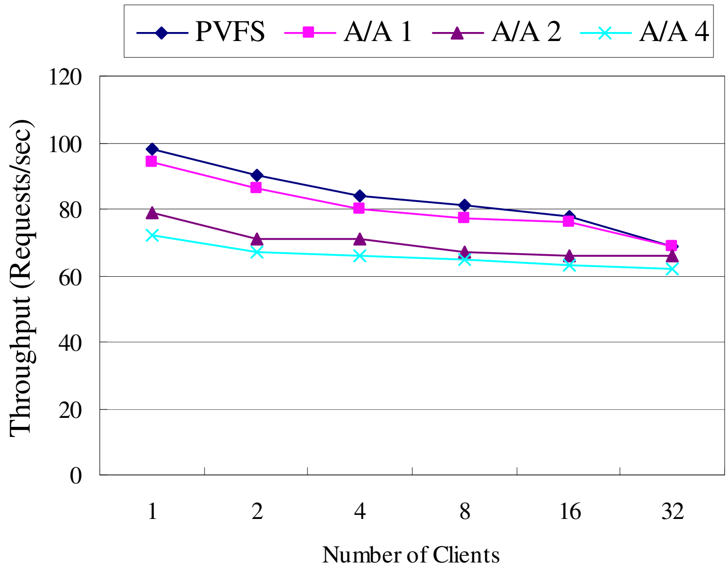

Figure 1.5.: Write Request Throughput Comparison of Single vs. Multiple Metadata Servers, A/A means Active/Active Servers [18]

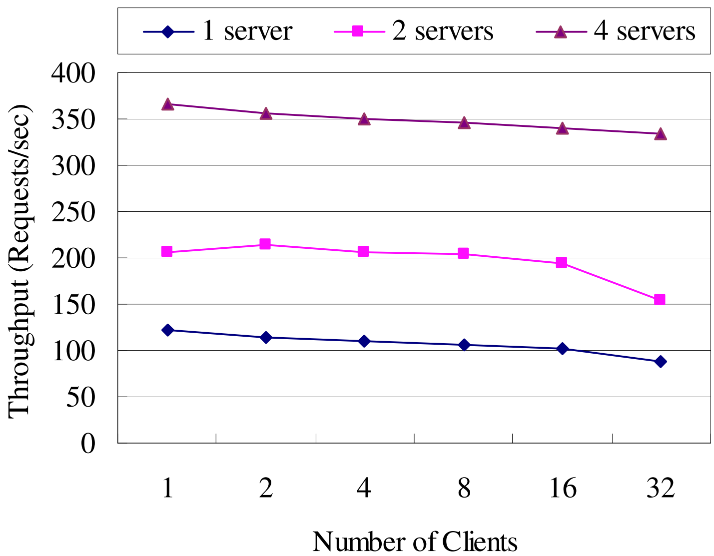

Figure 1.6.: Read Request Throughput Comparison of Single vs. Multiple Metadata Servers [18]

that every component is in the same state as the others. This can be achieved through a group communication system (GCS). The GCS is like a shell around the group of redundant components. It intercepts the requests from the system and distributes them to the group. In this step it also ensures total ordering of the messages. This way it is ensured that every component gets the same requests in the same order and produces therefore the same outputs. The GCS is also responsible for filtering of all the equal outputs from the redundant components of the group and sending each output only once to the system.

There are many different GCS available. Some of them are Isis, Horus, and Transis. The experience from the preceding HA projects[2,3] in the ORNL has shown that Transis[4] is the most suitable one. It is an open source group communication project from the Hebrew University of Jerusalem.

Transis can provide all necessary group communication facilities needed for the implementation of the high available job scheduler service system.

The Transis group communication framework provides:

- group communication daemon
- library with group communication interfaces
- group membership management
- support for message event based programming

Distributed locks or even distributed mutual exclusion solutions are not included and have to be implemented, if needed.

The fact that Transis is an open source project makes necessary adjustments possible. In the scope of the Metadata Service Project[2] Transis has been improved by Li Ou. Through the new "Fast Delivery Protocol" implementation it offers lower latency and better throughput than the standard Transis implementation.

[2]The JOSHUA Project [21]
[3]Symmetric Active/Active Metadata Service [18]
[4]The Transis Project [3]

The changes due to the "Fast Delivery Protocol" are described in the paper "A Fast Delivery Protocol for Total Order Broadcasting"[19].

Total order broadcasting is essential for group communication services, but the agreement on a total order usually bears a cost of performance: a message is not delivered immediately after being received, until all the communication machines reach agreement on a single total order of delivery. Generally, the cost is measured as latency of totally ordered messages, from the point the message is ready to be sent, to the time it is delivered by the sender machine.[19]

In communication history algorithms, total order messages can be sent by any machine at any time, without prior enforced order, and total order is ensured by delaying the delivery of messages, until enough information of communication history has been gathered from other machines.[19]

Communication history algorithms have a post-transmission delay. To collect enough information, the algorithm has to wait for a message from each machine in the group, and then deliver the set of messages that do not causally follow any other, in a predefined order, for example, by sender ID. The length of the delay is set by the slowest machine to respond with a message. The post-transmission delay is most apparent when the system is relatively idle, and when waiting for response from all other machines in the group. In the worst case, the delay may be equal to the interval of heart beat messages from an idle machine. On the contrary, if all machines produce messages and the communication in the group is heavy, the regular messages continuously form a total order, and the algorithm provides the potential for low latency of total order message delivery. In a parallel computing system, multiple concurrent requests are expected to arrive simultaneously. A communication history algorithm is preferred to order requests among multiple machines, since such algorithm performs well under heavy communication loads with concurrent requests. However, for relatively light load scenarios, the post-transmission delay is high. The fast delivery protocol reduces this post-transmission delay. It forms the total order by waiting for messages only from a subset of the machines in the group, and by fast acknowledging a message if necessary, thus it fast delivers total order messages.[19]

1.3 Key Problems and Specification

This master thesis aims to develop a HA solution for the Meta Data Server (MDS) of the Lustre File System.

So far, the Lustre File System provides only an active/standby architecture for the MDS. This solution uses one shared disk for both Meta Data Servers, and therefore suffers from a single point of failure.

The aim is to eliminate this last single point of failure in Lustre and to implement an active/active HA architecture for the MDS. This will replicate the MDS on several nodes using their own disk to hold the Metadata.

Thus, the result of the project should be a prototype providing the highest possible degree of availability for the MDS.

1.4 Software System Requirements and Milestones

To overcome the problems of the existing HA solution of Lustre the single point of failure must be eliminated. Therefore the design of Lustre has to be changed. To achieve the highest rate of availability for Lustre, a symmetric active/active architecture for the MDS needs to be implemented.

The work carried out to realize a symmetric active/active architecture for the MDS of PVFS gives an example solution to the problem.[5] In this project an internal replication of the MDS was implemented with the use of Transis as group communication facility.

To achieve a similar solution for the Lustre File System the MDS must be "isolated" from the other components of the system. After this step the MDS has to be replicated. This may be done in two ways. The "internal" and the "external" replication. Both methods have their own advantages and disadvantages. Which method to choose has to be investigated in the beginning of the project.

[5] Symmetric Active/Active Metadata Service [21]

If replication is done internally, the MDS of Lustre itself needs to be analysed in order include the group communication system into the code. If replication is done externally, a complete understanding of the Lustre networking and the MDS protocol is needed.

The most important part of the active/active HA architecture is the global state of the replicated MDS. Each MDS has to have the same state like the others. The MDS group has to run in virtual synchrony. To achieve this goal every possible communication to and also from the MDS has to be analysed. This communication has to be handled properly with the help of group communication software.

Furthermore, a solution for dynamic group reconfiguration has to be developed. The recovery, joining and leaving of group members must be masked from the user. Therefore the functionality of the MDS itself needs to be analysed.

Another key problem is the single instance execution problem. Because the MDS group members run in virtual synchrony every single MDS produces the same output. The group communication software has to be designed in a way, that makes sure the proper output is send only once to the requesting component of the system.

In order to mask a failing MDS from connected clients a connection failover mechanism needs to be implemented. If the connected MDS fails, the mechanism has to reconnect to another MDS group member. Therefore the client code must be adjusted and a list of available MDS group members has to be hold and updated at runtime.

The main goal is to design, implement and test a prototype software that meets the proposed criteria. The prototype should use the existing Transis group communication software as basis to implement the virtual synchrony functionality.

The following milestones are set up to help to evaluate every step during the development process toward the final implementation.

There are three different milestone categories, which outline the project development status:

- Milestone Category A - **minimal** criteria and requirements are met
- Milestone Category B - **optimal** criteria and requirements are met

- Milestone Category C - all requirements are met, including **extra** capabilities

The following requirement table will be the criteria foundation to judge the success of the later implementation and the project process. Especially the system tests will prove, whether all the requirements are met by the dissertation project.

required capability	**category**	**milestone**
analysis of MDS communication	**A**	**1**
choice of one replication method	**A**	**2**
replication of the MDS on the backup node in active/active mode	**A**	**3**
solution for single instance execution problem	**A**	**4**
MDS service stays available, as long as one node is up	**A**	**5**
replication of the MDS on more than two nodes	**B**	**6**
client connects to other MDS node if own fails	**B**	**7**
new MDS nodes can dynamically join	**B**	**8**
client table of MDS nodes is updated at runtime	**B**	**9**
performance improvements for prototype development	**C**	**10**

Table 1.3.: Requirements and Milestones Overview

2

Preliminary System Design

2.1 Analysis of Lustre

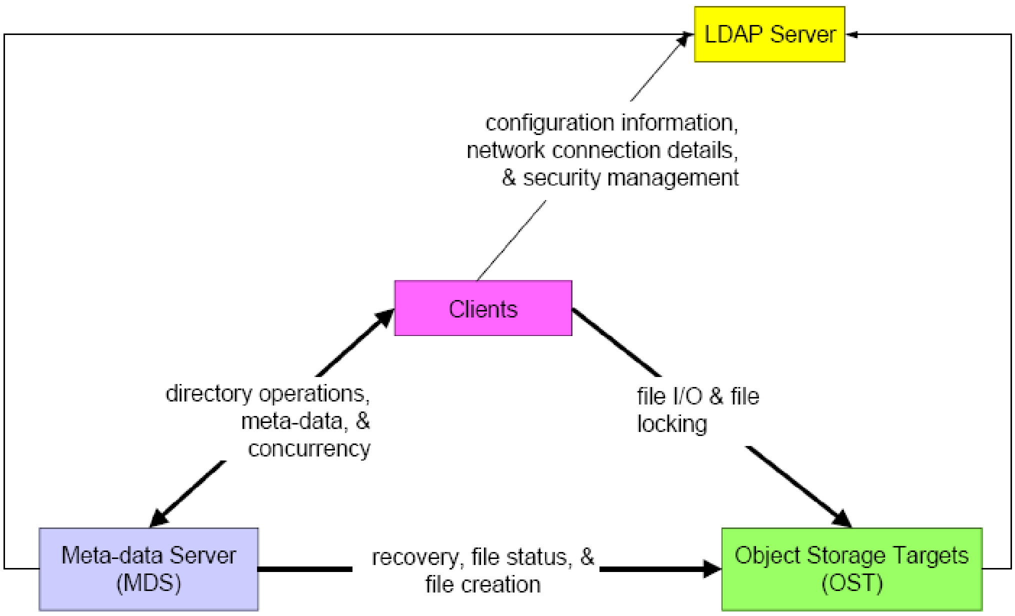

Figure 2.1.: Interactions between Lustre Subsystems [8]

In order to design a sufficient HA solution Lustre needs to be analysed. Goal is to understand partwise the inner workings of the relevant system components and the communication in particular.

The Lustre software distribution comes with a couple of papers and manuals describing the file system and its components in general. One crucial information needed to design the prototype is the exact communication (e.g. protocol, what format, what content, how much messages for one task ...) between the MDS and the other components. Lustre itself provides almost as much information on that matter as shown in Figure 2.1. This is by far to general and of little value for the prototype design. As a result, there is no way around reading and analysing the Lustre source code.

The analysis of the source code takes a lot of time due to almost no comments in the code and no description at all. The other problem is the code itself. The Lustre design is very complex and complicated what makes the code intransparent and hard to read. One example is that Lustre runs nearly all components as kernel modules. Thus they publish most of the functions to the kernel namespace. That way they can be called all the time from everywhere in the kernel. That makes it hard to point out the function call path like in a normal program. Also the code itself differs from a normal user space application due to the fact that it is kernel code.

2.1.1 Lustre Design

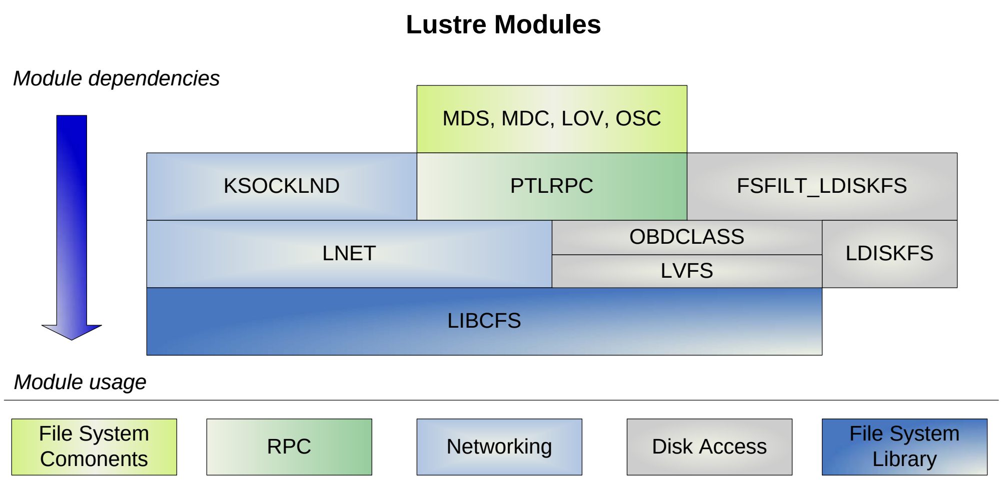

Figure 2.2.: Lustre Module Dependencies

The design of Lustre is highly modular. Figure 2.2 shows a snapshot of the loaded modules of a running Lustre. Table 2.1 gives the description of the modules provided in the source code. Besides the main components like OST or MDS, Lustre uses also a lot of other modules to do the networking or the disk access.

Module	Description
MDS	Metadata Server
MDC	Metadata Client
LOV	Logical Object Volume OBD Driver
OSC	Object Storage Client
PTLRPC	Request Processor and Lock Management
KSOCKLND	Kernel TCP Socket LND v1.00
LNET	Portals v3.1
FSFILT-LDISKFS	Lustre ext3 File System Helper
LDISKFS	Lustre ext3 File System
OBDCLASS	Lustre Class Driver
LVFS	Lustre Virtual File System
LIBCFS	Lustre File System Library

Table 2.1.: Lustre Module Description

For calls between modules Lustre uses its own kind of remote procedure call (RPC) sent via Sockets over the network. Because Lustre is written in C and there are no object oriented facilities available, Lustre uses structures extensively to organise data. Even the network component itself (LNET) is hold in a structure.

To perform a call from the client (in this case the MDC) to the server (the MDS) Lustre uses the modules in the way indicated in Figure 2.2. The data, the request itself and the needed information for the connection is assembled and packed from one structure into another from module to module. This scheme in shown in Figure 2.3. The response from the MDS takes the same way backwards.

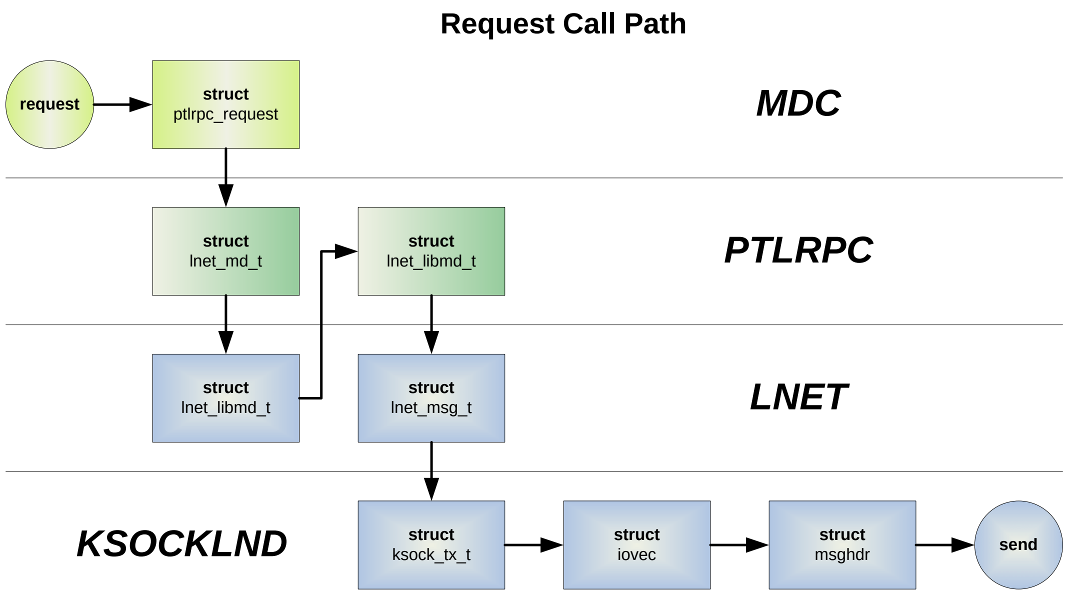

Figure 2.3.: Path of Metadata Client Request

2.1.2 Lustre Networking

Lustre is a tightly integrated system. All of its components are defined and assigned to nodes before the system starts. That way the file system knows all nodes and the complete setup in advance. As part of the Lustre security concept only messages from these defined nodes are accepted.

Lustre also accepts only direct sent messages and thus doesn't allow routing of messages. In order to check integrity of received messages Lustre looks into the message header. It compares the sender of the message given in the header with the address of the node from which the message was received. If they don't match the message is dropped.

The connections are set up like shown in Figure 2.4. First the OSTs are started. Afterwards the MDS is started. The MDS connects to the OSTs. At last the clients are started. They connect to the MDS as well as to the OSTs.

Each component initiates three single connections to the respective component. For instance, the Client opens three ports to the MDS. Another restriction of Lustre is that only three connections per node are accepted. In case a node opens more connections

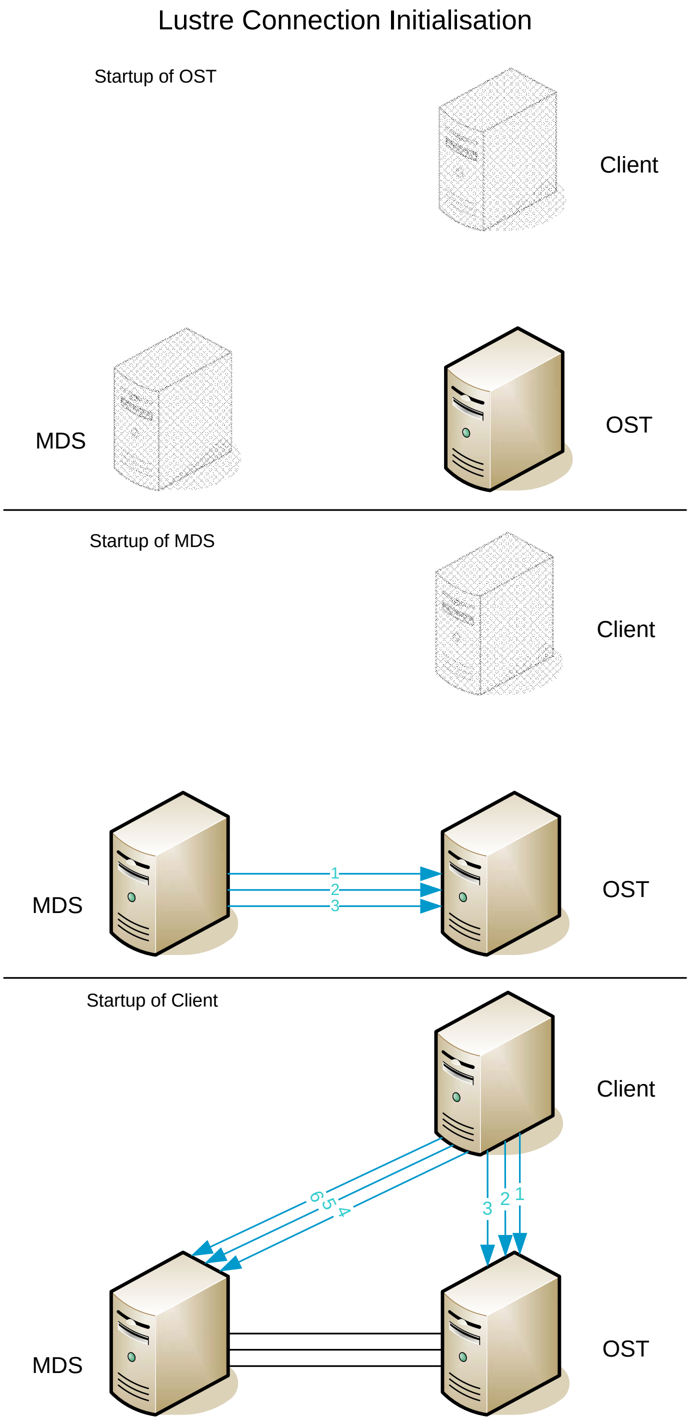

Figure 2.4.: Lustre Connection Initialisation

e.g., a client tries to establish a fourth connection, the first connection is dropped.

To initiate a connection between two components, the Lustre protocol must be followed. This process takes four messages explained in the following example of a client establishing a connection to the MDS.

Figure 2.5.: Lustre Acceptor Request Message

First the client sends an “Acceptor Request” message to the MDS. This message has the layout as shown in Figure 2.5. The message is 16 bytes long. The fist 4 bytes are the indicator of the used acceptor protocol. The next 4 bytes describe the protocol version. Whereas the number is split internally into two 2 byte values describing the minor and major version number. This is checked for compatibility reasons with later Lustre versions. The last 8 byte number identifies the target to which the connection should be established. This target nid consists of a 4 byte address and 4 byte network id. The address id is directly created from the IP address of the node. The network id identifies the network type e.g., TCP. This information is needed because Lustre is capable of using different network types at the same time. When this message arrives at the MDS and if the values are correct the connection from the client is accepted. Now the LNET layer of Lustre must be initialised. Therefore the MDS waits for the “LNET Hello” message from the client.

The “LNET Hello” message is indicated in Figure 2.6. It consists of a 32 bytes header and payload. The size of the payload is given in the header. However, in the “LNET Hello” message this size is zero and no payload is sent. The first 4 byte describe the

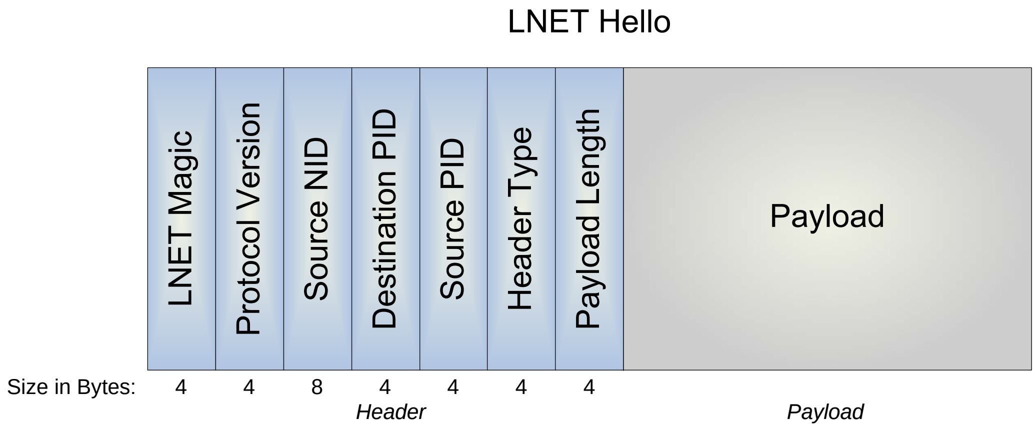

Figure 2.6.: Lustre LNET Hello Message

used LNET protocol. The next 4 byte, like in the Acceptor Request message, describe the protocol minor and major version. The following 8 byte hold information about the sender of this message. They contain the address and network type of the source node. The next two 4 byte values are used to identify and distinguish this message from other messages. The MDS for instance uses the Process Id (pid) numbers to identify a request and to send the processed response to that request. With the sent pid the client can identify the response from the MDS and assign it to this request. The 4 byte value "Header Type" type identifies type of the header. For metadata this value is always "SOCKNAL_RX_HEADER". This is due to the fact that one request is done in one message. For transport of file data, the header type could change to other values, like "SOCKNAL_RX_BODY", because more than one message may be needed to transfer the entire datablock. However, this field is of no concern in terms of metadata. The last 4 byte value holds the size of the payload. This value should be zero in "LNET Hello" messages.

The "LNET Hello" messages are exchanged in form of a handshake. Fist the client sends his "LNET Hello" message to the MDS. Then he waits for the "LNET Hello" from the MDS. When the MDS receives the "LNET Hello" from the client he checks the values and sends his "LNET Hello" message back to the client. After the "LNET Hello" messages are exchanged, one more message is needed to fully establish the connection. This message is described next.

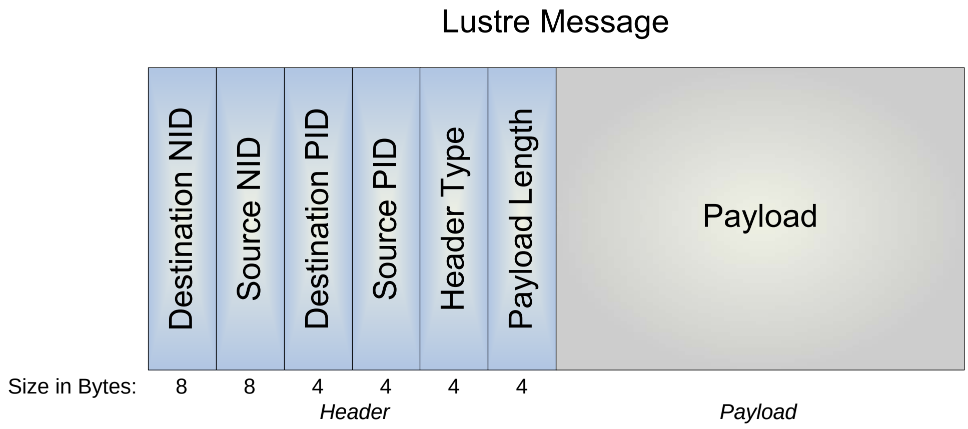

Figure 2.7.: Ordinary Lustre Message

The ordinary Lustre message format is shown in Figure 2.7. A Lustre message consists of the 32 bytes header and payload. The fist two 8 byte values hold the address and network type of the message source and destination node. The next three 4 byte values are the same like in the “LNET Hello” header. The pid values are used to identify the requests and responses. The header type is always “SOCKNAL_RX_BODY” because one request is transmitted completely in one message. The last 4 bytes of the header hold the size of the payload. This size is limited to 4KB in Lustre. The payload is sent directly behind the header.

To complete the communication initialisation after the “LNET Hello” handshake, one message is sent from the client to the MDS. This message holds the Universally Unique Identifier (UUID) of the client and the MDS in the payload. With this information the MDS can fully establish the connection to the client and process its requests.

A Universally Unique Identifier (UUID) is an identifier standard used in software construction, standardized by the Open Software Foundation (OSF) as part of the Distributed Computing Environment (DCE). The intent of UUIDs is to enable distributed systems to uniquely identify information without significant central coordination. Thus, anyone can create a UUID and use it to identify something with reasonable confidence that the identifier will never be unintentionally used by anyone for anything else. Information labelled with UUIDs can therefore be later combined into a single database without needing to resolve name conflicts. The most widespread use of this standard is in Microsoft's Globally Unique Identifiers (GUIDs) which implement this standard. Other significant users include Linux's ext2/ext3 filesystem, LUKS encrypted partitions, GNOME, KDE, and Mac OS X, all of which use implementations derived from the uuid library found in the e2fsprogs package.[4]

A UUID is essentially a 16-byte (128-bit) number. In its canonical hexadecimal form a UUID may look like this:

550e8400-e29b-41d4-a716-446655440000

The number of theoretically possible UUIDs is therefore $2^{128} = 256^{16}$ or about $3.4x10^{38}$. This means that 1 trillion UUIDs have to be created every nanosecond for 10 billion years to exhaust the number of UUIDs.[4]

2.2 Replication Method

Before the prototype can be designed, a decision about the replication method has to be made. This decision is vital as it affects the entire prototype design. Both replication methods have their own advantages and disadvantages. But it is not only the question what method suits best the needs of the prototype. The other important fact to consider is the feasibility of each method with respect to the Lustre design and the possibilities in the scope of this thesis.

2.2.1 Feasibility of Internal Replication

In the internal replication, as shown in Figure 2.8, the group communication system is implemented direct into the Lustre code. Thus no inter-process communication is needed and as a result this method should yield higher performance than the external.

In general there should be no problem with Lustre itself to realize this method. It would be possible to link into the MDS communication path[1] at some point, probably somewhere in the RPC module. In this module it is easy to filter the incoming and outgoing requests (structures) of the MDS and to distribute them to Transis.

[1]The path of the MDS is similar to the path of the MDC shown in Figure 2.3

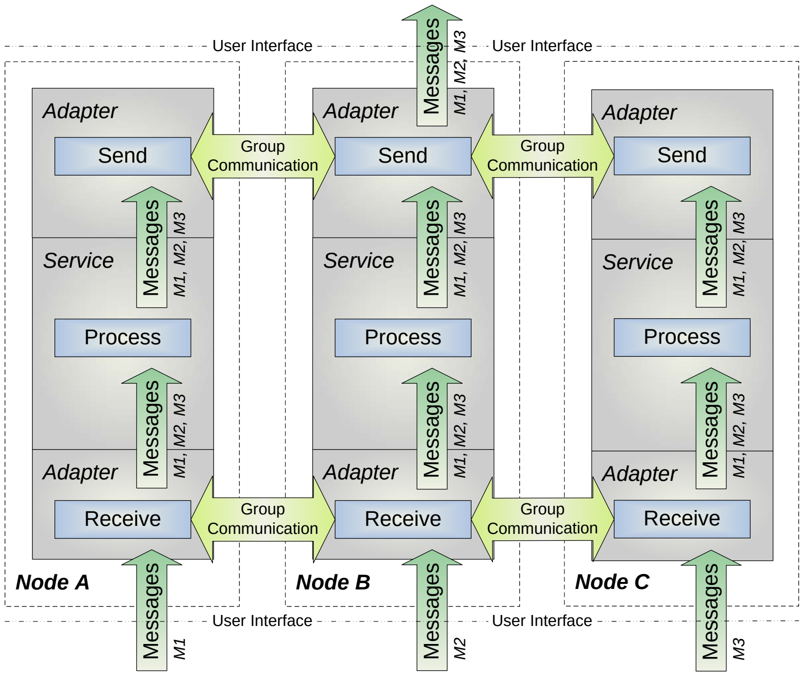

Figure 2.8.: Scheme of Internal Replication Method

The core problem in the design of an internal solution is not Lustre, it is Transis. Transis is a user-space program. Transis consist of a daemon running in userspace and a library to be linked to the user application. This application calls the library functions and the library calls the daemon, which does the group communication work. The problem is that Lustre is made up of kernel modules and runs therefore in kernel space. In order to include the group communication direct into the Lustre code, the Transis library needs to be linked into kernel space. This is not possible because the Transis library uses functions which are only available in user-space. The only workaround to this problem is to redesign the Transis library for kernel space. This is theoretically possible, but due to the limited time of this project not reasonable.

The other problem is the development of the prototype itself. Because the group communication system is implemented directly into the RPC module, the prototype becomes a new version of Lustre. This means, to test changes made during the development process Lustre has to be rebuild and reinstalled first. This takes a lot of time. Furthermore, the whole development of the prototype becomes kernel development. This is also not reasonable.

To summarize, this method could theoretically be implemented, but the goal within the scope of this project will be to design an external replication.

2.2.2 Feasibility of External Replication

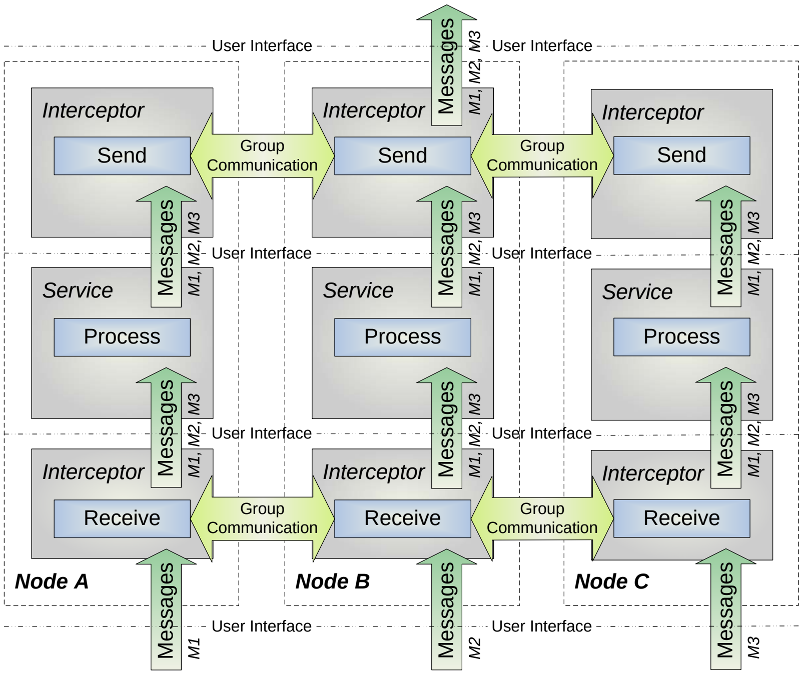

Figure 2.9.: Scheme of External Replication Method

The external replication method is shown in Figure 2.9. In this solution the group communication system is build like a shell around the MDS. The group communication system is placed into the Client-MDS communication path as an intermediate communication process, see Figure 2.12. This process intercepts the calls over the network to and from the MDS and distributes the TCP packages to Transis. As a result there is no need to touch the MDS code.

The disadvantage of this method is higher latency time due to inter-process communication. There is also the need to know the exact communication protocol and format between the MDS and the client.

The problem of the internal replication is not present in this solution. The interceptor process runs as normal user space application and thus there is no problem in linking the Transis library into it.

To realize this approach, Lustre must be configured in a way that differs from the standard setup. The Lustre setup, its network components and the tasks of each component, are configured in one XML file. Lustre assumes that every node in the file system uses the same XML file for the configuration and startup. However, there seems to be no big problem to use different XML files for different nodes. That way the external replication may be realized.

This method is feasible within the limits of the project and the objective of the master thesis now is to use this replication method for the prototype design.

2.3 System Design Approach

Two projects have implemented prototypes of active/active HA solutions so far. The aims of these the projects and their results are explained in Section 1.2. Using the experience of these preceding projects a first prototype design can be developed. This design provides the basic HA functionality and has to be adjusted to the special needs of Lustre later.

2.3.1 Standard Lustre Setup

Figure 2.10 shows an example of the standard setup of Lustre. For the development of the project this setup is used. It is only a very minimal setup of Lustre nevertheless it provides the full functionality of the file system.

The project setup of Lustre uses three computing nodes for the three main components of Lustre. One node (usr1) is used as client and mounts the file system. From this node the functionality of the prototype can be tested and performance tests of the file system can be run. On the second node two OSTs are installed. Each OST is an independent partition on the disk. The third node runs as MDS for the file system. The MDS stores its data on a partition of the disk as well.

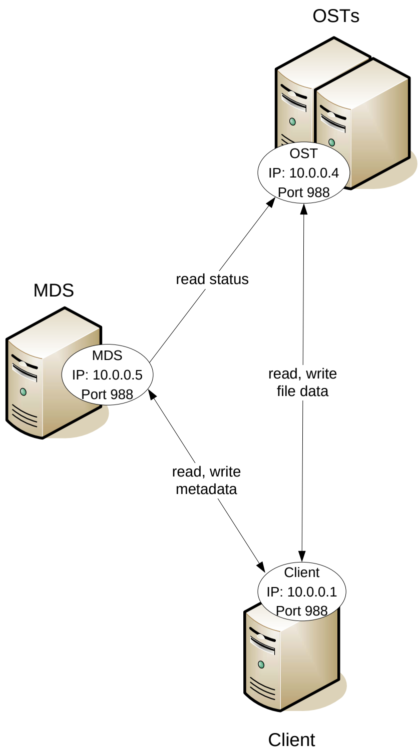

Figure 2.10.: Standard Lustre Setup

This approach is sufficient to develop the HA prototype. The full file system functionality can be tested with this setup and the separation of the components to different nodes allows easy handling and analyses.

2.3.2 Lustre using External Replication of the MDS

According to the Lustre design shown in Figure 1.1, in Section 1.1.2, the MDS is one component of the entire file system. This component needs to be replicated. To achieve a virtual synchrony environment the group communication system Transis has to be put around the MDS group.

Figure 2.11 shows the scheme of an active/active HA solution. A process (MDS) is replicated on several nodes. The group communication system (Transis) is placed before and behind this process. Before the process, Transis receives all requests and distributes

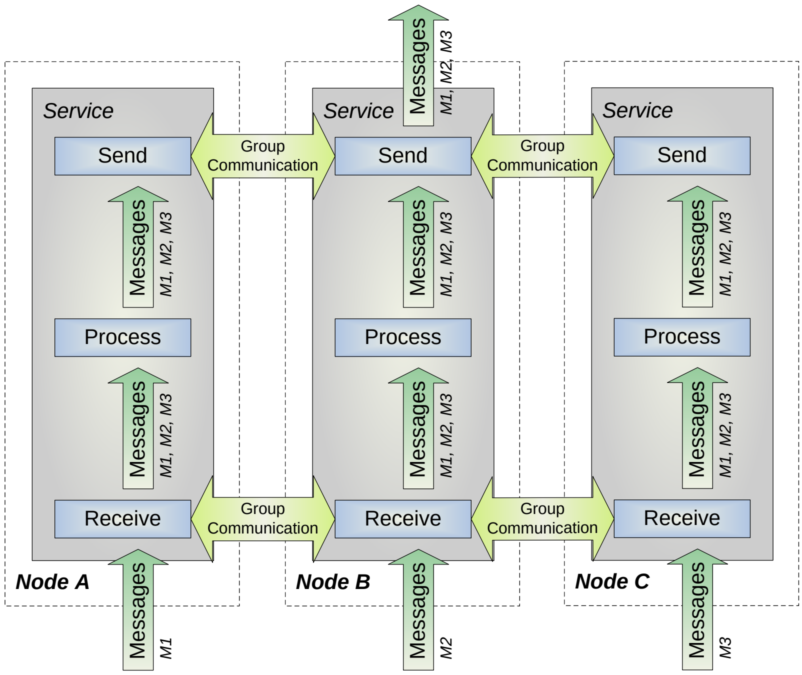

Figure 2.11.: Scheme of Active/Active HA

them to all nodes. In this step it ensures total message order. This means, all messages are delivered in the same order to all nodes. Thus, the MDS group runs in virtual synchrony. Then the requests are processed by all nodes independently. This however causes the single instance execution problem. The MDS group processes as much responses as members the group has. To overcome this hurdle the group communication system is placed behind the process as well. Here it receives the responses of the processes again. It makes sure each response is delivered only once to the system.

The system design of the preliminary prototype is shown in Figure 2.12. The major difference from the normal Lustre configuration is the group communication system Transis. The Transis daemon runs on each MDS node. This daemon provides the group communication functionality. The daemon can be accessed with the Transis library. In order to distribute the incoming messages to the Transis daemon and to receive messages from the daemon an interceptor program, implementing the Transis library, has to be written.

The interceptor implements all needed group communication and routing functionality.

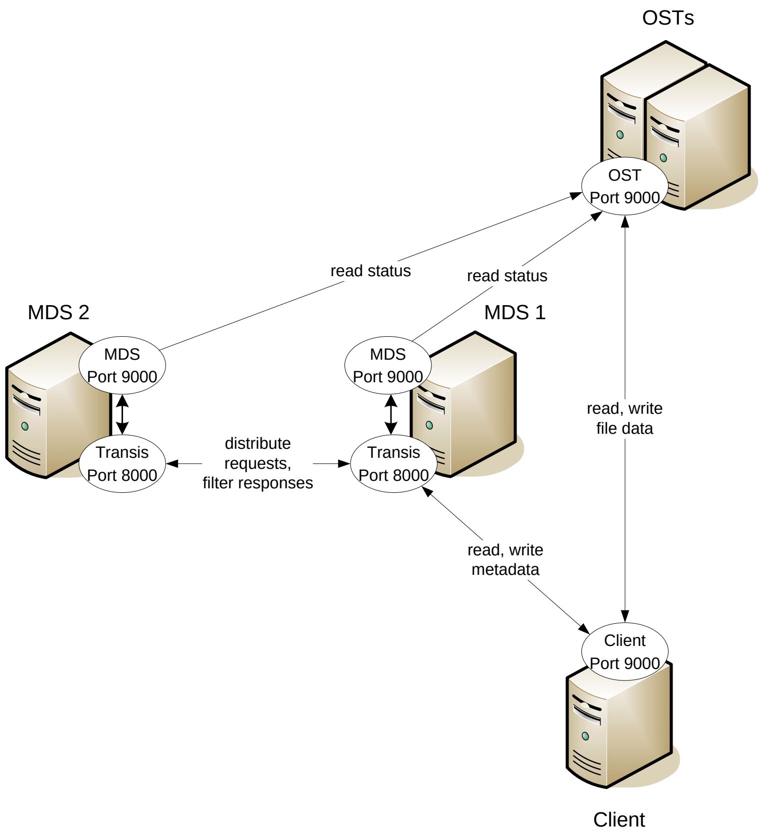

Figure 2.12.: Preliminary System Design

This program opens a port (e.g. 8000) to accept connections from the Lustre clients.

The MDS itself listens on its own port (e.g. 9000) for incoming messages from the clients, which are rerouted through the interceptors.

To get the file system working with interceptors, Lustre must be configured in a proper way. This may be done with the config XML file from Lustre, described in Section 3.1. Lustre reads its complete setup from one XML file for all nodes. The rule, to use one XML file for all nodes must be ignored. To configure Lustre, one XML file for each node has to be created. The XML files used to configure the MDS and the OST nodes have to set up the MDS on port 9000. Whereas the XML file used to configure the Client node has to set up the MDS on port 8000. Thus, the clients expect the MDS there and send the requests to this port. On this location (the MDS node on port 8000) however, the interceptor program is running. It catches the messages and routes them through the Transis daemon. The daemon orders all incoming messages and distributes them to all MDS nodes. The ordered messages are sent back by the daemon to their respective interceptor program. After this step, each interceptor forwards the messages from the daemon to the MDS running on his node.

The procedure of the response from the MDS to the client works the same way. All MDS nodes produce their result (all the same of course) independently. The MDS nodes send the result to their respective interceptor. The interceptor forwards the messages to Transis. Transis orders all messages and sends them back to the interceptor. The interceptor receives all those equal messages. To overcome the single instance execution problem, the interceptor has to analyse these messages and to make sure only one of all equal messages is forwarded to the client.

Furthermore, the interceptor program should be capable of dynamic group reconfiguration. This could be achieved with help of the Transis daemon. This daemon is aware of group configuration changes and sends notifications to the interceptor. The interceptor code has to handle those messages and to help in setting up new members in the MDS group properly.

Finally, the client code has to be adjusted to allow failover to new group members and to avoid it to broken group members that no longer remain in the MDS group and therefore not share the global state anymore.

2.4 Final System Design

Due to the difficulties pointed out in Section 3.3, the proposed preliminary design of the prototype has to be adjusted to the needs of Lustre. To meet the requirements of the project, two prototype designs have been developed.

2.4.1 Prototype 1

The first prototype design will replicate the MDS in an active/active fashion and is capable of dynamic group reconfiguration.

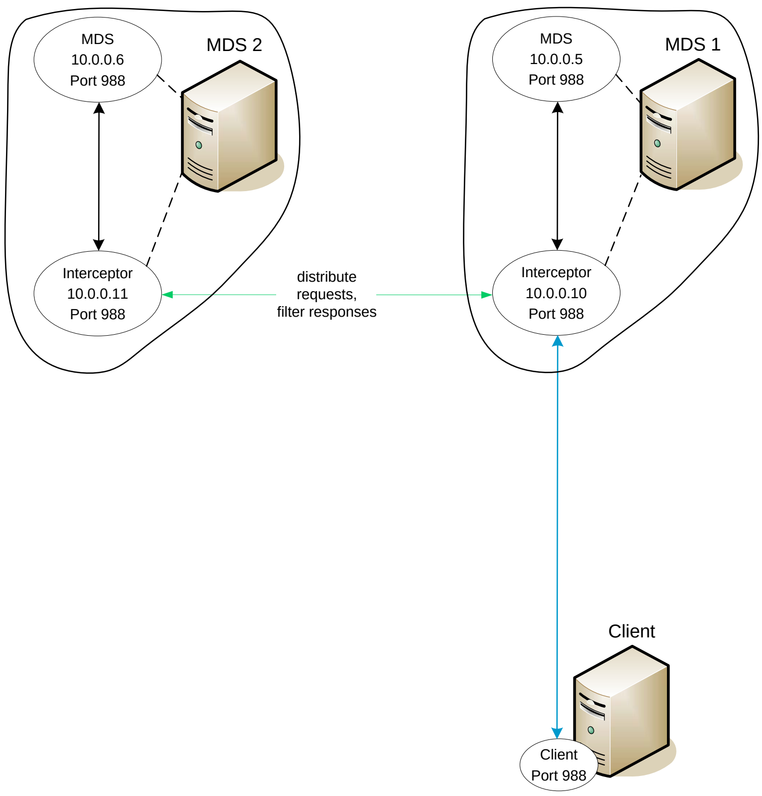

Figure 2.13.: Prototype 1

This redesign of the preliminary prototype will sort out a couple of problems caused by Lustre limitations. The problems solved are the following:

- no use of individual ports for Lustre components
- no routing of Lustre messages
- inflexible Lustre system configuration

The preliminary prototype runs the Lustre MDS and the interceptor process together on one node. Each process opens an individual port for incoming communication. This is needed to distinguish between both communication paths and to route messages to the individual components. Lustre's limitation to use only port 988 for all components, renders the proposed solution impossible. There is no way of configuring a client node to connect to the interceptor (e.g. port 8000). One possibility to solve this problem is to start the interceptor process on an own node. This way the interceptor could be started on port 988 as well. The client can be configured to expect the MDS on the interceptor node and to connect to this node. The downside of this solution is a significant performance impact. The communication from the interceptor to the MDS isn't local anymore, but goes now over the network. Also an own node for each interceptor is needed. This is not reasonable to do. The better solution to this problem is to make use of IP aliasing. With IP aliasing two network addresses can be bound to one network card. The advantage is that each address has its own ports and the communication between both addresses is still local. The latency time caused by communication between the both addresses is minimal (see performance tests for details in Section 3.4.2).

Using IP aliasing two addresses (e.g. 10.0.0.5 and 10.0.0.10) can be run on one node with one network card. That way the port 988 can be used for both servers. The MDS runs on address 10.0.0.5 and the interceptor runs on address 10.0.0.10.

Lustre itself can be configured as described in Section 3.1. The XML files need to be edited in a way that the interceptor is the client for the MDS and vice versa. If configured properly, the Lustre MDS and clients accept messages from the interceptors.

In order to provide full HA functionality and to avoid dropped messages due to routing, the prototype must make use of the message routing principles described in Section 4.1.

To provide a complete HA solution the prototype needs to be capable of dynamic group reconfiguration. With this functionality the prototype is able to start and join new MDS nodes in order to replace failed group members or to increase the level of availability. The other task of dynamic group reconfiguration is to deal properly with failing group members. This technology and its implementation are described in Section 4.3.

Finally, the single instance execution problem is solved using a shared connection table. This approach is described in more detail in Section 4.2.

The milestones listed in Section 1.4 are used to judge the project progress. Below listed are the milestones that are fulfilled with functionality provided by this prototype design:

- **A4** solution for single instance execution problem

- **A5** MDS service stays available, as long as one node is up
- **B6** replication of the MDS on more than two nodes
- **B8** new MDS nodes can dynamically join

2.4.2 Prototype 2

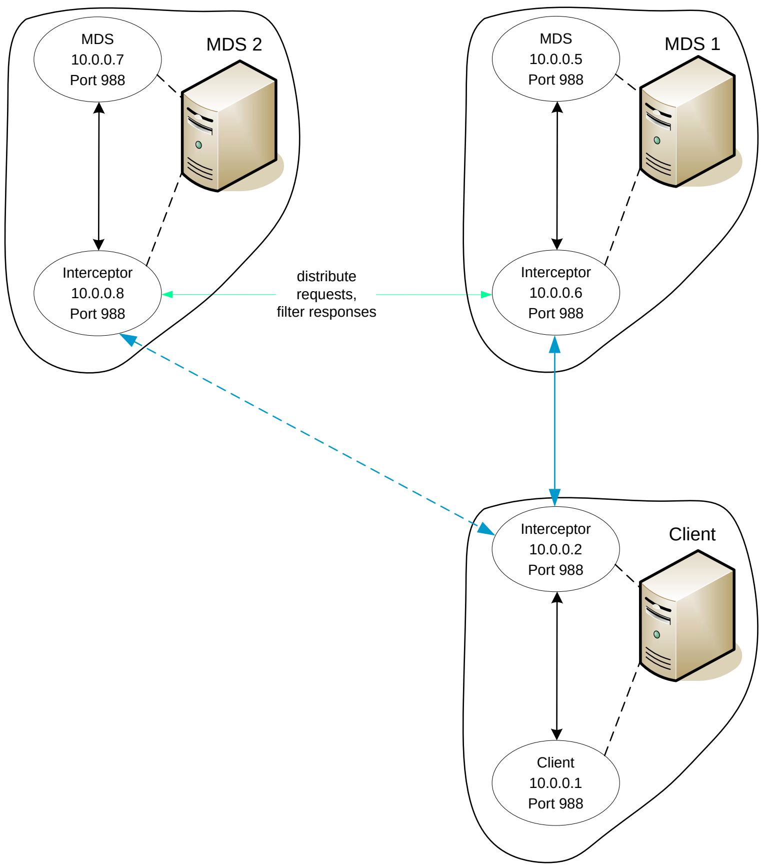

Figure 2.14.: Prototype 2

This prototype design is an extension of the Prototype 1. The first prototype still suffers from a lack of connection failover. This problem causes errors to clients if the connected MDS fails. To mask this kind of error from the user (client) is task of this prototype design. The connection failover procedure is described in more detail in Section 4.4.

In order to mask this error from the user, the client has to reconnect to another available MDS interceptor. Therefore, the client needs to hold a list of available MDS interceptors.

Due to the already mentioned reasons in Section 2.2.1 it is not reasonable to implement the needed functionality into the client code directly. The better solution is to use IP aliasing for the client as well. Thus, the client has its own interceptor.

This client interceptor routes the client messages directly without the use of Transis according to Section 4.1. The only difference is that the client interceptor forwards the messages to the chosen MDS interceptor instead to the MDS.

To get Lustre working with client interceptors as well, it has to be configured in a different way. The exact configuration is described in Section 3.1.

The additional milestones that are fulfilled by this prototype design are:

- **B7** client connects to other MDS node if own fails
- **B9** client table of MDS nodes is updated at runtime

This prototype design is capable of all proposed criteria and meets all requirements of the project.

3 Implementation Strategy

3.1 Lustre Configuration

The Lustre file system is configured with one XML file. This file is generated with help of a config script. The script used to configure Lustre for the development of the prototype is shown in Figure 3.1.

First, the user has to define all nodes the file system will use. The development setup uses three nodes (mds1, ost1, usr1). The next step is to define the network names of the nodes. For easy handling they should be the same, like the node names. Now, the file system components can be configured and assigned to the nodes. In the development setup node mds1 is configured as MDS. Node ost1 runs two OSTs (ost1 and ost2). All OSTs are bound together to one Logical Object Volume (LOV). For the MDS and OSTs, partitions for saving the file system metadata and data must be specified. For the prototype development files instead of partitions are used. The needed size of the file can be specified. After creation the files are mounted and behave like partitions. The last thing to configure, are the clients. The client node must know what LOV, MDS, and mount point to use.

The port each component uses for incoming connections can be edited directly in the XML file or in the config script with the option –port, e.g., to choose port 8000 the phrase "–port 8000" has to be put into the configuration line of the component. However, these configurations are completely ignored. Lustre uses one port number given in the source code for all components.

After the file system is configured the script can be run and Lustre generates the XML

```
#!/bin/sh

# Script configuring Lustre on three nodes

rm -f  config.xml

# Create nodes
lmc -m config.xml --add node --node ost1
lmc -m config.xml --add node --node mds1
lmc -m config.xml --add node --node usr1


# Add net
lmc -m config.xml --add net --node ost1 --nid ost1 --nettype tcp
lmc -m config.xml --add net --node mds1 --nid mds1 --nettype tcp
lmc -m config.xml --add net --node usr1 --nid usr1 --nettype tcp


# Configure MDS
lmc -m config.xml --add mds --node mds1 --mds mds1 --fstype ldiskfs \
                            --dev /lustretest/mds-mds1 --size 500000

# Configure LOV
lmc -m config.xml --add lov --lov lov1 --mds mds1 --stripe_sz 1048576 \
                            --stripe_cnt 0 --stripe_pattern 0

# Configure OSTs
lmc -m config.xml --add ost --node ost1 --lov lov1 --fstype ldiskfs \
                  --dev /lustretest/ost1 --size 1000000 --ost ost1
lmc -m config.xml --add ost --node ost1 --lov lov1 --fstype ldiskfs \
                  --dev /lustretest/ost2 --size 1000000 --ost ost2

# Configure client
lmc -m config.xml --add mtpt --node usr1 --path /mnt/lustre --mds mds1 --lov lov1
```

Figure 3.1.: Lustre Configuration Script

file. The name of the XML file is also defined in the config script. This XML file has to be used to start up every node in the file system. First the OSTs, then the MDS, and at last the clients. The from the config script generated XML file is appended in Section A.3.

Now, a normal Lustre setup like shown in Figure 2.10 is configured. To get Lustre working with interceptors the configuration has to be adjusted.

In spite of Lustre's rule to use the same XML file for all nodes, a XML file for every node needs to be created. The approach to write an own config script for every node and to generate the different XML files doesn't work, because Lustre generates different UUID keys for the same nodes and the file system refuses its own messages. The way to go is to edit the XML file directly. The important points are the nid tags in the file. The nid tag holds the network name (or network address) of the defined node. The network names of all available nodes are defined and assigned to IP addresses in the file `/etc/hosts`. Lustre uses the network names given in the nid tags to address the file system components. These nid tags need to be adjusted to the desired setup.

Changes in the respective XML file of the components in case of Prototype 1 (see Figure 2.13):

- **OST**: no changes
- **MDS**: nid of Client node needs to be changed to MDS interceptor
- **Client**: nid of MDS node needs to be changed to MDS interceptor

Changes in the respective XML file of the components in case of Prototype 2 (see Figure 2.14):

- **OST**: no changes
- **MDS**: nid of Client node needs to be changed to MDS interceptor
- **Client**: nid of MDS node needs to be changed to Client interceptor

3.2 Messaging Mechanisms

The communication of the prototype is realized via sockets. The TCP protocol is used. The implementation of the communication could be done in various different ways. Goal is to find the fastest and most stable solution.

One general question is what type of sockets to use. Both, blocking/non-blocking sockets have been tested during the development of the prototype.

Non-blocking sockets have the advantage that the server doesn't wait for a message on one socket and blocks until a message arrives. This behaviour could improve performance due to no delay times on other sockets with already waiting messages. However, blocking

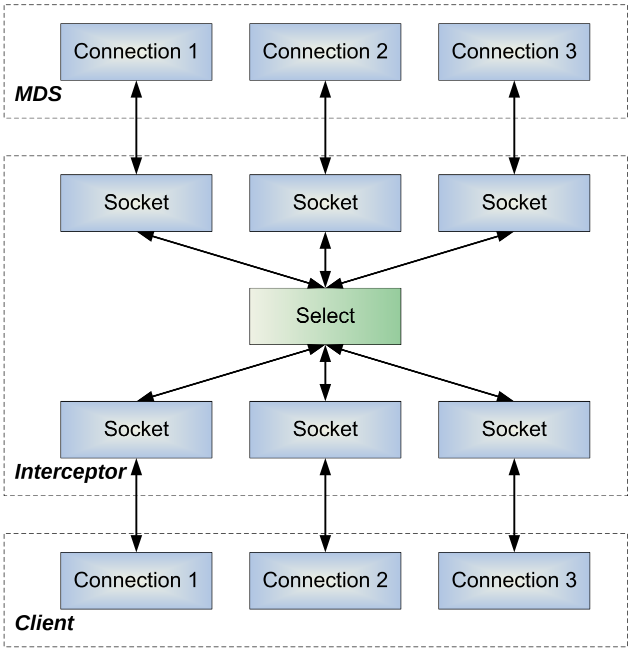

Figure 3.2.: Message Forwarding using one Thread

sockets have the advantage that they are very likely to deliver and receive the complete message. This results in easier handling.

Another important fact is that blocking sockets are more performance friendly. In a non-blocking receive procedure the program polls the socket until a message arrives. For this process the program uses the cpu all the time. In a blocking receive procedure the process is set sleeping until the message arrives. This saves resources as it gives the cpu free for other tasks.

The decision for the prototype implementation falls to blocking communication. The downside of blocking communication, the possibility of blocking and waiting for one socket and ignoring another with already available messages is sorted out with the use of the `select` system call. The `select` call listens to all given sockets for incoming data. If one socket has a message available, `select` gives this socket back to the program. The program just has to go to the socket and can get the message. Using `select` there is no blocking of sockets because every time a socket is called it is ensured the socket has a message available. Of course, the `select` call is blocking itself. Thus, the process is set to sleep if no messages are available and no cpu time is wasted.

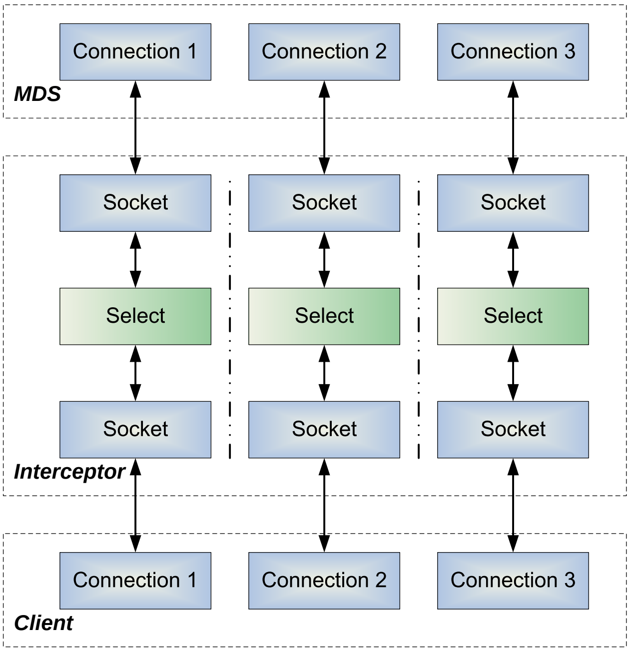

Figure 3.3.: Message Forwarding using Multithreading

If `select` gives back a socket, it is most likely that one complete message can be processed. This is due to Lustre's behaviour to send one request in one message. Before the message is sent, Lustre assembles all data and sets up the header and puts the request in the payload. Also the size of the message is limited by Lustre (Payload max. 4KB, see Section 2.1.2). When the `select` call gives a socket back to the program, the message has arrived at the socket. Now, the header and the payload can be read out. If the message is received without error it can be routed to its destination.

The other decision to make is how to use threads. One possibility is to use only one thread. This thread deals with all connections. Figure 3.2 shows this method in the example of the three connections of one client. Here, one `select` call checks all sockets for incoming messages. This method works comletely in serial. It has the disadvantage of worse performance in relation to multiple threads for communication and the advantage of easier code structure.

The other approach is to use one thread per communication path. Figure 3.3 shows this method. For each connection a thread is started. This thread holds two sockets controlled by a `select` call. The `select` call checks whether the client or the MDS wants to send a message. The advantage is that all connections can be processed in parallel. This approach is faster than the serial one with only one thread. It would be the preferred method for direct routing. However, the performance plus due to this method is minimal and tests have shown no significant difference between both methods.

For the prototype design the fist method using one thread for communication has been chosen. The reason is Transis. The interceptor needs to route all messages through Transis. Transis however runs not stable in a multithreaded environment and is likely to produce errors. With the help of mutual exclusion locks the Transis calls have to be serialised. Thus, the entire communication of the interceptor is inherently serial and the single threaded method can be chosen anyway.

3.3 Implementation Challenges

The design of Lustre is complex and tightly integrated. This makes adjustments to the prototype design difficult.

Implementation challenges for prototype development:

- no use of individual ports for Lustre components
- inflexible Lustre system configuration
- no routing of Lustre messages
- distributed locking mechanisms within Lustre
- existing active/standby failover behaviour of the MDS
- only three connections per node allowed

No use of individual ports for Lustre components
Lustre allows to configure the port for components individually in its config file. However, this capability is kind of leftover from former Lustre versions and not used anymore. Now, Lustre uses one hard coded port. As a result, it is not possible to assign individual ports to components.

This limitation has a significant impact on the preliminary design. Solution to this problem is the use of IP aliasing as described in Section 2.4.

Inflexible Lustre system configuration

Lustre needs to know its setup in advance. A config script is therefore written, configuring the entire file system. From this config script a XML file is generated. This XML file is used to start Lustre.

Due to the Lustre security concept only messages from nodes configured in the XML file are allowed. The problem is that in a normal Lustre configuration all messages from the interceptors are rejected. To get Lustre working with interceptors the file system must be configured differently and not in the intended way. How this configuration is done is described in Section 3.1.

No routing of Lustre messages

As part of the Lustre security concept routing of messages is forbidden. Messages that are not sent directly are dropped.

To route messages is essential for the prototype. To be able to route messages the prototype has to look into the messages and to trick Lustre. It has to adjust the messages in a way that Lustre thinks the messages are sent directly. This procedure is described in Section 4.1.

Distributed locking mechanisms within Lustre

Lustre uses only one MDS to serve thousands of clients. To hold the metadata state of the file system consistent distributed locking mechanisms are used.

These mechanisms however cause problems in the setup of an MDS group. The problems to implement an active/active MDS group are described in more detail in Section 4.3.

Existing active/standby failover behaviour of the MDS

Lustre provides an active/standby HA solution. In the scope of this solution it is possible to shutdown the running MDS and to start the backup MDS. The shutdown is useful to commit all pending requests to disk.

The problem is that only one MDS can be running at a given time. It is not possible to start the backup MDS as long as the active MDS is still running. The other problem is that only two MDS can be configured. These limitations render the setup of the MDS group impossible. To run a proper MDS group in an active/active fashion, it is needed to start and run two and more MDS at the same time.

These limitations also prevent the dynamic group reconfiguration from proper functionality.

Only three connections per node allowed
Lustre is designed to accept only three connections from one IP address.

This causes problems to run the prototype with multiple clients. In the prototype design all clients are routed through one interceptor. This would lead to more than three connections from the interceptor IP address. If a second client connects to Lustre, the interceptor opens a fourth, fifth and sixth connection to the MDS. This would kick out the first three connections of the fist client. To overcome this problem one interceptor on the MDS side for each client would be needed. This is not reasonable to do. As a result, the prototype design and tests use just one client.

3.4 System Tests

The process of software testing is used to identify the correctness, completeness and quality of the developed software. Testing is nothing more but criticism and comparison towards comparing the state and behaviour of the software against a specification. [1]

The specification for the prototype is given in the beginning of this work in Section 1.4.

All tests are performed in a dedicated cluster environment setup for the development and tests of the Lustre HA prototype. Each node in the cluster has the following properties:

- Hardware

  CPU Dual Core Intel Pentium 4 3.0GHz

  Memory 1024MB

  Network Ethernet 100MBit/s and 1GBit/s, full duplex

- Software

  Operating System Fedora Core 4

  Kernel Red Hat 2.6.9-42.0.3, patched with Lustre

  C Compiler gcc version 3.4.2 (Red Hat 3.4.2-6.fc3)

  Transis daemon and library version 1.03, patched with Fast Delivery Protocol

  Lustre version 1.4.8

To evaluate the prototype and its components different setups of the file system and prototype are used. The following listed prototype configurations are especially valuable for performance tests.

- Standard Lustre
- MDS Interceptor
- Client Interceptor
- MDS Interceptor and Client Interceptor
- Prototype 1
- Prototype 2

Standard Lustre
This is the standard Lustre setup, as shown in Figure 3.4, without any changes or manipulations. Lustre is configured as intended on three nodes. One node runs two OSTs. The second node provides the MDS and the third node is the client of the file system and mounts Lustre. This setup is used to get the performance of the standard file system to determine the delay caused by the prototype.

MDS Interceptor
Additionally to the original Lustre, this setup uses one interceptor on the MDS side. The setup is shown in Figure 3.5. The MDS interceptor makes no use of the group

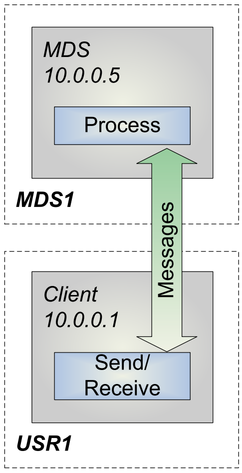

Figure 3.4.: Test Setup: Standard Lustre

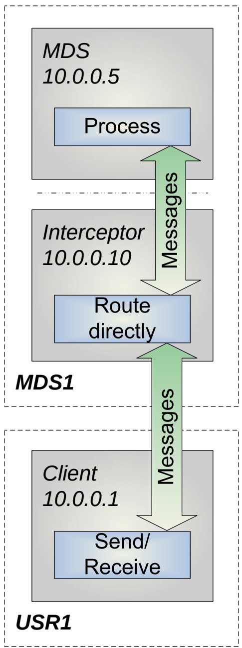

Figure 3.5.: Test Setup: MDS Interceptor

communication facilities. Thus, only the delay time caused by the message routing mechanisms on the MDS side can be measured.

Client Interceptor

This is a similar setup as the previous, except that this time the interceptor is located

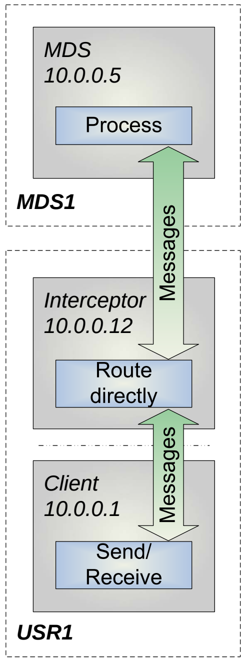

Figure 3.6.: Test Setup: Client Interceptor

on the client side, see Figure 3.6. Here again the interceptor makes no use of the group communication facilities. With this configuration the delay caused by the message routing mechanisms on the client side can be measured.

MDS Interceptor and Client Interceptor

This setup is a combination of the last two. It makes use of both interceptors, see Figure 3.7. That way, the delay caused by the message routing mechanisms on the client and the MDS side can be measured.

Prototype 1

This setup is the standard Lustre with use of an interceptor on the MDS side. This time the interceptor routes the messages through Transis, see Figure 3.8. This setup should allow to determine the delay caused by the group communication facilities. This setup is tested in three different steps. One time with one MDS group member, one time with two, and one time with three. These configurations allow to measure the delay time caused by the group communication facility itself as well as the impact of several group members on the performance due to the acknowledgement process.

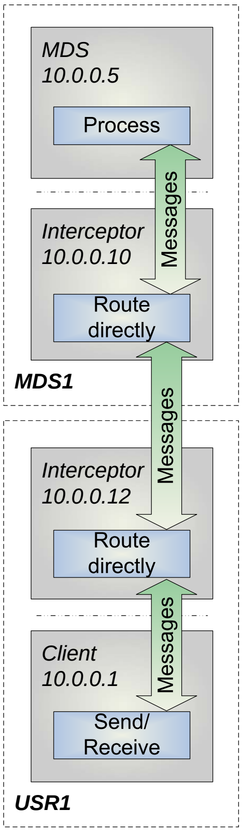

Figure 3.7.: Test Setup: MDS Interceptor and Client Interceptor

Prototype 2

This test series uses both interceptors on the MDS and the client side respectively, see Figure 3.9. The interceptor on the client side just routes the messages directly. The interceptor on the MDS side routes the messages through the group communication facilities. This test series measures the impact of up to three group members and allows conclusions about the performance of a solution capable of connection failover.

The client node is used to test the file system. Here the provided functionality of Lustre is accessible. Files can be created, deleted, read, and written. The usage and the free memory of the file system can also be shown.

For the tests an own benchmark program has been written. Its source is provided in Appendix A.2. The program creates, reads the metadata, and deletes a given number of files. It does this in a given number of test runs and builds the arithmetic mean

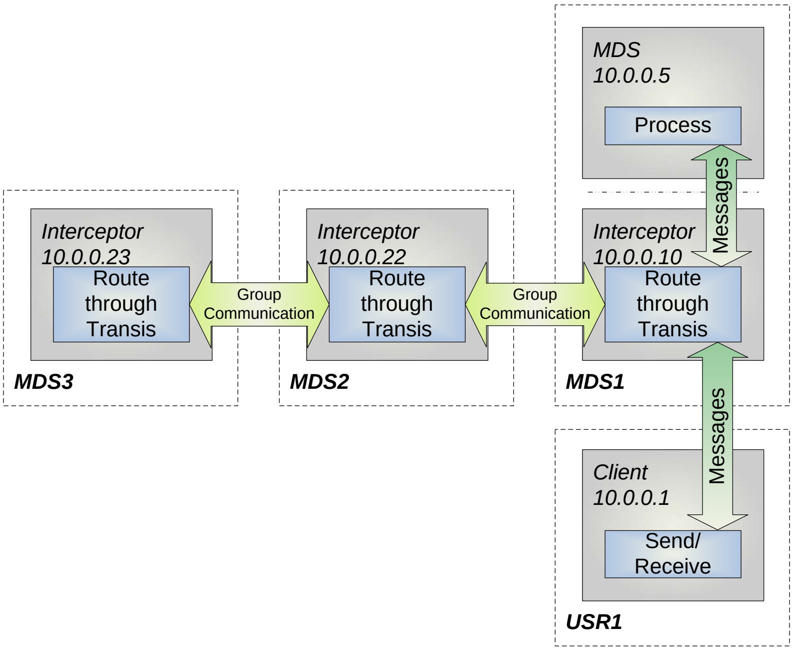

Figure 3.8.: Test Setup: Prototype 1

values. From the measured times it calculates the operations per second the file system is capable of. It also calculates the time needed for one operation.

3.4.1 Functionality

Due to restrictions given by Lustre the functionality tests could only performed partwise. Goal of this section normally should be to test and evaluate the proper functionality of the prototypes in terms of high availability. However, a complete HA version of the prototype implementations is not running. This limits the possibilities for the functionality tests. For instance, connection failover cannot be tested. What can be done is to test the developed parts of the solution for their proper functionality.

The functionality of the developed prototypes that can be tested:

- Message Routing, one MDS Node
- Group Communication System

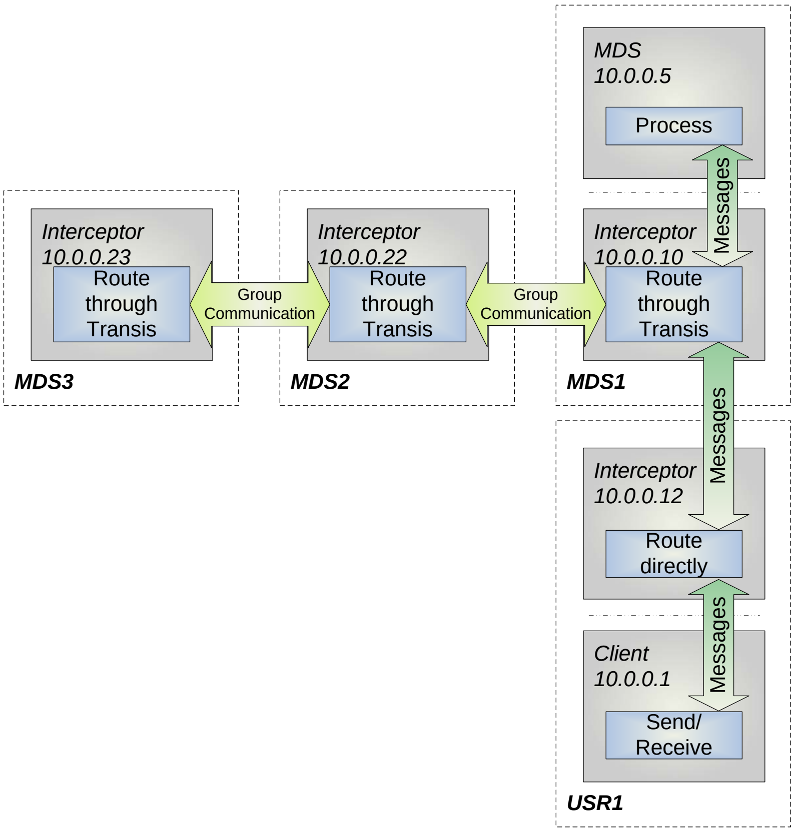

Figure 3.9.: Test Setup: Prototype 2

- Message Routing, multiple MDS Nodes
- Single Instance Execution Problem
- Mount Lustre
- Unmount Lustre
- Lustre File System Status
- File Operation: read
- Lustre File Operation: write
- Lustre File Operation: create
- Lustre File Operation: delete

Message Routing, one MDS Node

This test evaluates the correct function of the message routing of the prototypes described in Section 4.1 Message Routing. This part can be tested with the simplest test setup "MDS Interceptor". In this setup one interceptor is placed in the MDS communication path. The interceptor just forwards and adjusts the messages as described. If the message routing works correctly, Lustre accepts the interceptor and mounts the file system. The same test must be done with client interceptor as well. Because the client interceptor uses the same routing algorithm, Lustre should mount properly.

pass: ✓

Group Communication System

The correct function and implementation of the group communication system into the prototype also needs to be tested. This can be done in two steps. The first is to test the group communication system alone on one node. For this test the setup "Prototype 1" with one group member can be used. Here, the MDS interceptor uses Transis to route the messages. If the group communication system is included correctly, the interceptor should start the Transis "MDS Group" and Lustre should mount properly. The second step is to start another interceptor on a second node. This interceptor should join the "MDS Group" if everything goes right.

pass: ✓

Message Routing, multiple MDS Nodes

This test is an extension of the first two tests. Here the setups "Prototype 1" and "Prototype 2" with three group members are used. To evaluate if the message routing of all three nodes works properly, own servers that act as MDS must be used. One node starts the Lustre MDS. The other two nodes start their own server. These servers open a connection at the port 988 and receive messages like the MDS would do. In this step these own "fake" MDS servers check the message header for the correct source and destination. To pass the test, Lustre should mount properly and the own servers should receive messages as well and report no errors.

pass: ✓

Single Instance Execution Problem

The correct function of this part can be tested with an extension of the own "fake"

MDS servers. The same setups like in test "Message Routing, multiple MDS Nodes" are used. The difference is, that the "fake" MDS bounce received messages back to their interceptors. That way, they cause own output messages. If the single instance execution problem is solved correctly, duplicated output messages should not be sent to the client and thus not confuse Lustre. This test is passed if Lustre mounts and works properly.

pass: ✓

The following tests show the proper functionality of the Lustre file system with the prototype implementations. For these tests the "Prototype 2" setups with three group members is used. Also own "fake" MDS servers, as described in the "Single Instance Execution Problem" test, are used. This setup is the closest possible to a working production type HA solution for Lustre.

Mount Lustre

Lustre should be capable to mount without errors.

pass: ✓

Unmount Lustre

Lustre should also be capable to unmount and to shutdown without errors.

pass: ✓

Lustre File System Status

During use of Lustre the command "lfs df -h" should show the usage state of all OSTs and the MDS.

pass: ✓

File Operation: read

Test of the file system capability to read files.

pass: ✓

Lustre File Operation: write

Test of the file system capability to write to files.

pass: ✓

Lustre File Operation: create

Test of the file system capability to create files.

pass: ✓

Lustre File Operation: delete
Test of the file system capability to delete files.
pass: ✓

The functionality listed below could not be tested. It is the HA functionality in general. Due to the fact that it is impossible to run two MDS at the same time no real HA solution could be tested.

The functionality of the prototype implementations that cannot be tested:

- dynamic group reconfiguration
- connection failover
- saved state of file system as long as one node is up

The results of the functionality tests give proof of working components, like interceptors or the group communication system. But an entire HA solution of Lustre could not be tested. Even though the working components do not provide the functionality of an HA prototype, they nevertheless consist of almost everything a working solution would need. The fact that Lustre is working with the implemented solution makes performance tests possible. These tests will allow to draw conclusions about the impact a full working HA solution would have on the performance of Lustre.

3.4.2 Performance

As described in the functionality tests the prototypes do not provide the full functionality of a HA solution. However they are very close to this solution in terms of performance. A full working HA prototype would have almost the same impact on performance, like the implemented Prototype 2 in this project. Thus, these tests allow considerations about the performance a full HA solution.

Tested are the different setups described in the beginning of the test section.

For all performance tests the file system cache was deactivated. This step is essential to compare the performance of the different test setups. All tests have been done in two different network setups. One time with 100MBit and one time with 1GBit network.

To evaluate the performance a benchmark program has been written. The source of the program is attached to this work in Appendix A.2. The program creates a given number of files, reads the metadata of the files, and eventually deletes the files. In order to evaluate the performance the program takes the time needed for each operation. To achieve a measurement with a low error the program performs a given number of test runs and calculates the mean time for each operation from all test runs.

Lustre High Availability Prototype 100MBit Test Runs

Operations per second	100 Mbit					
	1 file			100 files		
	create	read	delete	create	read	delete
Standard Lustre	538.199	462.389	1,129.114	551.799	452.459	1,721.659
MDS Interceptor	6.196	23.632	12.352	8.184	12.205	24.437
Client Interceptor	6.187	22.908	12.368	8.178	11.978	24.349
Client Int. and MDS Int.	6.163	23.386	12.290	8.135	12.136	24.314
Prototype 1, 1 Group Member	6.103	11.840	12.165	8.030	8.052	24.064
Prototype 1, 2 Group Members	6.104	11.846	12.170	8.026	8.060	23.775
Prototype 1, 3 Group Members	6.108	11.844	12.165	8.025	8.062	23.964
Prototype 2, 1 Group Member	6.056	11.758	12.094	7.966	8.051	23.895
Prototype 2, 2 Group Members	6.051	11.732	12.047	7.964	8.045	23.889
Prototype 2, 3 Group Members	6.037	11.782	12.092	7.918	8.046	23.894

Time taken for one operation (msec)	100 Mbit					
	1 file			100 files		
	create	read	delete	create	read	delete
Standard Lustre	1.858	2.163	0.886	1.812	2.210	0.581
MDS Interceptor	161.403	42.315	80.957	122.191	81.936	40.921
Client Interceptor	161.637	43.653	80.854	122.285	83.485	41.069
Client Int. and MDS Int.	162.248	42.760	81.370	122.929	82.401	41.129
Prototype 1, 1 Group Member	163.859	84.463	82.202	124.538	124.186	41.557
Prototype 1, 2 Group Members	163.827	84.418	82.172	124.593	124.074	42.061
Prototype 1, 3 Group Members	163.707	84.433	82.202	124.607	124.041	41.729
Prototype 2, 1 Group Member	165.125	85.050	82.686	125.529	124.211	41.849
Prototype 2, 2 Group Members	165.248	85.240	83.009	125.558	124.299	41.860
Prototype 2, 3 Group Members	165.647	84.874	82.698	126.298	124.290	41.852

Figure 3.10.: Performance Test Results 100MBit

The results of the test runs are shown in the Tables 3.10 and 3.11. At first glance the significant performance impacts of all HA solutions are striking. The default Lustre setup performs up to 89 times faster than the tested prototype setups. This performance impact is odd and not expected. The JOSHUA project [21] achieved latency times of

Lustre High Availability Prototype 1GBit Test Runs

Operations per second	1 Gbit					
	1 file			100 files		
	create	read	delete	create	read	delete
Standard Lustre	622.247	550.658	1,330.973	636.749	520.497	1,951.101
MDS Interceptor	6.212	23.828	12.380	8.206	12.288	24.485
Client Interceptor	6.196	22.219	12.382	8.194	11.880	24.379
Client Int. and MDS Int.	6.169	23.300	12.312	8.152	12.177	24.331
Prototype 1, 1 Group Member	6.181	12.710	12.314	8.157	8.252	24.359
Prototype 1, 2 Group Members	6.140	12.038	12.221	8.082	8.179	24.238
Prototype 1, 3 Group Members	6.128	11.939	12.207	8.067	8.138	24.209
Prototype 2, 1 Group Member	6.138	12.144	12.248	8.106	8.217	24.224
Prototype 2, 2 Group Members	6.091	11.926	12.156	8.023	8.134	24.037
Prototype 2, 3 Group Members	6.086	11.900	12.142	8.010	8.125	24.021

Time taken for one operation (msec)	1 Gbit					
	1 file			100 files		
	create	read	delete	create	read	delete
Standard Lustre	1.607	1.816	0.751	1.570	1.921	0.513
MDS Interceptor	160.984	41.967	80.776	121.855	81.383	40.841
Client Interceptor	161.394	45.007	80.765	122.038	84.173	41.018
Client Int. and MDS Int.	162.097	42.918	81.222	122.675	82.122	41.100
Prototype 1, 1 Group Member	161.786	78.680	81.211	122.598	121.184	41.052
Prototype 1, 2 Group Members	162.871	83.071	81.825	123.734	122.269	41.257
Prototype 1, 3 Group Members	163.193	83.762	81.919	123.964	122.882	41.308
Prototype 2, 1 Group Member	162.920	82.348	81.649	123.364	121.696	41.282
Prototype 2, 2 Group Members	164.165	83.850	82.263	124.646	122.937	41.602
Prototype 2, 3 Group Members	164.310	84.033	82.359	124.840	123.078	41.630

Figure 3.11.: Performance Test Results 1GBit

about 200ms. In the “Metadata Service for Highly Available Cluster Storage Systems” project the latency times for one client are about 15ms, however these times result from internal replication. The latency times from the JOSHUA project are gained with a similar test setup like in this master thesis. Hence the 200ms form the mark of the expected latency times.

The measured latency times in the test runs are in the range from 165ms - 40ms, depending on the operation performed and network type used. This seems okay, but the problem is the overhead caused to the file system. The measured overhead to the system in the JOSHUA project is 256% with four group members. The overhead of Prototype 2 with three group members using 100MBit network in comparison to the default Lustre configuration is about 8815%! Another possibility to compare this significant impact is to look at the request throughput achieved in the “Metadata Service for Highly Available Cluster Storage Systems” project, see Figure 1.6. There, the file system has a through-

Delay Time of IP Aliasing

100MBit Network	
Local Connection	29.483 μsec
IP Alias Connection	29.318 μsec

1GBit Network	
Local Connection	29.458 μsec
IP Alias Connection	29.350 μsec

Table 3.1.: Delay Time of IP Aliasing

put of about 125 read requests with one client using one metadata server. With the use of more metadata servers this throughput even increases due to the advantage of parallelism. In case of four metadata servers the gained throughput of read requests per second with one client is about 360.

Quite different the results of the prototypes of this master thesis. The default Lustre setup achieves a read request throughput of about 450 to 550 depending on the used network and the number of files to read in one test run. Of course, the advantage of parallelism cannot be taken into account, because all prototype setups still work with only one MDS. However, the measured values are by far under the expectations. For instance in case of the Prototype 2 test run with 3 group members and use of 1GBit network and 100 files the read throughput breaks down from 520 to 8 requests per second. Such a result renders the proposed HA solution unreasonable in terms of performance.

The performance results are contrary to the results of the preceding two HA projects. The experience from the preceding projects shows that HA solutions don't come for free, but the performance impact is reasonable and the advantage of higher availability outweighs this downside. This is not the case in this project. The latency times introduced by the prototypes are too high to use the Lustre file system in a reasonable way. This raises the question for the reasons of these high latency times.

To gain a better understanding of the measured values, tests to evaluate the pure network performance of the test cluster are useful. Also a check of the caused delay by the IP

100MBit Network Latency			
	Size	Latency	Bandwidth
Client-Server	10 B	200.05 us	49.99 KB/s
	100 B	149.93 us	666.98 KB/s
	1.00 KB	284.30 us	3.52 MB/s
	10.00 KB	1.90 ms	5.25 MB/s
	100.00 KB	22.28 ms	4.49 MB/s
	1.00 MB	218.34 ms	4.58 MB/s
	10.00 MB	2.29 s	4.38 MB/s
	Size	Latency	Bandwidth
Client-Interceptor-Server	10 B	343.57 us	29.11 KB/s
	100 B	150.62 us	663.92 KB/s
	1.00 KB	314.57 us	3.18 MB/s
	10.00 KB	1.94 ms	5.16 MB/s
	100.00 KB	21.93 ms	4.56 MB/s
	1.00 MB	219.71 ms	4.55 MB/s
	10.00 MB	2.30 s	4.35 MB/s
	Size	Latency	Bandwidth
Client-Interceptor-Interceptor-Server	10 B	352.65 us	28.36 KB/s
	100 B	178.42 us	560.48 KB/s
	1.00 KB	346.70 us	2.88 MB/s
	10.00 KB	1.99 ms	5.03 MB/s
	100.00 KB	22.72 ms	4.40 MB/s
	1.00 MB	226.96 ms	4.41 MB/s
	10.00 MB	2.32 s	4.31 MB/s

Table 3.2.: 100MBit Network Latency

aliasing is needed.

To measure the delay caused by the IP aliasing a simple test program can be written. The program starts a server on the original node address on a given port. This server just bounces back messages. Then the program establishes two connections to this server. One time from the same local address and one time from the IP alias address. Now, the program sends a sting to the server and measures the time it takes to receive the string again.

Table 3.1 shows the results of this test. The delay times for the both connections are almost the same. Also the network types make no difference. This was expected, because the communication happened only local without use of the network. As Table 3.1 shows, the use of IP aliasing causes no considerable delays and thus cannot be the source of the significant performance problems of the prototype.

1GBit Network Latency			
	Size	Latency	Bandwidth
	10 B	102.29 us	97.76 KB/s
	100 B	237.95 us	420.26 KB/s
Client-Server	1.00 KB	193.60 us	5.17 MB/s
	10.00 KB	332.54 us	30.07 MB/s
	100.00 KB	1.85 ms	53.93 MB/s
	1.00 MB	17.15 ms	58.30 MB/s
	10.00 MB	170.12 ms	58.78 MB/s
	Size	Latency	Bandwidth
	10 B	337.11 us	29.66 KB/s
	100 B	126.84 us	788.39 KB/s
Client-Interceptor-Server	1.00 KB	175.89 us	5.69 MB/s
	10.00 KB	384.31 us	26.02 MB/s
	100.00 KB	2.06 ms	48.48 MB/s
	1.00 MB	19.47 ms	51.36 MB/s
	10.00 MB	196.76 ms	50.82 MB/s
	Size	Latency	Bandwidth
	10 B	353.49 us	28.29 KB/s
	100 B	156.73 us	638.04 KB/s
Client-Interceptor-Interceptor-Server	1.00 KB	205.82 us	4.86 MB/s
	10.00 KB	420.77 us	23.77 MB/s
	100.00 KB	2.28 ms	43.77 MB/s
	1.00 MB	21.81 ms	45.85 MB/s
	10.00 MB	222.89 ms	44.86 MB/s

Table 3.3.: 1GBit Network Latency

The next step is to measure the delay times caused by the network. Therefore, the latency time of the different network paths must be measured. This is done with another test program. This program sends byte packages of increasing size over the given network path. It measures the latency time caused by the network and calculates the bandwidth of the connection.

Considering the size of metadata messages, the test runs show that the latency time of the network lies in the range of milliseconds. This is even the highest possible latency time. Average metadata messages of Lustre are not bigger than 1KB. For the Gigabit network test, this latency time even for the longest path was not much more than 200 μs. So the network is unlikely to be the reason causing the performance issues of the prototypes.

The IP aliasing and the network itself are not the reason for the high latency times.

Another possibility is the implementation of the prototypes itself.

The core component of the prototypes is the message routing. The proper functionality of this component is proven in Section 3.4.1. In terms of performance the problems discussed in Section 3.2 are essential. All of the different mentioned approaches have been tested. The parallel approach is a bit faster than the serial used in the performance tests. However, the gained performance plus is so little, that it makes no real difference in the measured values of the performance tests. As a result, the prototype implementations show no errors responsible for causing the significant performance impact.

The last possibility of the performance problems is the Lustre code itself. The file system cache has been deactivated in order to get consistent results. But due to the complex and intransparent design, it is likely that Lustre uses internally techniques that are blocked by the interceptors and thus cause the performance impact. However this is speculation and cannot be proven.

In spite of the performance problems it is worth to take a closer look at the measured values.

The general trend of the measured values is alright. The test runs performed on 1Gbit network give lower latency times/more operations per second than the test runs performed on 100MBit network. The read operation performs better if called only one time, like in case of the 1 file test runs. Quite the contrary the create and delete operations. They achieve better results if called several times like in the 100 files test runs. The delete operation achieves twice the throughput in the 100 files test runs than in the 1 file test runs. This can be the result of internal caching in the MDS of Lustre. The MDS, for instance, caches several requests in memory before it commits them to disk. This behaviour cannot be avoided.

However, there are some inconsistencies in the values. For instance, the values of Prototype 1, using 100MBit network, 1 file. Here the prototype achieves lower latency times with three group members than with one. At first glace, this seem odd. But this could happen with the "Fast Delivery Protocol" introduced in Section 1.2.2. The reason is that every member in the group can acknowledge a message. In the test setups only one group member actually runs a MDS the other group members only run an interceptor with Transis. These nodes are less occupied than the one node running the MDS. They

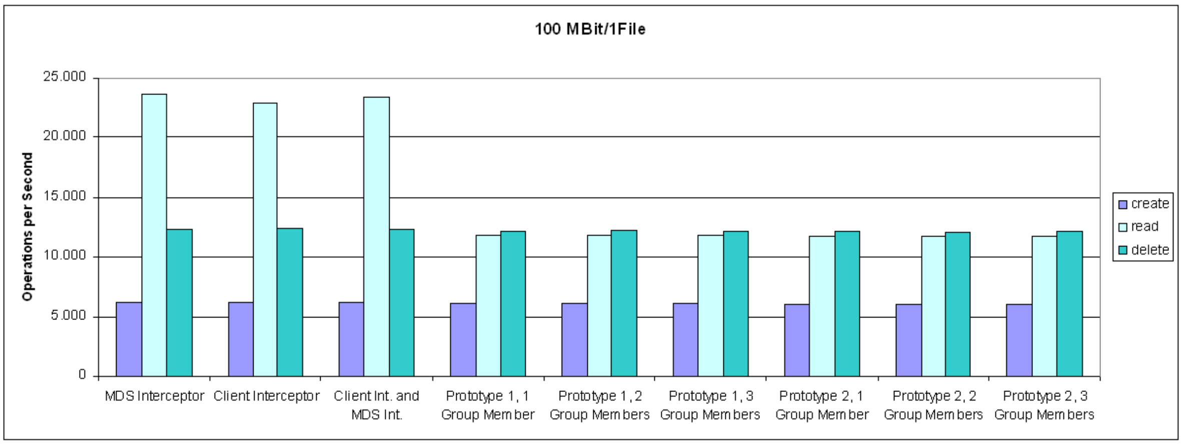

Figure 3.12.: 100MBit, 1File Test Runs

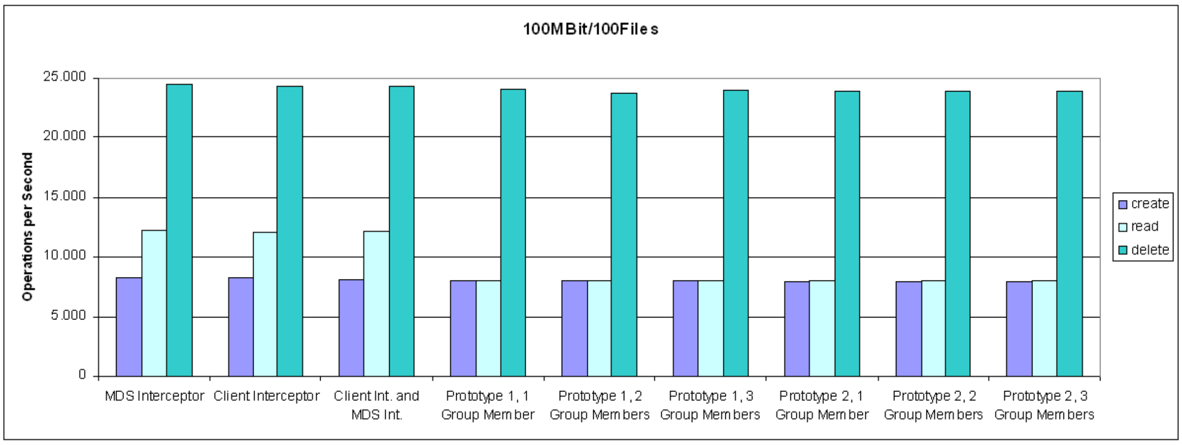

Figure 3.13.: 100MBit, 100Files Test Runs

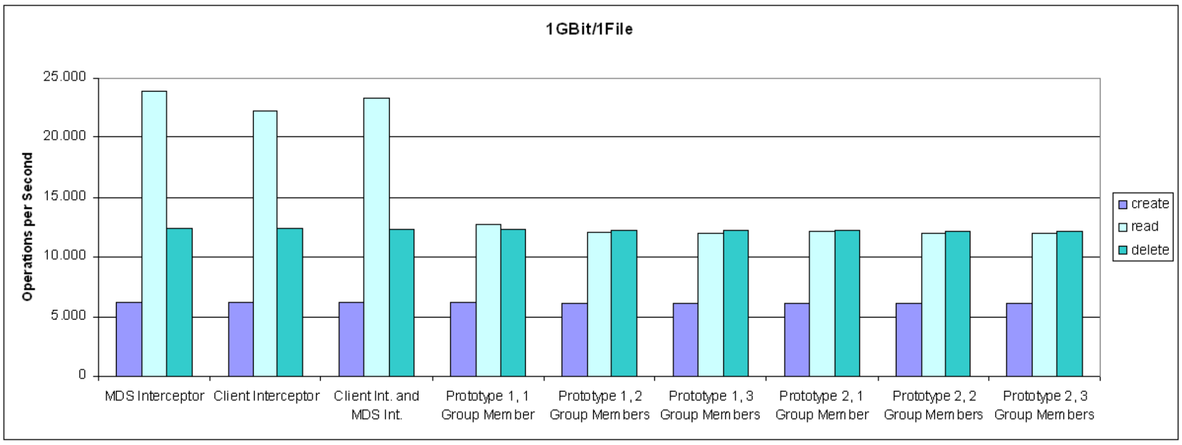

Figure 3.14.: 1GBit, 1File Test Runs

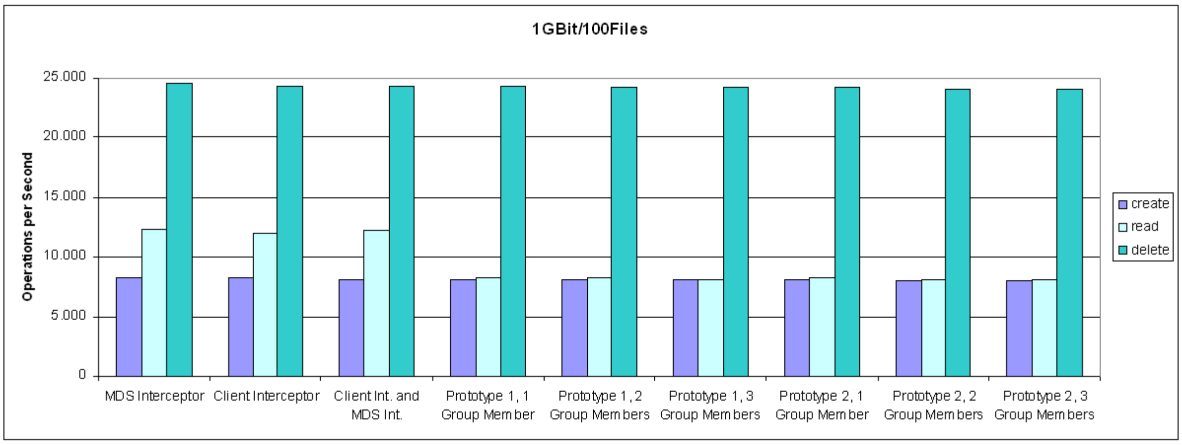

Figure 3.15.: 1GBit, 100Files Test Runs

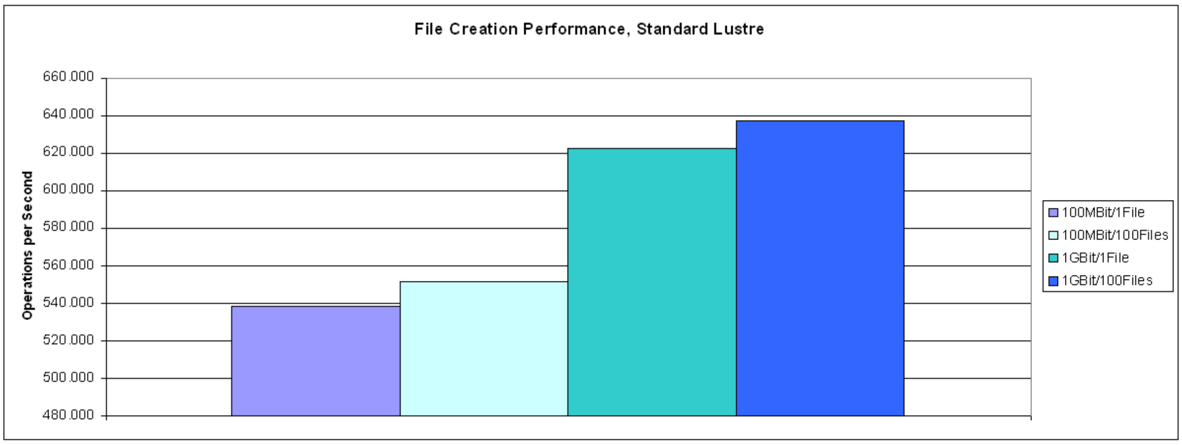

Figure 3.16.: File Creation Performance of Lustre

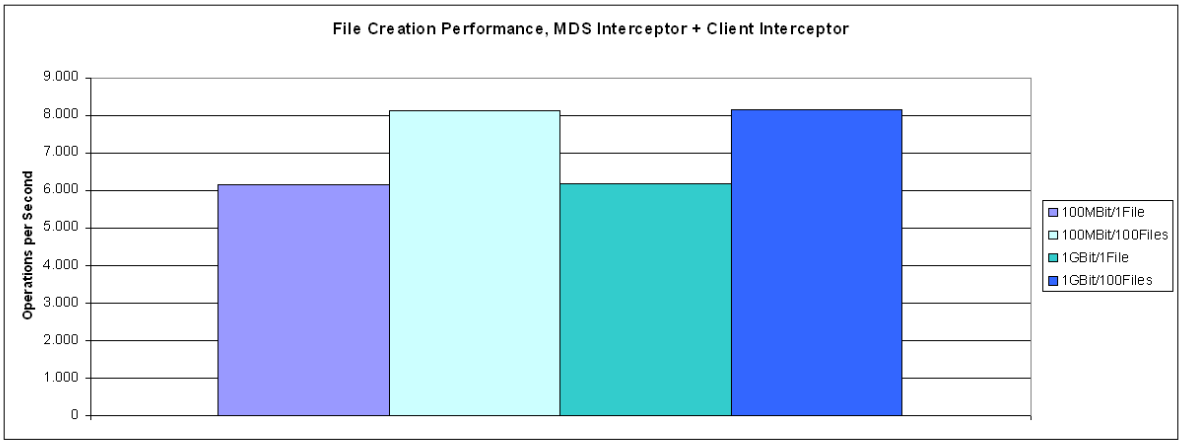

Figure 3.17.: File Creation Performance using MDS Interceptor and Client Interceptor

just wait for incoming messages without any processing. It is likely that one of these nodes can acknowledge a message faster than the one node running the MDS. This could be the reason for the lower latency times with three group members than with one group member.

Another inconsistency can be seen in the measured values of the interceptor latency times. In the 1GBit, 1 file, read command test run the measured performance of the test setup with the client interceptor alone is 22.219 operations per second. However, the measured performance of the test setup with client and MDS interceptor is 23.300 operations per second. This is not reasonable and should not happen. Source of this error in the measurements might be changing occupation of the nodes due to other running processes in the background or different workload on the network during the individual test runs.

The Figures 3.16 and 3.17 show a different behaviour of the default Lustre setup in contrast to the file system with included interceptors. As shown in the figures, the advantage of the faster Gigabit network is much bigger in the default Lustre setup. This result also indicates some problems with the correct adaptation of the interceptors to the file system.

To summarise, the measured values show some light inconsistencies, but nevertheless appear to be okay. The major result of the test runs is the big performance impact of the prototype designs on the file system. This impact renders the proposed HA solution unreasonable in terms of performance. The source of the significant latency times is most likely to find in the file system code itself. To fully understand the reason of the performance impact, Lustre needs to be analysed and understood completely. This is not possible in the limited time of this master thesis and therefore the reason of the performance impact remains a speculation.

4 Detailed Software Design

4.1 Message Routing

Core component of the prototype design is the message routing. This component is responsible for managing the connections and routing the messages to the appropriate nodes.

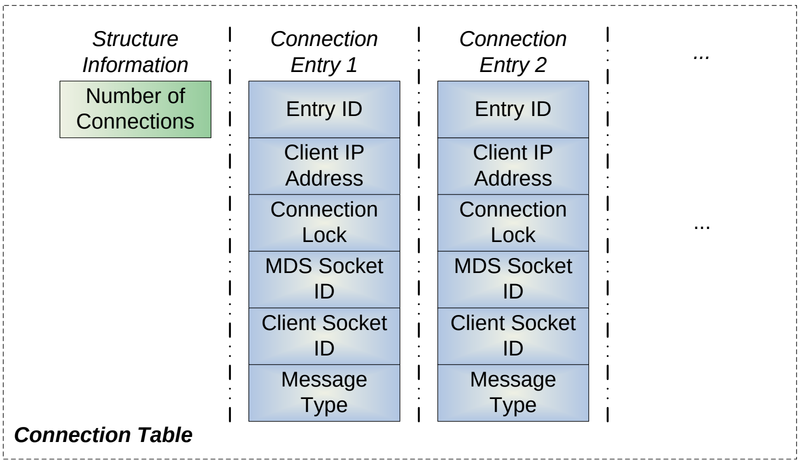

Figure 4.1.: Connection Table

Figure 4.1 shows the connection table structure. This structure is responsible for holding and maintaining all connection information. Because the connection table is a shared resource it needs to be locked. Mutual exclusion locks are used for this purpose. They avoid simultaneous access from the Transis receive thread and the interceptor receive thread. This is most important, because each thread can manipulate the allocated memory of the connection table. As a result, simultaneous access could lead to segmentation fault and crash of the program.

Each interceptor holds an own connection table. In order to keep the information consistent between all connection tables the group communication system is used.

The initiation of a connection is always the same process. Fist, each interceptor listens for incoming connections from the clients. If one interceptor gets an incoming connection it creates an entry in its connection table. In this step it stores the socket identifier of the client connection in this entry. The interceptor also sets the connection lock of this entry. This should prevent further message routing until the connection is fully established. Then, the interceptor uses the group communication system to send the id of the entry and the request to connect to the MDS. All interceptors, the sending one included, receive this request. All create the connection to their respective MDS. The socket identifier of this connection needs then to be stored in the table entry associated with the id sent in the request. Also, the connection lock of this entry must be unset after successful connection to the MDS. The interceptor connected to the client already holds an entry with this id in the connection table, and just adds the socket identifier of the MDS connection to this entry. It also unsets the connection lock. All other interceptors create a new entry with this id and add the socket identifier of their MDS connection. The connection lock is already unset in the new created entries.

The other information stored in the connection table is the IP address of the client. This information is not needed in the actual prototype implementations, but could be used to identify the client in case of connection failover. The use of the field `Message Type` is described later in this section.

If one client disconnects, the procedure to perform is similar to the connection process. Fist, the interceptor connected to this client sends a request to disconnect to the group communication system. After the connections are closed the appropriate table entries are deleted.

Figure 4.2 shows the connection state of a setup with three group members and one client. The client uses three connections for communication. Each connection is associated with one table entry. The only information needed to route each individual message are the id of the related connection table entry and the destination of the message (`CLIENT` or

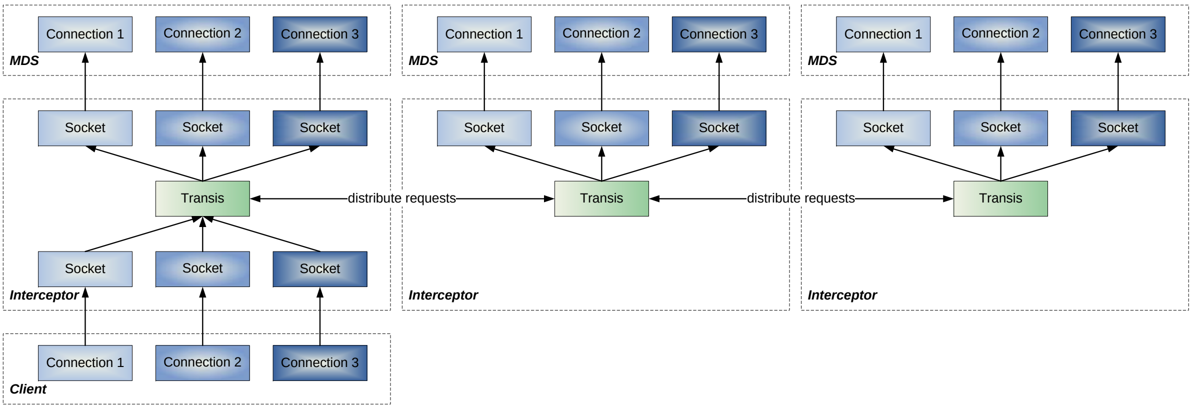

Figure 4.2.: Message Routing, Request from Client to MDS

MDS) to determine the direction.

In case of a message or request from the client to the MDS, the interceptor connected to the client receives the message. It then adds the needed routing information to the message and passes the message on to Transis. The group communication system distributes the message to all interceptors. They receive the message and read the routing information. The destination MDS tells them to choose a MDS socket and the entry id determines what connection to use. With help of this information the interceptors can pass on the message to the appropriate MDS connections.

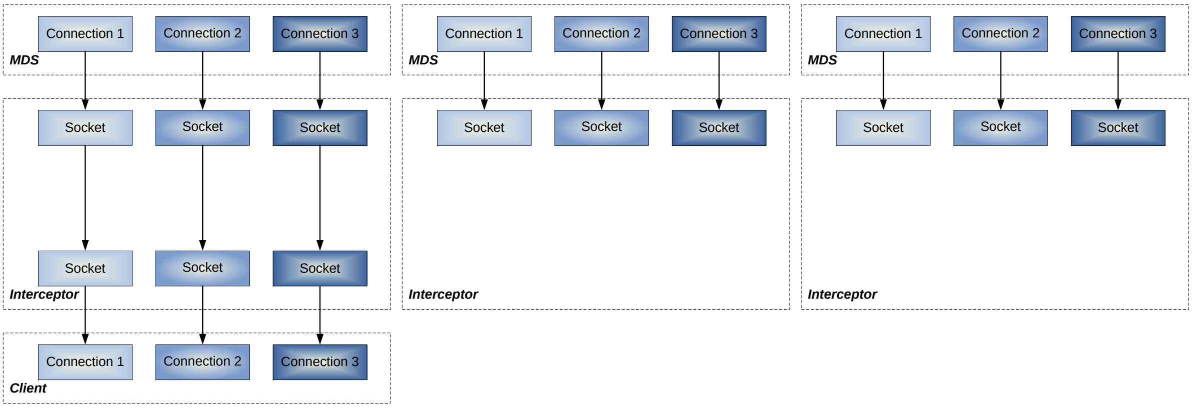

Figure 4.3.: Message Routing, Response from MDS to Client

In case of a response from the MDS to the client, all interceptors receive the response from their MDS, see Figure 4.3. Only the interceptor connected to the client holds information about the client socket in the respective connection table entry. Thus, only this interceptor passes the message on the client.

To meet the rules of Lustre's networking, messages need to be modified. Each interceptor needs to adjust the message header, in a way, that it acts as client for the MDS and vice versa. The important fields to change are the message Source NID and the Target/Destination NID, as described in Section 2.1.2. To avoid rejected messages from Lustre the interceptor has to change the IP address in the Source NID to its own IP address. Furthermore, it has to change the IP address in the Target/Destination NID to the IP address of the client and the MDS respectively.

Because the positions of the NID fields vary in the three different Lustre message types, the last field in a connection table entry is used. The field `Message Type` is set accordingly to the Lustre protocol. That way, it is ensured that throughout the connection initialisation the appropriate header type of the received message is known and the right values are changed. After the initialisation process this field is no longer used, due to the facts that only "Lustre Messages" are exchanges anymore.

4.2 Single Instance Execution Problem

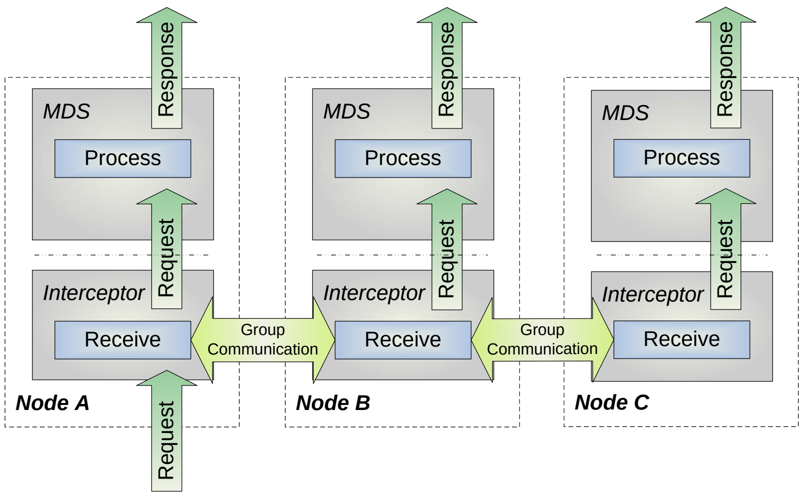

Figure 4.4.: Single Instance Execution Problem

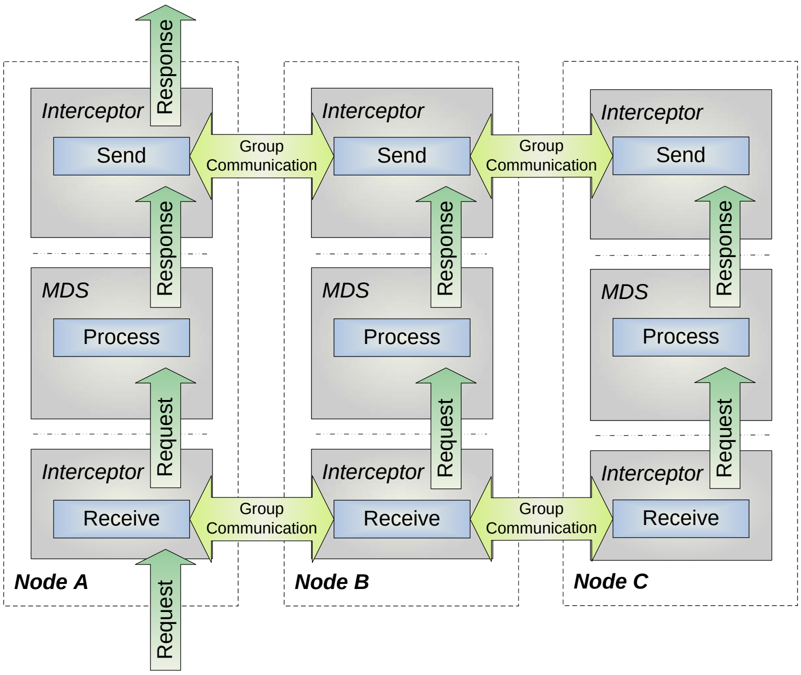

Figure 4.5.: Single Instance Execution Problem Solved

In an active/active architecture the replicated components work independent from each other. The group communication system distributes the incoming requests in the right order to the group and holds thus the group members in virtual synchrony. The problem here is that each member produces a response and wants to send this response to the system. The system however expects only one response to one request. Multiple responses are dropped in the best case or lead to inconsistencies, or crash in the worst case.

To sort out this problem, the group communication system has to be used again. As indicated in Figure 4.4, it has to be set between the output of the MDS and the rest of Lustre. In this position, it has the task of filtering all requests and sending only one back to Lustre.

In the prototype implementation, this problem is solved with help of the connection table described in Section 4.1. This table holds identifiers of existing client connections. When a response is received the group members look in the connection table for an appropriate client connection. Only the group member actually connected to the client sends the response to the system. The other members drop their request. Because one client is connected to one group member only, the response is sent only once to the system. Thus, the connection table can be used to filter the responses.

Another possibility to sort out this problem is to send the responses through the group communication system first. The group communication system distributes the responses to all group members. This raises the problem that the group member connected to the client gets the responses from the other group members as well. In this situation an identifier to recognize all equal responses from the group members is needed.

The approach to send all responses through the group communication system has an advantage. It could be used to detect errors in the response. This may be achieved with help of voting algorithms. Possibilities are for instance majority or unanimous voting algorithms. Fist, all responses from the group members need to be compared. In case of a majority voting algorithm all equal responses are counted. One response from the group with the highest number of equal responses is sent back to the system. All other responses are dropped. In case of a unanimous voting algorithm all responses have to be the same. If only one response differs from the others, not response at all is sent back to the system.

4.3 Dynamic Group Reconfiguration

Dynamic group reconfiguration is essential for running a group of members in an active/active fashion. Normally the system is started with one group member. In case of Lustre the file system is started, like intended, with one MDS. In order to build up the active/active group new members (MDS) must join.

The sense of HA is to provide uninterrupted service. To realize this goal the active/active group must be able to be reconfigured at runtime. If members fail they must be repaired or replaced with new ones. This functionality provides dynamic group reconfiguration.

The group communication system Transis keeps track of active group members. If the configuration of the group changes it sends a message with the new configuration. This message can be used to initiate the appropriate reconfiguration procedure.

The process of leaving members is simple. Because all members share the same state they can continue operation without new reconfiguration. The only thing to do, is to update the group member list of the client interceptors to avoid failover to broken group members that no longer share the global state.

To keep the state of the active/active group during the join process consistent the following steps must be performed in the right order:

1. stop all members from accepting requests
2. copy the group state from one elected member to the new member
3. start accepting requests again

Fist, all members must stop to accept new requests from the clients. Now an elected member has to send his state to the new member. This can be done with copying the partition in which the MDS data is stored to the new member. Now the entire group is in virtual synchrony again and can start to accept requests.

If something goes wrong during the join process, the new member shuts down itself to ensure that no member is online which does not share the exact same global state in order to sustain the virtual synchrony.

The design of Lustre raises some issues that avoid successful implementation of this capability in the prototype.

One problem could occur with server timeouts. During the whole join process the MDS is stopped, or better, occurs dead to the client. However this seems likely to be no problem, because the Lustre MDS is designed for heavy load. Lustre already has a similar problem when tens of thousands clients send requests to this one server at the same time. In this case the server is under such heavy load that it appears dead to some clients for minutes. To overcome this problem Lustre has already set the server timeout to 100 seconds, and in some cases, like in the Lawrence Livermore National Laboratory to 300 seconds.

Another problem to face is the reinitiation of connections to new MDS. Because Transis is implemented externally and Lustre uses three active connections for one client, it's not enough to copy the state (partition) to the new MDS. The new group member (interceptor) needs to connect the active clients to the new started MDS. Therefore the state of connections must also be copied. To establish a connection the interceptor has to follow the Lustre protocol. One possibility to solve this problem is to save the original initiation messages of each connection and reuse them for new members.

Lustre's MDS also works with caching of requests. This is another source of inconsistency. Because it is never ensured that the state on the disk (the partition) is the same like the state in the RAM (the running MDS).

The main challenge is to start the new MDS. This point rendered the dynamic group reconfiguration impossible within the limits of this project. The Lustre design doesn't allow two active MDS at the same time. For failover Lustre first shuts down the failed MDS and starts then the new MDS. As long as one MDS is up, it is impossible to start a second MDS. Even if this hurdle could be sorted out, the Lustre design still causes plenty of problems. For example distributed locking and the fact that the MDS talks with the OSTs. For one request, each MDS in the group would try to get the same lock from the OSTs or try to create the same file.

4.4 Connection Failover

Connection failover is an integral part in the HA solution. It ensures the masking of errors to the connected clients. If a client is connected to a MDS and this MDS fails, the client gets an error and cannot use the service anymore. The state is still saved as long as another MDS is up. However, in an active/active HA solution uninterrupted service should be provided.

Solution to the problem is connection failover. It is the ability of the client to change to another active MDS.

To realize this solution, the client needs to hold a list of all available MDS. If the connection to the MDS fails, the client looks in the list and connects to another MDS. That way the error of a failing connected MDS is also masked from the client.

One problem with inconsistency could occur, when a request is already in the queue of

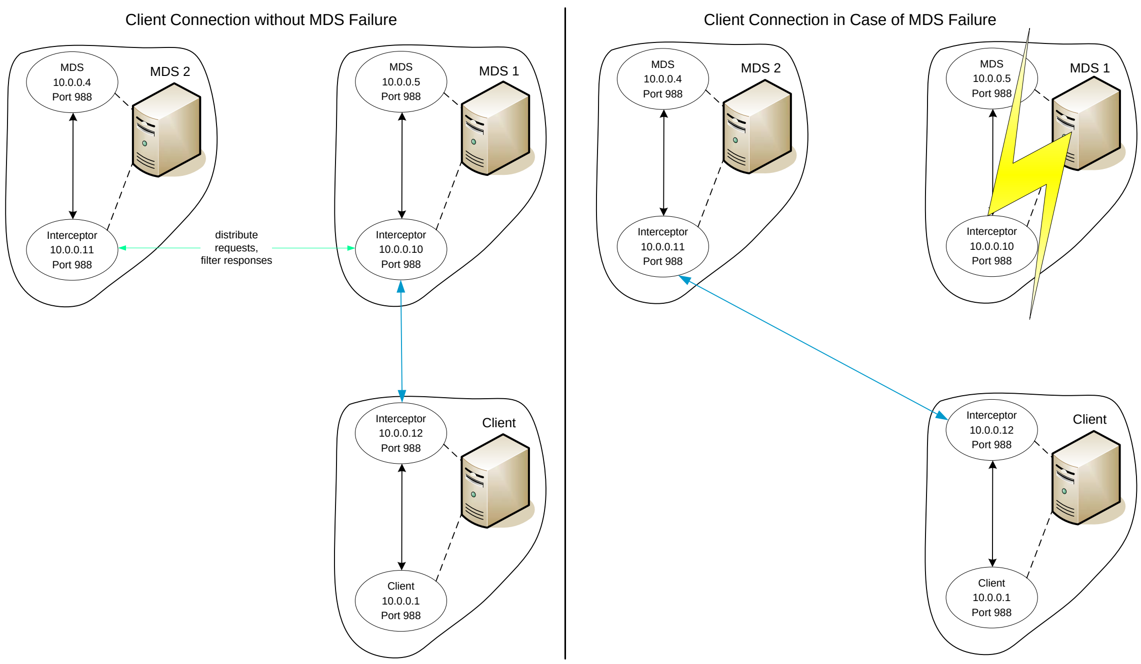

Figure 4.6.: Connection Failover

the connected MDS but is not distributed yet before the MDS fails. To avoid such errors an acknowledgment scheme is needed.

5 Conclusions

5.1 Results

This Master thesis project aims to improve the availability of the Lustre file system. Major concern of this project is the metadata server (MDS) of the file system.

The MDS of Lustre suffers from the last single point of failure in the file system. Lustre already provides an active/standby high availability (HA) solution for the MDS. Downside of this solution is the shared disk between the two MDS to store the metadata. If this disk fails, the state of the entire file system is lost.

To overcome this single point of failure a new active/active HA approach is introduced. In the active/active mode the MDS is replicated on several nodes, each using its own disk to share the metadata.

To achieve a shared global state among the multiple MDS nodes an existing group communication framework is used.

The new file system design with multiple MDS nodes running in virtual synchrony provides active/active high availability and leads to a significant increase of availability.

Goal of the project is to develop a proof-of-concept implementation based on the experience attained in preceding two active/active HA projects[1,2] at the Oak Ridge National Laboratory.

[1] The JOSHUA Project [21]
[2] Symmetric Active/Active Metadata Service [18]

As a final result achieved of this Master thesis project, all general system design tasks have been finished. As shown in the previous sections an overall system design to solve the key problems of the dissertation has been created.

For proper development and testing a working environment has been build and set up. The development was done on a small dedicated cluster with one to three nodes serving as MDS, one node serving as object storage target (OST), and one node serving as client for the file system. All nodes are homogeneous and identical in hardware and software setup. The system tests have been done on 100MBit and 1GBit network.

Two prototype implementations have been developed with the aim, to show how the proposed system design and its new realized form of symmetric active/active high availability can be accomplished in practice.

The Lustre networking has been analysed in order to include the HA system components into the file system.

The functionality tests of the prototypes prove working components like interceptors or the group communication system. However, they also show missing functionality of the prototypes. Components like dynamic group reconfiguration or connection failover couldn't be implemented. With lack of this functionality no working active/active HA solution can be provided with this Master thesis. Reason for the missing components is the Lustre design. It doesn't allow multiple running MDS at the same time. Furthermore, the MDS is so tightly included into the file system, that there is no reasonable workaround to this problem.

The performance tests show a significant performance impact of the prototypes on the file system. This impact renders the proposed HA solution unreasonable in terms of performance. After several tests, the problem causing this impact seems to be in the Lustre implementation. However, this is mere speculation and cannot be proven.

The results of this dissertation show the difficulties of an implementation of an active/active HA solution for MDS of Lustre. The insufficient documentation and the complicated and intransparent design of Lustre prohibit an adaptation to this solution. An easy adaptation of the file system to the active/active HA design like in the case of the parallel virtual file system (PVFS) in one of the preceding projects[3] is not possible with Lustre.

Nevertheless, the results and findings of this Master thesis may be used for further improvement of high availability for distributed file systems.

5.2 Future Work

The results and findings of this Master thesis cannot provide a working solution to the last single point of failure in Lustre.

The work provides a complete system design that needs to be adapted to Lustre. This adaptation requires further investigation of the file system.

In order to implement a fully working production type active/active HA solution, the inner workings of the Lustre components must be understood and adjusted. The need to run multiple MDS at the same time requires a change of the entire Lustre design.

To overcome the performance problems of the prototypes of this project, the source of the significant performance impact needs to be found.

Another problem is the group communication system Transis. Its inability to run in a multithreaded environment limits the possibilities of the prototype design. Transis needs to be replaced by a more sophisticated group communication system.

Due to the requirement of performing changes in the Lustre code anyway and the performance issues of the project prototype implementations, the internal replication method seems to be preferred for further work on active/active HA for Lustre.

[3] Symmetric Active/Active Metadata Service [18]

References

[1] Software Testing explained at Wikipedia. Available at http://en.wikipedia.org/wiki/Software_test.

[2] The Parallel Virtual File System (PVFS) Project. Available at http://www.pvfs.org/index.html.

[3] Transis group communication system project at Hebrew University of Jerusalem, Israel. Available at http://www.cs.huji.ac.il/labs/transis.

[4] Universally Unique Identifier (UUID) explained at Wikipedia. Available at http://en.wikipedia.org/wiki/Uuid.

[5] R. Alexander, C. Kerner, J. Kuehn, J. Layton, P. Luca, H. Ong, S. Oral, L. Stein, J. Schroeder, S. Woods, and S. Studham. *Lustre^{TM}: A How To Guide for Installing and Configuring Lustre Version 1.4.1*, 2005. Available at www.ncsa.uiuc.edu/News/datalink/0507/LustreHowTo.pdf.

[6] S. Bafna, S. Dalvi, A. Kampasi, and A. Kulkarni. CHIRAYU: A Highly Available Metadata Server for Object Based Storage Cluster File System. In *IEEE Bombay Section*, Apr. 2003. Available at www.cs.utexas.edu/~abhinay/research_papers/chirayu.pdf.

[7] S. Bafna, S. Dalvi, A. Kampasi, and A. Kulkarni. Increasing current Lustre availability to 99.9% with a backup Metadata Server. Jan. 2003. Available at http://abhinaykampasi.tripod.com/TechDocs/HAMDSCharacteristics.pdf.

[8] Cluster File Systems, Inc. *Lustre White Paper*, 2004. Available at http://www.lustre.org.

[9] Cluster File Systems, Inc. *Lustre 1.4.7 Operations Manual, Version 1.4.7.1-man-v35 (09/14/2006)*, 2006. Available at http://www.lustre.org.

[10] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[11] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He. Symmetric Active/Active High Availability for High-Performance Computing System Services. *Journal of Computers*, 1(8):43–54, 2006.

[12] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He. Towards High Availability for High-Performance Computing System Services: Accomplishments and Limitations. In *Proceedings of High Availability and Performance Workshop*, Santa Fe, NM, USA, Oct. 17, 2006. Available at `www.csm.ornl.gov/~engelman/publications/engelmann06towards.pdf`.

[13] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He. On Programming Models for Service-Level High Availability. In *Proceedings of* 2^{nd} *International Conference on Availability, Reliability and Security*, Vienna, Austria, Apr. 10-13, 2007.

[14] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He. Transparent Symmetric Active/Active Replication for Service-Level High Availability. In *Proceedings of* 7^{th} *IEEE International Symposium on Cluster Computing and the Grid*, Rio de Janeiro, Brazil, May 14-17, 2007. To appear.

[15] X. He, L. Ou, C. Engelmann, X. Chen, and S. L. Scott. Symmetric Active/Active Metadata Service for High Availability Parallel File Systems. 2007. (under review).

[16] C. Leangsuksun, V. K. Munganuru, T. Liu, S. L. Scott, and C. Engelmann. Asymmetric Active-Active High Availability for High-end Computing. In *Proceedings of* 2^{nd} *International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters*, Cambridge, MA, USA, June 19, 2005.

[17] D. Malki. *The Transis User Tutorial*, 2004. Available at `http://www.cs.huji.ac.il/labs/transis`.

[18] L. Ou, C. Engelmann, X. He, X. Chen, and S. L. Scott. Symmetric Active/Active Metadata Service for Highly Available Cluster Storage Systems. 2007. (under review).

[19] L. Ou, X. He, C. Engelmann, and S. L. Scott. A Fast Delivery Protocol for Total Order Broadcasting. 2007. (under review).

[20] K. Uhlemann. High Availability for High-End Scientific Computing. Master's thesis, Department of Computer Science, University of Reading, UK, Mar. 2006.

[21] K. Uhlemann, C. Engelmann, and S. L. Scott. JOSHUA: Symmetric Active/Active Replication for Highly Available HPC Job and Resource Management. In *Proceedings of IEEE International Conference on Cluster Computing*, Barcelona, Spain, Sept. 25-28, 2006.

A Appendix

A.1 Lustre HA Daemon Source Code

A.1.1 lustreHAdaemon.c

```c
// ---------------------------------------------------------------------------
// Lustre High Availability Daemon
//
// lustreHAdaemon.c  --source file--
//
// version 0.52rev
//
// by Matthias Weber
// ---------------------------------------------------------------------------


#include "transis.h"
#include "lustreHAdaemon.h"
#include "lustreMessageAdjust.h"


// Globals
__u8                fileCounterR = 0;     /* counter for debug files Receive */
int                 interceptorSocketID;  /* the id of the interceptor server socket */
struct hostent      *hostinfo;            /* hold host information */
connection_table_t  *connectionTable;     /* table of available connections */
int   LusterAcceptorPort = LUSTRE_MAX_ACC_PORT;        /* local secure port for MDS */
pthread_mutex_t mutexCT = PTHREAD_MUTEX_INITIALIZER;   /* connection table lock */


// ---------------------------------------------------------------------------
// Get information about host running on
//
// returns: 0 on success / -1 if error occurs
// ---------------------------------------------------------------------------
int GetHostInfo()
{
  char hostname[HOSTNAME_LENGTH];

  /* get host information */
  if (gethostname(hostname, HOSTNAME_LENGTH) != 0) {
    perror("error getting hostname");
    return -1;
```

```c
  }
  if ((hostinfo = gethostbyname(hostname)) == NULL) {
    herror("error getting host by name");
    return -1;
  }

  printf("Official host name: [%s]\n", hostinfo->h_name);
  printf("Official host addr: [%s]\n", inet_ntoa(
                                        *(struct in_addr *)hostinfo->h_addr_list[0]));

  return 0;
}


// --------------------------------------------------------------------------
// starts the MDS/Client interceptor server
//
// returns: 0 on success / -1 if error occurs
// --------------------------------------------------------------------------
int StartInterceptorServer()
{
  int rc;
  struct sockaddr_in socketServer;

  /* setting server up */
  interceptorSocketID = socket(AF_INET, SOCK_STREAM, 0);
  if (interceptorSocketID < 0) {
    perror("error opening interceptor socket");
    return -1;
  }

  socketServer.sin_family        = AF_INET;
  socketServer.sin_addr.s_addr = inet_addr(INTERCEPTOR_ADDR);
  socketServer.sin_port          = htons(LUSTRE_SERVER_PORT);
  bzero (socketServer.sin_zero, 8);

  printf("Binding Interceptor port: [%i] on addr: [%s]\n", LUSTRE_SERVER_PORT,
                                                           INTERCEPTOR_ADDR);

  rc = bind(interceptorSocketID, (struct sockaddr *)&socketServer,
                                  sizeof(socketServer));
  if(rc < 0){
    perror("error binding interceptor socket");
    return -1;
  }

  rc = listen(interceptorSocketID, NUM_CONNECTIONS);
  if(rc < 0){
    perror("error listening to interceptor socket");
    return -1;
  }

  return 0;
}


// --------------------------------------------------------------------------
// Main Loop;
//
// checks Sockets for messages and processes them,
// looks for incomming connections as well
//
// returns: 0 on success / -1 if error occurs
```

```
// ----------------------------------------------------------------------------
int MessagePassOn ()
{
  int     rc;
  int     i;
  int     ls;
  fd_set  readfds;
  int     maxfd;                                     /* maximum file desciptor used */
  int     noe;                                       /* number of connection entries */
  int     MDSSockets[NUM_CONNECTIONS];               /* MDS sockets */
  int     CLIENTSockets[NUM_CONNECTIONS];            /* CLIENT sockets */
  int     IDOfIndex[NUM_CONNECTIONS];                /* IDs of connection entries */
  int     MessageType[NUM_CONNECTIONS];              /* message types of connection entries */
  int     closedConnections[NUM_CONNECTIONS];        /* closed connection entries */
  int     numberOfClsConn;                           /* number of closed connection entries */


  /* Lustre pass through */
  while(1){

    numberOfClsConn   = 0;

    /* get connection table lock */
    rc = pthread_mutex_lock(&mutexCT);   /* get lock */
    if(rc != 0) {
      perror("error getting connection table lock");
      return -1;
    }

    /* check for active connections */
    noe = GetNumberOfEnties();
    FD_ZERO(&readfds);
    FD_SET(interceptorSocketID, &readfds);   /* look for incomming connections */
    maxfd = interceptorSocketID;             /* set max fd */

    /* set the active connections */
    for (i=0; i<noe; i++) {
      /* set MDS */
      MDSSockets[i] = connectionTable->connection[i].MDSSocket;
      if(MDSSockets[i] != -1){
        FD_SET(MDSSockets[i], &readfds);
        if(MDSSockets[i] > maxfd)
          maxfd = MDSSockets[i];
      }
      /* set Client */
      CLIENTSockets[i] = connectionTable->connection[i].ClientSocket;
      if(CLIENTSockets[i] != -1){
        FD_SET(CLIENTSockets[i], &readfds);
        if(CLIENTSockets[i] > maxfd)
          maxfd = CLIENTSockets[i];
      }

      /* get connection id */
      IDOfIndex[i] = connectionTable->connection[i].id;

      /* get message type */
      MessageType[i] = connectionTable->connection[i].MessageType;
    }//for

    /* release connection table lock */
    rc = pthread_mutex_unlock(&mutexCT);   /* release lock */
    if(rc != 0) {
      perror("error releasing connection table lock");
```

```
      return -1;
    }

    /* wait for data on sockets */
    rc = select(maxfd+1, &readfds, NULL, NULL, NULL);
    if(rc == -1) {
      perror("error select");
      return -1;
    }

    /* process connections */
    for(i=0; i<noe; i++) {
      int closed = 0;

      /* check Client */
      if(CLIENTSockets[i] != -1){
        if(FD_ISSET(CLIENTSockets[i], &readfds)){

          /* process message */
          switch (MessageType[i]) {
            case LUSTRE_ACCEPTOR_CONNREQ:
              rc = ReceiveAcceptorRequest(IDOfIndex[i], CLIENTSockets[i], MDS);
              if(rc == -1)
                return -1;
              if(rc == -2){
                closedConnections[numberOfClsConn++] = IDOfIndex[i];
                closed = 1;
              }
              break;
            case LUSTRE_LNET_HELLO:
              rc = ReceiveLNETHello(IDOfIndex[i], CLIENTSockets[i], MDS);
              if(rc == -1)
                return -1;
              if(rc == -2){
                closedConnections[numberOfClsConn++] = IDOfIndex[i];
                closed = 1;
              }
              break;
            case LUSTRE_MESSAGE:
              rc = ReceiveLustreMessage(IDOfIndex[i], CLIENTSockets[i], MDS);
              if(rc == -1)
                return -1;
              if(rc == -2){
                closedConnections[numberOfClsConn++] = IDOfIndex[i];
                closed = 1;
              }
              break;
            default:
              fprintf(stderr, "error, got wrong message type\n");
              return -1;
              break;
          }//switch
        }//if
      }//if

      /* check if connection was closed */
      if(closed == 1)
        continue;

      /* check MDS */
      if(MDSSockets[i] != -1){
        if(FD_ISSET(MDSSockets[i], &readfds)){

```

```
            /* process message */
            switch (MessageType[i]) {
              case LUSTRE_ACCEPTOR_CONNREQ:
                rc = ReceiveAcceptorRequest(IDOfIndex[i], MDSSockets[i], CLIENT);
                if(rc == -1)
                  return -1;
                if(rc == -2){
                  closedConnections[numberOfClsConn++] = IDOfIndex[i];
                  closed = 1;
                }
                break;
              case LUSTRE_LNET_HELLO:
                rc = ReceiveLNETHello(IDOfIndex[i], MDSSockets[i], CLIENT);
                if(rc == -1)
                  return -1;
                if(rc == -2){
                  closedConnections[numberOfClsConn++] = IDOfIndex[i];
                  closed = 1;
                }
                break;
              case LUSTRE_MESSAGE:
                rc = ReceiveLustreMessage(IDOfIndex[i], MDSSockets[i], CLIENT);
                if(rc == -1)
                  return -1;
                if(rc == -2){
                  closedConnections[numberOfClsConn++] = IDOfIndex[i];
                  closed = 1;
                }
                break;
              default:
                fprintf(stderr, "error, got wrong message type\n");
                return -1;
                break;
            }//switch
          }//if
        }//if
      }//for

      /* close connections */
      for(i=0; i<numberOfClsConn; i++){
        /* get connection table lock */
        ls = pthread_mutex_lock(&mutexCT); /* get lock */
        if(ls != 0){
          perror("error getting connection table lock");
          return -1;
        }
        rc = CloseConnection(closedConnections[i]);
        /* release connection table lock */
        ls = pthread_mutex_unlock(&mutexCT); /* release lock */
        if(ls != 0){
          perror("error releasing connection table lock");
          return -1;
        }
        if(rc == -1)
          return -1;
      }

      /* handle new Client connection */
      if(FD_ISSET(interceptorSocketID, &readfds)) {
        rc = GetNewClient();
        if(rc == -1)
          return -1;
      }
```

```

  }//while

  return 0;
}


// -----------------------------------------------------------------------
// routine to close one connection between Client and MDS
//
// sockets are closed and the connection table enty is removed
//
// id - id of table entry with the connection details
//
// returns: 0 on success / -1 if error occurs
// -----------------------------------------------------------------------
int CloseConnection (int id)
{
  int  rc;
  int  socket;

  /* close MDS socket */
  rc = GetSocketFromConnectionTable (id, MDS, &socket);
  switch (rc) {
    case 0:
      close(socket);
      break;
    case -1:
      fprintf(stderr, "error getting socket from MDS connection\n");
      return -1;
      break;
    case -2:
      break;
  }

  /* close Client socket */
  rc = GetSocketFromConnectionTable (id, CLIENT, &socket);
  switch (rc) {
    case 0:
      close(socket);
      break;
    case -1:
      fprintf(stderr, "error getting socket from Client connection\n");
      return -1;
      break;
    case -2:
      break;
  }

  /* Remove connection entry from table */
  rc = RemoveEntryFromConnectionTable(id);
  if(rc == -1)
    return -1;

  printf("Connection with id: %i disconnected!\n", id);
  return 0;
}


// -----------------------------------------------------------------------
// set up incomming client connection
//
// if connection comes in, Client is accepted, connection table is
```

```
// set up and request to connect to MDS is sent to Transis,
// function waits for lock and returns after connection
// is established
//
// returns: 0 on success / -1 if error occurs
// ---------------------------------------------------------------------------
int GetNewClient ()
{
  int rc;
  int ls;
  int id;
  int socket;
#ifndef TRANSIS_BYPASS
  __u32        *header;
#endif
  struct       sockaddr_in socketClient;
  unsigned int lengthClient = sizeof(socketClient);

  printf("Getting new client...\n");

  /* get Client */
  socket = accept(interceptorSocketID, (struct sockaddr *)&socketClient,
                  &lengthClient);
  if(socket < 0) {
    if (errno == EWOULDBLOCK) {
      perror("Error accept Interceptor Client");
      return -1;
    }
    perror("Error accept Interceptor Client");
    return -1;
  }

  /* get connection table lock */
  ls = pthread_mutex_lock(&mutexCT); /* get lock */
  if(ls != 0) {
    perror("error getting connection table lock");
    return -1;
  }

  /* get new connection table id */
  GetConnectionID(&id);

  /* set up new connection table entry */
  rc = AddEntryToConnectionTable(id, -1, socket,
                                 (char *)inet_ntoa(socketClient.sin_addr));
  if(rc == -1) {
    fprintf(stderr, "error setting up connection table entry\n");
    return -1;
  }

  printf("--- got client with id: %i, connecting to MDS ... ---\n", id);

  /* Got client, tell Transis to connect the Interceptor nodes to their MDS */
  rc = EditMDSLock(id, SET); /* set MDS Lock! */
  if(rc == -1)
    return -1;

  /* release connection table lock */
  ls = pthread_mutex_unlock(&mutexCT); /* release lock */
  if(ls != 0) {
    perror("error releasing connection table lock");
    return -1;
  }
```

```

#ifndef TRANSIS_BYPASS
  /* set up header data for transis message */
  header       = (__u32 *)BufferToTransis; /* pointer to beginning of message */
  *(header++) = CREATE_CONNECTION; /* type of the message (specified in transis.h) */
  *(header++) = (4*sizeof(__u32)); /* size of the message */
  *(header++) = id;                /* identifier of entry in the connection table */
  *(header++) = NO_TARGET;         /* target of the message (No, Client or MDS) */
  /* send message */
  rc = SendMessageToTransis(BufferToTransis, (4*sizeof(__u32)));
  if(rc == -1)
    return -1;
#else
  rc = ConnectToMDS(id);
  if(rc == -1)
    return -1;
#endif

  /* wait for MDS lock release; if released, connection to MDS is established */
  do {
    /* get connection table lock */
    ls = pthread_mutex_lock(&mutexCT);  /* get lock */
    if(ls != 0) {
      perror("error getting connection table lock");
      return -1;
    }
    /* get MDS lock status */
    rc = GetMDSLock(id);
    /* release connection table lock */
    ls = pthread_mutex_unlock(&mutexCT);  /* release lock */
    if(ls != 0) {
      perror("error releasing connection table lock");
      return -1;
    }
    if(rc == -1)
      return -1;

  }while (rc != UNSET);

  return 0;
}


// ------------------------------------------------------------------------
// establish connection to the MDS
//
// uses local secure port (Acceptor Port) to connect to the MDS,
// after connection is set up, the connection table is updated
// and the MDS lock is released
//
// id - connection identifier
//
// returns: 0 if success / -1 if error occurs
// ------------------------------------------------------------------------
int ConnectToMDS (int id)
{
  int rc;
  int option;
  int mdsSocketID;
  struct sockaddr_in socketServer;
  struct sockaddr_in socketConnect;

  mdsSocketID = socket(PF_INET, SOCK_STREAM, 0);
```

```
  if(mdsSocketID == -1) {
    perror("Error, can't create MDS Socket!");
    return -1;
  }

  /* set socket options */
  option = 1;
  rc = setsockopt(mdsSocketID, SOL_SOCKET, SO_REUSEADDR,
                  (char *)&option, sizeof(option));
  if(rc != 0) {
    perror("Error, can't set socket options for MDS Socket!");
    return -1;
  }

  /* bind socket to local secure port */
  socketServer.sin_family      = AF_INET;
  socketServer.sin_port        = htons(LusterAcceptorPort--);
  socketServer.sin_addr.s_addr = inet_addr(INTERCEPTOR_ADDR);
  /* bind socket */
  rc = bind(mdsSocketID, (struct sockaddr *)&socketServer, sizeof(socketServer));
  if(rc != 0) {
    perror("error binding local secure MDS port");
    return -1;
  }

  /* set up MDS data */
  socketConnect.sin_family      = AF_INET;
  socketConnect.sin_port        = htons(LUSTRE_SERVER_PORT);
  socketConnect.sin_addr.s_addr = inet_addr(LUSTRE_MDS_ADDR);
  /* connect socket */
  rc = connect(mdsSocketID, (struct sockaddr *)&socketConnect, sizeof(socketConnect));
  if( rc != 0 ) {
    perror("Error connecting to Lustre MDS");
    return -1;
  }

  /* get connection table lock */
  rc = pthread_mutex_lock(&mutexCT); /* get lock */
  if(rc != 0){
    perror("error getting connection table lock");
    return -1;
  }

  /* check if entry in connection table already exists, and make_new/edit_old entry */
  rc = CheckConnectionID(id);
  if(rc == 0){
    /* no entry in table */
    rc = AddEntryToConnectionTable (id, mdsSocketID, -1, NULL);
    if(rc == -1){
      close(mdsSocketID);
      /* release connection table lock */
      rc = pthread_mutex_unlock(&mutexCT); /* release lock */
      if(rc != 0){
        perror("error releasing connection table lock");
        return -1;
      }
      return -1;
    }
  }else{
    /* found entry in table */
    rc = EditConnectionTableEntry (id, mdsSocketID, -1, NULL);
    if(rc == -1){
      close(mdsSocketID);
```

```
      /* release connection table lock */
      rc = pthread_mutex_unlock(&mutexCT); /* release lock */
      if(rc != 0){
        perror("error releasing connection table lock");
        return -1;
      }
      return -1;
    }
  }

  /* release MDS Lock! */
  rc = EditMDSLock(id, UNSET);
  if(rc == -1){
    /* release connection table lock */
    rc = pthread_mutex_unlock(&mutexCT); /* release lock */
    if(rc != 0){
      perror("error releasing connection table lock");
      return -1;
    }
    return -1;
  }

  /* release connection table lock */
  rc = pthread_mutex_unlock(&mutexCT); /* release lock */
  if(rc != 0){
    perror("error releasing connection table lock");
    return -1;
  }

  printf("connection with id: %i connected to MDS\n", id);
  return 0;
}


// -----------------------------------------------------------------------
// receives LUSTRE ACCEPTOR_REQUEST and passes the message on to Transis
//
// id     - connection identifier
// socket - the socket identifier
// target - indicate the target of the message (MDS, CLIENT)
//
// returns: 0 if success / -1 if error occurs / -2 if peer closed connection
// -----------------------------------------------------------------------
int ReceiveAcceptorRequest (int id, int socket, int target)
{
  int rc;
  int ls;
  __u32 *header;
  __u32 messageLength = (4*sizeof(__u32)) + sizeof(lnet_acceptor_connreq_t);

  /* set up header for transis message */
  header      = (__u32 *)BufferToTransis; /* pointer to beginning of message */
  *(header++) = LUSTRE_ACCEPTOR_CONNREQ;  /* type of the message (see transis.h) */
  *(header++) = messageLength;            /* size of the message */
  *(header++) = id;                       /* identifier of entry in connection table */
  *(header++) = target;                   /* target of message (No, Client or MDS) */

  /* receive acceptor request and put behind the header */
  rc = ReceiveBuffer(socket, header, sizeof(lnet_acceptor_connreq_t), BLOCK);
  switch (rc) {
    case -1:
      fprintf(stderr, "Error receiving acceptor request.\n");
      return -1;
```

```
      break;
    case -2:
      fprintf(stderr,
              "ReceiveAcceptorRequest - peer closed connection; id: %i; socket: %i\n",
              id, socket);
      return -2;
      break;
    default:
      if(rc != sizeof(lnet_acceptor_connreq_t)) {
        fprintf(stderr, "Didn't receive complete acceptor request structure.\n");
        return -1;
      }
      break;
  }

#ifdef DEBUG
  {
    int  fileTemp;
    char fileName[30];
    char fileNumber[20];

    strcpy (fileName,"recv");
    sprintf(fileNumber, "%d", fileCounterR++);
    strcat (fileName, fileNumber);

    fileTemp=open(fileName, O_CREAT | O_TRUNC | O_RDWR, 0666 );
    if(fileTemp < 0){
      perror("error creating file");
      return -1;
    }

    rc = write(fileTemp, header, sizeof(lnet_acceptor_connreq_t));
    if(rc == -1){
      perror("error writing to debug file");
      return -1;
    }

    rc = close(fileTemp);
    if(rc == -1){
      perror("error closing debug file");
      return -1;
    }
  }
#endif

#ifndef TRANSIS_BYPASS
  /* send message to Transis */
  rc = SendMessageToTransis(BufferToTransis, messageLength);
  if(rc == -1) {
    fprintf(stderr, "error sending acceptor request\n");
    return -1;
  }
#endif

  /* get connection table lock */
  ls = pthread_mutex_lock(&mutexCT);   /* get lock */
  if(ls != 0){
    perror("error getting connection table lock");
    return -1;
  }

  /* set message type to the next in Lustre protocol */
  rc = SetMessageType (id, LUSTRE_LNET_HELLO);
```

```
  /* release connection table lock */
  ls = pthread_mutex_unlock(&mutexCT);  /* release lock */
  if(ls != 0){
    perror("error releasing connection table lock");
    return -1;
  }
  if(rc == -1)
    return -1;

#ifdef TRANSIS_BYPASS
  /* Check message and pass on to Lustre */
  rc = CheckAndSendAcceptorRequest();
  if(rc == -1)
    return -1;
#endif

  return 0;
}


// ------------------------------------------------------------------
// receives LUSTRE LNET_HELLO and passes the message on to Transis
//
// id     - connection identifier
// socket - the socket identifier
// target - indicate the target of the message (MDS, CLIENT)
//
// returns: 0 if success / -1 if error occurs / -2 if peer closed connection
// ------------------------------------------------------------------
int ReceiveLNETHello (int id, int socket, int target)
{
  int rc;
  int ls;
  lnet_hdr_t *hdr;     /* pointer to Lustre LNET header */
  __u32      *header;
  __u32      messageLength = (4*sizeof(__u32)) + sizeof(lnet_hdr_t);

  /* set up header for transis message */
  header      = (__u32 *)BufferToTransis; /* pointer to beginning of message */
  *(header++) = LUSTRE_LNET_HELLO;        /* type of the message (see transis.h) */
  *(header++) = messageLength;            /* size of the message */
  *(header++) = id;                       /* identifier of entry in connection table */
  *(header++) = target;                   /* target of message (No, Client or MDS) */

  /* receive LNET hello and put behind the header */
  rc = ReceiveBuffer(socket, header, sizeof(lnet_hdr_t), BLOCK);
  switch (rc) {
    case -1:
      fprintf(stderr, "Error receiving LNET hello.\n");
      return -1;
      break;
    case -2:
      fprintf(stderr,
              "ReceiveLNETHello - peer closed connection; id: %i; socket: %i\n",
              id, socket);
      return -2;
      break;
    default:
      if(rc != sizeof(lnet_hdr_t)) {
        fprintf(stderr, "Didn't receive complete LNET hello header.\n");
        return -1;
      }
      break;
```

```
  }

  /* check for payload */
  hdr = (lnet_hdr_t *)header;
  if(hdr->payload_length != 0){
    fprintf(stderr, "got payload in LNET Hello header!!!\n");
    return -1;
  }

#ifdef DEBUG
  {
    int  fileTemp;
    char fileName[30];
    char fileNumber[20];

    strcpy (fileName,"recv");
    sprintf(fileNumber, "%d", fileCounterR++);
    strcat (fileName, fileNumber);

    fileTemp=open(fileName, O_CREAT | O_TRUNC | O_RDWR, 0666 );
    if(fileTemp < 0){
      perror("error creating file");
      return -1;
    }

    rc = write(fileTemp, header, sizeof(lnet_hdr_t));
    if(rc == -1){
      perror("error writing to debug file");
      return -1;
    }

    rc = close(fileTemp);
    if(rc == -1){
      perror("error closing debug file");
      return -1;
    }
  }
#endif

#ifndef TRANSIS_BYPASS
  /* send message to Transis */
  rc = SendMessageToTransis(BufferToTransis, messageLength);
  if(rc == -1) {
    fprintf(stderr, "error sending LNET hello header\n");
    return -1;
  }
#endif

  /* set message type to the next in Lustre protocol */
  if(target == CLIENT){
    /* get connection table lock */
    ls = pthread_mutex_lock(&mutexCT);   /* get lock */
    if(ls != 0){
      perror("error getting connection table lock");
      return -1;
    }
    /* set message type */
    rc = SetMessageType (id, LUSTRE_MESSAGE);
    /* release connection table lock */
    ls = pthread_mutex_unlock(&mutexCT);   /* release lock */
    if(ls != 0){
      perror("error releasing connection table lock");
      return -1;
```

```
    }
    if(rc == -1)
      return -1;
  }

#ifdef TRANSIS_BYPASS
  /* Check message and pass on to Lustre */
  rc = CheckAndSendLNETHello();
  if(rc == -1)
    return -1;
#endif

  return 0;
}


// ----------------------------------------------------------------------------
// receives LUSTRE message and passes the message on to Transis
//
// id     - connection identifier
// socket - the socket identifier
// target - indicate the target of the message (MDS, CLIENT)
//
// returns: 0 if success / -1 if error occurs / -2 if peer closed connection
// ----------------------------------------------------------------------------
int ReceiveLustreMessage (int id, int socket, int target)
{
  int rc;
  lnet_hdr_t *hdr;      /* pointer to Lustre message header */
  __u32        *header;
  __u32        *messageLength;

  /* set up header for transis message */
  header        = (__u32 *)BufferToTransis; /* pointer to beginning of message */
  *(header++)   = LUSTRE_MESSAGE;           /* type of the message (see transis.h) */
  messageLength = header++;                 /* pointer to size of message in header */
  *(header++)   = id;                       /* id of entry in connection table */
  *(header++)   = target;                   /* target of message (No, Client or MDS) */

  /* get the Lustre message header and put behind transis message header */
  rc = ReceiveBuffer(socket, header, sizeof(lnet_hdr_t), BLOCK);
  switch (rc) {
    case -1:
      fprintf(stderr, "Error receiving Message.\n");
      return -1;
      break;
    case -2:
      fprintf(stderr, "ReceiveLustreMessage, header - peer closed connection;\
              id: %i; socket: %i\n", id, socket);
      return -2;
      break;
    default:
      if(rc != sizeof(lnet_hdr_t)) {
        fprintf(stderr, "Didn't receive complete message header.\n");
        return -1;
      }
      break;
  }

  /* check for Payload length */
  hdr = (lnet_hdr_t *)header;
  if ( (hdr->payload_length + sizeof(lnet_hdr_t)) > MESSAGE_BUFFER_SIZE ) {
    fprintf(stderr, "Bad payload length %ld\n", le32_to_cpu (hdr->payload_length));
```

```
      return -1;
    }

    /* get payload if needed */
    if (hdr->payload_length > 0) {
      /* receive payload and put behind Lustre message header */
      rc = ReceiveBuffer(socket, (__u8 *)(header + (sizeof(lnet_hdr_t)/sizeof(__u32))),
                         hdr->payload_length, BLOCK);
      switch (rc) {
        case -1:
          fprintf(stderr, "Error receiving Message.\n");
          return -1;
          break;
        case -2:
          fprintf(stderr, "ReceiveLustreMessage, payload - peer closed connection;\
                  id: %i; socket: %i\n", id, socket);
          return -2;
          break;
        default:
          if(rc != hdr->payload_length) {
            fprintf(stderr, "Didn't receive complete message payload.\n");
            return -1;
          }
          break;
      }
    }

#ifdef DEBUG
    {
      int  fileTemp;
      char fileName[30];
      char fileNumber[20];

      strcpy (fileName,"recv");
      sprintf(fileNumber, "%d", fileCounterR++);
      strcat (fileName, fileNumber);

      fileTemp=open(fileName, O_CREAT | O_TRUNC | O_RDWR, 0666 );
      if(fileTemp < 0){
        perror("error creating file");
        return -1;
      }

      rc = write(fileTemp, header, sizeof(lnet_hdr_t) + hdr->payload_length);
      if(rc == -1){
        perror("error writing to debug file");
        return -1;
      }

      rc = close(fileTemp);
      if(rc == -1){
        perror("error closing debug file");
        return -1;
      }
    }
#endif

    /* set message length in transis message header */
    *messageLength = (4*sizeof(__u32)) + sizeof(lnet_hdr_t) + hdr->payload_length;

#ifndef TRANSIS_BYPASS
    /* send message to Transis */
    rc = SendMessageToTransis(BufferToTransis, *messageLength);
```

```
  if(rc == -1) {
    fprintf(stderr, "error sending Lustre Message\n");
    return -1;
  }
#else
  /* Check message and pass on to Lustre */
  rc = CheckAndSendMessage();
  if(rc == -1)
    return -1;
#endif

  return 0;
}


// ---------------------------------------------------------------------------
// Reads a buffer from a file descriptor (non-/blocking).
//
// fd     - The file descriptor to read from.
// buffer - The buffer to read into.
// length - The maximum buffer length to read.
// block  - The (non-)blocking flag (0 = non-blocking, 1 = blocking).
//
// returns: number of bytes read on success, -2 on closed file descriptor
//          or -1 on any other error with errno set appropriately.
// ---------------------------------------------------------------------------
int ReceiveBuffer (int fd, void *buffer, unsigned int length, unsigned int block)
{
  int          bytes;
  unsigned int index;

  for (index = 0; index < length;) {
    /* Read some data. */
    switch (bytes = read(fd, buffer + index, length - index)) {
      case -1: {
        switch (errno) {
          case EINTR: {
            break;
          }
          case EAGAIN: {
            if (0 == block) {
              return index;
            }
            break;
          }
          default: {
            perror("unable to read from file descriptor");
            return -1;
          }
        }
        break;
      }
      case 0: {
        errno = EPIPE;
        if (0 != index) {
          perror("unable to read from closed file descriptor");
        }
        return -2;
      }
      default: {
        index += bytes;
        if (0 == block) {
          return index;
```

```
          }
        }
      }
    }
    return index;
}


// ------------------------------------------------------------------------
// Writes a buffer into a file descriptor (blocking).
//
// fd      - The file descriptor to write to.
// buffer  - The buffer to write from.
// length  - The buffer length to write.
//
// returns: 0 on success, -2 on closed file descriptor or -1 on any
//          other error with errno set appropriately.
// ------------------------------------------------------------------------
int SendBuffer (int fd, const void *buffer, unsigned int length)
{
    int          bytes;
    unsigned int index;

    for (index = 0; index < length;) {
        /* Write some data. */
        switch (bytes = write(fd, buffer + index, length - index)) {
            case -1: {
                switch (errno) {
                    case EINTR:
                    case EAGAIN: {
                        break;
                    }
                    case EPIPE: {
                        if (0 != index) {
                            perror("unable to write to closed file descriptor");
                        }
                        return -2;
                    }
                    default: {
                        perror("unable to write to file descriptor");
                        return -1;
                    }
                }
                break;
            }
            default: {
                index += bytes;
            }
        }
    }
    return 0;
}


// ------------------------------------------------------------------------
// Add entry to connection table
//
// id           - identifier of the connection
// MDSSocket    - number of socket to MDS, -1 if not connected
// ClientSocket - number of socket to Client, -1 if not connected
// ipAddress    - the IP Address of the Client, NULL if no enty
//
// returns: 0 on success / -1 if error occurs
```

```c
// ---------------------------------------------------------------------------
int AddEntryToConnectionTable(int id, int MDSSocket, int ClientSocket, char *ipAddress)
{
  int  index;
  void *connection = NULL;

  /* Increase registry size. */
  index = connectionTable->count;
  connectionTable->count++;

  /* Reallocate registry. */
  if (NULL == (connection = realloc(connectionTable->connection,
     (connectionTable->count * sizeof(connectionTable->connection[0]))))) {
    perror("realloc");
    return -1;
  }
  connectionTable->connection = connection;

  /* Set connection entries. */
  connectionTable->connection[index].id           = id;
  connectionTable->connection[index].MDSLock      = UNSET;
  connectionTable->connection[index].MDSSocket    = MDSSocket;
  connectionTable->connection[index].ClientSocket = ClientSocket;
  connectionTable->connection[index].MessageType  = LUSTRE_ACCEPTOR_CONNREQ;
  if(ipAddress != NULL)
    strcpy(connectionTable->connection[index].IPAddress, ipAddress);
  else
    strcpy(connectionTable->connection[index].IPAddress, "0.0.0.0");

  return 0;
}


// ---------------------------------------------------------------------------
// Edit entry in the connection table
//
// id:           - the entry with the given id will be edited
// MDSSocket     - number of socket to MDS, -1 if not to be set
// ClientSocket - number of socket to Client, -1 if not to be set
// ipAddress     - the IP Address of the Client, NULL if not to be set
//
// returns: 0 on success / -1 if error occurs
// ---------------------------------------------------------------------------
int EditConnectionTableEntry (int id, int MDSSocket, int ClientSocket, char *ipAddress)
{
  int i;
  int index = -1;

  /* get index of id */
  for(i=0; i<connectionTable->count; i++) {
    if(connectionTable->connection[i].id == id){
      index = i;
      break;
    }
  }

  /* id not found */
  if(index == -1){
    fprintf(stderr, "Error editing connection table entry: id not found!\n");
    return -1;
  }

  /* Edit connection entries. */
```

```c
  connectionTable->connection[index].MessageType      = LUSTRE_ACCEPTOR_CONNREQ;
  if(MDSSocket != -1)
    connectionTable->connection[index].MDSSocket      = MDSSocket;
  if(ClientSocket != -1)
    connectionTable->connection[index].ClientSocket = ClientSocket;
  if(ipAddress != NULL)
    strcpy(connectionTable->connection[index].IPAddress, ipAddress);

  return 0;
}


// -------------------------------------------------------------------------
// Remove entry from connection table
//
// id - the entry with the given id will be removed
//
// returns: 0 on success / -1 if error occurs
// -------------------------------------------------------------------------
int RemoveEntryFromConnectionTable (int id)
{
  int   i;
  int   index = -1;
  void *connection = NULL;

  /* get index of id */
  for(i=0; i<connectionTable->count; i++) {
    if(connectionTable->connection[i].id == id){
      index = i;
      break;
    }
  }

  /* id not found */
  if(index == -1){
    fprintf(stderr, "Error removing connection from table: id not found!\n");
    return -1;
  }

  /* Remove entry from registry. */
  connectionTable->count--;

  memmove(connectionTable->connection + index, connectionTable->connection + index + 1,
          (connectionTable->count - index) * sizeof(connectionTable->connection[0]) );

  /* Reallocate registry. */
  if (0 == connectionTable->count) {
    free(connectionTable->connection);
    connectionTable->connection = NULL;
  } else if (NULL == (connection = realloc(connectionTable->connection,
                connectionTable->count * sizeof(connectionTable->connection[0])))) {
    perror("realloc");
    return -1;
  } else {
    connectionTable->connection = connection;
  }

  return 0;
}


// -------------------------------------------------------------------------
// Function returns an unused connection id
```

```
//
// *id - pointer to the returned id
//
// ---------------------------------------------------------------------------
void GetConnectionID (int *id)
{
  int rc;
  int rn;

  do {
    /* generate random number */
    rn = random();

    /* check if random number is already used, if not use it as id */
    rc = CheckConnectionID(rn);
    if(0 == rc) {
      *id = rn;
      return;
    }
  } while(1);
}


// ---------------------------------------------------------------------------
// Checks if connection ID is already used
//
// id - the connection id to check
//
// returns: 0 if id is not used / -1 if id is already used
// ---------------------------------------------------------------------------
int CheckConnectionID (int id)
{
  int i;

  /* check if id is already used */
  for(i=0; i<connectionTable->count; i++) {
    if(connectionTable->connection[i].id == id){
      return -1;
    }
  }

  return 0;
}


// ---------------------------------------------------------------------------
// Returns the number of entries in the connection table
//
//
// returns: >=0 the number of entries
// ---------------------------------------------------------------------------
int GetNumberOfEnties ()
{
  return connectionTable->count;
}


// ---------------------------------------------------------------------------
// gets the socket id from the connection table
//
// id      - connection identifier
// choose  - indicate the socket to get back (MDS, CLIENT)
// *socket - pointer to hold the socket identifier
```

```
//
// returns: 0 if success / -1 if error occurs / -2 if not connected
// ---------------------------------------------------------------------
int GetSocketFromConnectionTable (int id, int choose, int *socket)
{
  int i;

  /* look for connection */
  for(i=0; i<connectionTable->count; i++) {
    if(connectionTable->connection[i].id == id) {
      if(choose == MDS)   /* need MDS Socket */
        *socket = connectionTable->connection[i].MDSSocket;
      else   /* need Client Socket */
        *socket = connectionTable->connection[i].ClientSocket;
      /* check for connection */
      if(*socket == -1)
        return -2; /* not connected */
      else
        return 0;  /* return socket id */
    }//if
  }//for

  return -1;
}


// ---------------------------------------------------------------------
// Returns the MDS Lock status for the given table enty
//
// id - connection identifier
//
// returns: -1 if error occurs / 0 (UNSET) if Lock is not set /
//              1 (SET) if Lock is set
// ---------------------------------------------------------------------
int GetMDSLock (int id)
{
  int i;

  /* look for connection enty */
  for(i=0; i<connectionTable->count; i++) {
    if(connectionTable->connection[i].id == id) {
      /* check status */
      switch (connectionTable->connection[i].MDSLock) {
        case SET:
          return SET;
          break;
        case UNSET:
          return UNSET;
        default:
          break;
      }//switch
    }//if
  }//for

  fprintf(stderr,
          "error finding, or false MDS Lock entry for connection with id: %i\n", id);
  return -1;
}


// ---------------------------------------------------------------------
// Set/Unset the MDS Lock from the given entry in the connection table
//
```

```
// id           - connection identifier
// lockStatus - the status to set the MDSLock to
//
// returns: 0 if success / -1 if error occurs
// ---------------------------------------------------------------------------
int EditMDSLock (int id, int lockStatus)
{
  int i;

  /* look for connection entry */
  for(i=0; i<connectionTable->count; i++) {
    if(connectionTable->connection[i].id == id) {
      /* set/unset the Lock */
      connectionTable->connection[i].MDSLock = lockStatus;
      return 0;
    }//if
  }//for

  fprintf(stderr, "cannot set/unset MDS Lock for connection with id: %i\n", id);
  return -1;
}


// ---------------------------------------------------------------------------
// gets the message type of an connection table enty
//
// id              - connection identifier
// *messageType - pointer to hold the message type
//
// returns: 0 if success / -1 if error occurs
// ---------------------------------------------------------------------------
int GetMessageType (int id, int *messageType)
{
  int i;

  /* look for connection */
  for(i=0; i<connectionTable->count; i++) {
    if(connectionTable->connection[i].id == id) {
      *messageType = connectionTable->connection[i].MessageType;
      return 0;
    }//if
  }//for

  fprintf(stderr, "could not get message type\n");
  return -1;
}


// ---------------------------------------------------------------------------
// Sets the message type of an connection table enty
//
// id             - connection identifier
// messageType - the message type to set enty to
//
// returns: 0 if success / -1 if error occurs
// ---------------------------------------------------------------------------
int SetMessageType (int id, int messageType)
{
  int i;

  /* look for connection */
  for(i=0; i<connectionTable->count; i++) {
    if(connectionTable->connection[i].id == id) {
```

```
      connectionTable->connection[i].MessageType = messageType;
      return 0;
    }//if
  }//for

  fprintf(stderr, "could not set message type\n");
  return -1;
}


// ---------------------------------------------------------------------------
// Application main entry point
//
//
// programm exits or breaks up only here
// ---------------------------------------------------------------------------
int main ( int argc, char *argv[] )
{
  int rc;
  connection_table_t  connTab; /* the connection table */

  /* set up the connection table */
  connectionTable             = (connection_table_t *)&connTab;
  connectionTable->connection = NULL;
  connectionTable->count      = 0;

  /* release connection table lock */
  rc = pthread_mutex_unlock(&mutexCT); /* release lock */
  if(rc != 0)
    exit(-1);

  rc = GetHostInfo();
  if(rc == -1)
    exit(-1);

#ifndef TRANSIS_BYPASS
  rc = SetUpTransis();
  if(rc == -1)
    exit(-1);

  rc = StartTransisReceiveThread();
  if(rc == -1)
    exit(-1);
#endif

#ifdef FAKE_MDS
  for(;;){} /* Let Transis run ... */
#else
  rc = StartInterceptorServer();
  if(rc == -1)
    exit(-1);

  rc = MessagePassOn();
  if(rc == -1)
    exit(-1);
#endif

#ifndef TRANSIS_BYPASS
  rc = LeaveTransis();
  if(rc == -1)
    exit(-1);
#endif

```

```
  exit(0);
}


// ---------------------------------------------------------------------------
// End of file
// ---------------------------------------------------------------------------
```

A.1.2 lustreHAdaemon.h

```
// ---------------------------------------------------------------------------
// Lustre High Availability Daemon
//
// lustreHAdaemon.h  --header file--
//
// version 0.52rev
//
// by Matthias Weber
// ---------------------------------------------------------------------------

#ifndef LUSTREHADAEMON_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <errno.h>
#include <pthread.h>
#include <stddef.h>
#include <ctype.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <unistd.h>


// Defines
#define HOSTNAME_LENGTH 20
#define NUM_CONNECTIONS 10
/* MDS/Connection Table Lock defines */
#define SET    1
#define UNSET  0
typedef struct {
  unsigned int count;    /* number of connections */
  struct {
    int  id;             /* the connection id of entry */
    char IPAddress[20];  /* the IP address of the client, NULL if not connected */
    int  MDSLock;        /* connection to MDS in progress (1) / established (0) */
    int  MDSSocket;      /* identifier of MDS socket */
    int  ClientSocket;   /* identifier of Client socket, -1 if no connection exists */
    int  MessageType;    /* next message according to Lustre Protocol (transis.h) */
  } *connection;         /* connection information struct */
} connection_table_t;


// Prototypes
int  GetHostInfo                  ();
int  StartInterceptorServer       ();
```

```
int  MessagePassOn                     ();
int  CloseConnection                   (int id);
int  GetNewClient                      ();
int  ConnectToMDS                      (int id);
int  ReceiveAcceptorRequest            (int id, int socket, int target);
int  ReceiveLNETHello                  (int id, int socket, int target);
int  ReceiveLustreMessage              (int id, int socket, int target);
int  ReceiveBuffer                     (int fd, void *buffer, unsigned int length,
                                        unsigned int block);
int  SendBuffer                        (int fd, const void *buffer, unsigned int length);
int  AddEntryToConnectionTable         (int id, int MDSSocket, int ClientSocket,
                                        char *ipAddress);
int  EditConnectionTableEntry          (int id, int MDSSocket, int ClientSocket,
                                        char *ipAddress);
int  RemoveEntryFromConnectionTable    (int id);
void GetConnectionID                   (int *id);
int  CheckConnectionID                 (int id);
int  GetNumberOfEnties                 ();
int  GetSocketFromConnectionTable      (int id, int choose, int *socket);
int  GetMDSLock                        (int id);
int  EditMDSLock                       (int id, int lockStatus);
int  GetMessageType                    (int id, int *messageType);
int  SetMessageType                    (int id, int messageType);


// Globals
extern struct hostent    *hostinfo;    /* hold host information */
extern pthread_mutex_t   mutexCT;      /* pthread lock for connection table */

#endif


// -----------------------------------------------------------------------------
// End of file
// -----------------------------------------------------------------------------
```

A.1.3 transis.c

```
// -----------------------------------------------------------------------------
// Lustre High Availability Daemon
//
// transis.c  --source file--
//
// version 0.52rev
//
// by Matthias Weber
// -----------------------------------------------------------------------------


#include "transis.h"
#include "lustreHAdaemon.h"
#include "lustreMessageAdjust.h"


// Globals
__u8 fileCounterTR = 0; /* counter for debug files Transis Receive */
__u8 fileCounterTS = 0; /* counter for debug files Transis Send */
__s8 BufferToTransis   [MAX_MSG_SIZE];
__s8 BufferFromTransis [MAX_MSG_SIZE];
pthread_t              ReceiveThread;  /* transis receive thread */
pthread_mutex_t        mutexTRANSIS;   /* pthread lock for transis */
```

```
static zzz_mbox_cap     TransisGroup;  /* Transis Group */


// -----------------------------------------------------------------
// connect to transis deamon, join MDS group,
// and set up receive handler
//
// returns: 0 on success / -1 if error occurs
// -----------------------------------------------------------------
int SetUpTransis ()
{
  /* connect to transis */
  TransisGroup = zzz_Connect(hostinfo->h_name, (void *)0, SET_GROUP_SERVICE);
  if(TransisGroup == 0) {
    fprintf(stderr, "error connecting to transis!\n");
    return -1;
  }

  /* join group */
  zzz_Join(TransisGroup, GROUPNAME);

  /* set up message receive handler */
  zzz_Add_Upcall(TransisGroup, TransisReceiveHandler, USER_PRIORITY, 0);

  return 0;
}


// -----------------------------------------------------------------
// removes receive handler and leaves MDS group
//
// returns: 0 on success / -1 if error occurs
// -----------------------------------------------------------------
int LeaveTransis ()
{
  int rc;

  /* remove receive handler */
  rc = zzz_Remove_Upcall(TransisGroup);
  if(rc == -1){
    fprintf(stderr, "error removing receive handler\n");
    return -1;
  }

  /* leaving group */
  zzz_Leave(TransisGroup, GROUPNAME);

  return 0;
}


// -----------------------------------------------------------------
// starts thread that listens to transis for pending messages
//
// returns: 0 on success / -1 if error occurs
// -----------------------------------------------------------------
int StartTransisReceiveThread ()
{
  int rc;

  /* start thread */
  rc = pthread_create(&ReceiveThread, NULL, Transis_Receive_Thread, NULL);
  if(rc != 0) {
```

```
    perror("error creating Transis receive thread");
    return -1;
  }

  printf("Thread listening to Transis started.\n");
  return 0;
}


// ------------------------------------------------------------------------
// Thread that gives control to Transis. Transis polls for pending
// messages and invokes TransisReceiveHandler to deal with messages.
//
// ------------------------------------------------------------------------
void *Transis_Receive_Thread ()
{
  /* give control to transis */
  E_main_loop();

  pthread_exit(NULL);
}


// ------------------------------------------------------------------------
// handler invoked if transis message is pending
//
// ------------------------------------------------------------------------
void TransisReceiveHandler ()
{
  int rc;

  /* receive pending message */
  rc = ReceiveTransisMessage();
  if(rc == -1){
    fprintf(stderr, "error receiving transis message\n");
  }
}


// ------------------------------------------------------------------------
// check received message from Transis and invoke appropriate
// function to deal with message
//
// returns: 0 on success / -1 if error occurs
// ------------------------------------------------------------------------
int CheckTransisMessage ()
{
#ifndef FAKE_MDS
  int rc;
  __u32 *type;

  /* set pointer to message type */
  type = (__u32 *)BufferFromTransis;

  /* process message */
  switch (*type) {
    case CREATE_CONNECTION:
      rc = ConnectToMDS(*(type+2)); /* *(type+2) pointer to connection id */
      if(rc == -1)
        return -1;
      break;
    case LUSTRE_ACCEPTOR_CONNREQ:
      rc = CheckAndSendAcceptorRequest();
```

```
      if(rc == -1)
        return -1;
      break;
    case LUSTRE_LNET_HELLO:
      rc = CheckAndSendLNETHello();
      if(rc == -1)
        return -1;
      break;
    case LUSTRE_MESSAGE:
      rc = CheckAndSendMessage();
      if(rc == -1)
        return -1;
      break;
    default:
      fprintf(stderr, "Got wrong Transis message type!\n");
      return -1;
      break;
  }
#else
  /* print a dot instead */
  printf(".");
#endif

  return 0;
}


// ----------------------------------------------------------------------
// receives message from Transis
//
// returns: 0 on success / -1 if error occurs
// ----------------------------------------------------------------------
int ReceiveTransisMessage ()
{
  int rc;
  int recvType;
  view *gview;

  /* obtaining lock */
  rc = pthread_mutex_lock(&mutexTRANSIS);
  if(rc != 0) {
    perror("error obtaining transis lock");
    return -1;
  }

  /* receive message */
  rc = zzz_Receive(TransisGroup, BufferFromTransis, MAX_MSG_SIZE, &recvType, &gview);
  if(rc == -1) {
    fprintf(stderr, "error receiving message from Transis.\n");
    return -1;
  }

  /* release lock */
  rc = pthread_mutex_unlock(&mutexTRANSIS);
  if(rc != 0) {
    perror("error releasing transis lock");
    return -1;
  }

  if(recvType != VIEW_CHANGE) {

#ifdef DEBUG
    {
```

```
      __u32 *type;
      type = (__u32 *)BufferFromTransis;
      if(*type != CREATE_CONNECTION){
        int  fileTemp;
        char fileName[30];
        char fileNumber[20];

        strcpy (fileName,"TRrecv");
        sprintf(fileNumber, "%d", fileCounterTR++);
        strcat (fileName, fileNumber);

        fileTemp=open(fileName, O_CREAT | O_TRUNC | O_RDWR, 0666 );
        if(fileTemp < 0){
          perror("error creating file");
          return -1;
        }

        rc = write(fileTemp, BufferFromTransis, rc);
        if(rc == -1){
          perror("error writing to debug file");
          return -1;
        }

        rc = close(fileTemp);
        if(rc == -1){
          perror("error closing debug file");
          return -1;
        }
      }
    }
#endif

    /* process received message */
    rc = CheckTransisMessage();
    if(rc == -1)
      return -1;
  } else {
    /* display new group status */
    printf("change in group configuration:\n");
    printf("  group is %s\n", gview->members[0]);
    printf("  no. of clients is %ld\n", gview->nmembers);
  }

  return 0;
}


// ---------------------------------------------------------------
// sends buffer to Transis
//
// *message      - pointer to the buffer holding the message
// messageLength - length of the message
//
// returns: 0 on success / -1 if error occurs
// ---------------------------------------------------------------
int SendMessageToTransis (char *message, int messageLength)
{
  int rc;

  /* check message length */
  if(messageLength > MAX_MSG_SIZE){
    fprintf(stderr, "error message to big for transis: %i bytes\n", messageLength);
    return -1;
```

```
  }

#ifdef DEBUG
  {
    __u32 *type;
    type = (__u32 *)message;
    if(*type != CREATE_CONNECTION){
      int   fileTemp;
      char  fileName[30];
      char  fileNumber[20];

      strcpy (fileName,"TRsend");
      sprintf(fileNumber, "%d", fileCounterTS++);
      strcat (fileName, fileNumber);

      fileTemp=open(fileName, O_CREAT | O_TRUNC | O_RDWR, 0666);
      if(fileTemp < 0){
        printf("error creating file\n");
        return -1;
      }

      rc = write(fileTemp, message, messageLength);
      if(rc == -1){
        perror("error writing to debug file");
        return -1;
      }

      rc = close(fileTemp);
      if(rc == -1){
        perror("error closing debug file");
        return -1;
      }
    }
  }
#endif

  /* obtaining lock */
  rc = pthread_mutex_lock(&mutexTRANSIS);
  if(rc != 0) {
    perror("error obtaining transis lock");
    return -1;
  }

  /* send messages to transis */
  rc = zzz_VaSend(TransisGroup, AGREED, 0, messageLength, message, GROUPNAME, NULL);
  if(rc < messageLength){
    fprintf(stderr, "error sending message to transis!\n");
    return -1;
  }

  /* release lock */
  rc = pthread_mutex_unlock(&mutexTRANSIS);
  if(rc != 0) {
    perror("error releasing transis lock");
    return -1;
  }

  return 0;
}


// ---------------------------------------------------------------------------
// End of file
```

```
// ---------------------------------------------------------------------------
```

A.1.4 transis.h

```
// ---------------------------------------------------------------------------
// Lustre High Availability Daemon
//
// transis.h --header file--
//
// version 0.52rev
//
// by Matthias Weber
// ---------------------------------------------------------------------------

#ifndef TRANSIS_H

#include <stdio.h>
#include "zzz_layer.h"   /* Transis */
#include "events.h"      /* Transis Event handler */


// Defines
#define GROUPNAME "MDSGroup"
/* define Transis message types */
#define CREATE_CONNECTION         1    /* establish connection to MDS */
#define LUSTRE_ACCEPTOR_CONNREQ  2    /* Lustre acceptor connection request */
#define LUSTRE_LNET_HELLO         3    /* Lustre LNET hello message */
#define LUSTRE_MESSAGE            4    /* ordinary Lustre message */
/* Transis message targets */
#define MDS        0
#define CLIENT     1
#define NO_TARGET -1


// Prototypes
int   SetUpTransis                ();
int   LeaveTransis                ();
int   StartTransisReceiveThread   ();
void  *Transis_Receive_Thread     ();
void  TransisReceiveHandler       ();
int   CheckTransisMessage         ();
int   ReceiveTransisMessage       ();
int   SendMessageToTransis        (char *message, int messageLength);


// Globals
extern char BufferToTransis   [MAX_MSG_SIZE]; /* buffer holding messages to Transis */
extern char BufferFromTransis [MAX_MSG_SIZE]; /* buffer holding messages from Transis */

#endif


// ---------------------------------------------------------------------------
// End of file
// ---------------------------------------------------------------------------
```

A.1.5 lustreMessageAdjust.c

```
// ---------------------------------------------------------------------------
```

```c
// Lustre High Availability Daemon
//
// lustreMessageAdjust.c --source file--
//
// version 0.52rev
//
// by Matthias Weber
// ----------------------------------------------------------------------------


#include "transis.h"
#include "lustreHAdaemon.h"
#include "lustreMessageAdjust.h"


// Globals
char ipString[128];    /* Array to hold ip string for message adjust operations */
__u8 fileCounterS = 0; /* counter for debug files Send */


// ----------------------------------------------------------------------------
// Checks the acceptor request message and passes the message on
//
// returns: 0 on success / -1 if error occurs
// ----------------------------------------------------------------------------
int CheckAndSendAcceptorRequest ()
{
  int    rc;
  int    socket;
  __u32 id;
  __u32 target;
  __u32 *hdrTran;                    /* pointer to transis message header */
  lnet_acceptor_connreq_t *cr;       /* pointer to Lustre acceptor request message */

  /* set pointer to the structures in the buffer */
#ifndef TRANSIS_BYPASS
  hdrTran = (__u32 *)BufferFromTransis;
#else
  hdrTran = (__u32 *)BufferToTransis;
#endif
  cr      = (lnet_acceptor_connreq_t *)(hdrTran+4);

  /* get message data from the transis message header */
  id     = *(hdrTran+2); /* connection id */
  target = *(hdrTran+3); /* message target */

  /* check acceptor request */
  /* check acceptor magic */
  if (!lnet_accept_magic(cr->acr_magic, LNET_PROTO_ACCEPTOR_MAGIC)) {
    fprintf(stderr, "No recognised acceptor magic\n");
    return -1;
  }
  /* check acceptor magic version number */
  if (cr->acr_version != LNET_PROTO_ACCEPTOR_VERSION) {
    fprintf(stderr, "wrong acceptor magic version\n");
    return -1;
  }
  /* check target nid */
  if(0 == strcmp(libcfs_nid2str(cr->acr_nid), INTERCEPTOR_ADDR)) {
    if(target == CLIENT){ /* message target is Client */
      fprintf(stderr, "Acceptor Packet from MDS to Client!!!\n");
      return -1;
    } else { /* message target is MDS */
```

```
      change_string(&cr->acr_nid, LUSTRE_MDS_ADDR);
    }
  }

  /* get connection table lock */
  rc = pthread_mutex_lock(&mutexCT);   /* get lock */
  if(rc != 0){
    perror("error getting connection table lock");
    return -1;
  }

  /* get socket to send message to */
  rc = GetSocketFromConnectionTable (id, target, &socket);
  switch (rc) {
    case 0:
      /* OK, go on... */
      break;
    case -1:
      fprintf(stderr, "error getting socket from connection table\n");
      /* release connection table lock */
      rc = pthread_mutex_unlock(&mutexCT);   /* release lock */
      if(rc != 0){
        perror("error releasing connection table lock");
        return -1;
      }
      return -1;
      break;
    case -2:
      /* OK, no connection, no reply ;) */
      /* release connection table lock */
      rc = pthread_mutex_unlock(&mutexCT);   /* release lock */
      if(rc != 0){
        perror("error releasing connection table lock");
        return -1;
      }
      return 0;
      break;
  }

  /* release connection table lock */
  rc = pthread_mutex_unlock(&mutexCT);   /* release lock */
  if(rc != 0){
    perror("error releasing connection table lock");
    return -1;
  }

#ifdef DEBUG
  {
    int  fileTemp;
    char fileName[30];
    char fileNumber[20];

    strcpy (fileName,"send");
    sprintf(fileNumber, "%d", fileCounterS++);
    strcat (fileName, fileNumber);

    fileTemp=open(fileName, O_CREAT | O_TRUNC | O_RDWR, 0666 );
    if(fileTemp < 0){
      perror("error creating file");
      return -1;
    }

    rc = write(fileTemp, cr, sizeof(lnet_acceptor_connreq_t));
```

```
    if(rc == -1){
      perror("error writing to debug file");
      return -1;
    }

    rc = close(fileTemp);
    if(rc == -1){
      perror("error closing debug file");
      return -1;
    }
  }
#endif

  /* pass on Lustre acceptor request message */
  rc = SendBuffer(socket, cr, sizeof(lnet_acceptor_connreq_t));
  switch (rc) {
    case -1:
      fprintf(stderr, "Error sending Acceptor Request.\n");
      return -1;
      break;
    case -2:
      fprintf(stderr, "peer closed connection.\n");
      return -1;
      break;
    case 0:
      /* OK, go on... */
      break;
  }

  return 0;
}


// ----------------------------------------------------------------
// Checks the LNET hello message and passes the message on
//
// returns: 0 if success / -1 if error occurs
// ----------------------------------------------------------------
int CheckAndSendLNETHello ()
{
  int    rc;
  int    socket;
  __u32 id;
  __u32 target;
  lnet_hdr_t             *hdr;     /* pointer to Lustre message header */
  lnet_magicversion_t    *hmv;     /* pointer to Lustre Magic */
  __u32                  *hdrTran; /* pointer to transis message header */

  /* set pointer to the structures in the buffer */
#ifndef TRANSIS_BYPASS
  hdrTran = (__u32 *)BufferFromTransis;
#else
  hdrTran = (__u32 *)BufferToTransis;
#endif
  hdr     = (lnet_hdr_t *)(hdrTran+4);
  hmv     = (lnet_magicversion_t *)&hdr->dest_nid;

  /* get message data from the transis message header */
  id     = *(hdrTran+2); /* connection id */
  target = *(hdrTran+3); /* message target */

  /* check LNET hello header */
  /* check magic */
```

```
if (hmv->magic != le32_to_cpu (LNET_PROTO_TCP_MAGIC)) {
  fprintf(stderr, "LNET TCP PROTO magic check failed!\n");
  return -1;
}
/* check magic version */
if (hmv->version_major != cpu_to_le16 (LNET_PROTO_TCP_VERSION_MAJOR) ||
    hmv->version_minor != cpu_to_le16 (LNET_PROTO_TCP_VERSION_MINOR)) {
  fprintf(stderr, "LNET TCP PROTO magic version check failed!\n");
  return -1;
}
/* check header type */
if (hdr->type != cpu_to_le32 (LNET_MSG_HELLO)) {
  fprintf(stderr, "Expecting a HELLO header, but got type %ld\n",
          le32_to_cpu(hdr->type));
  return -1;
}
/* check source address */
if (le64_to_cpu(hdr->src_nid) == LNET_NID_ANY) {
  fprintf(stderr, "Expecting a HELLO header with a NID, but got LNET_NID_ANY\n");
  return -1;
}
/* change source address */
if( 0 == strcmp(libcfs_nid2str(hdr->src_nid),CLIENT_ADDR) ||
    0 == strcmp(libcfs_nid2str(hdr->src_nid),LUSTRE_MDS_ADDR) )
  change_string(&hdr->src_nid, INTERCEPTOR_ADDR);

/* get connection table lock */
rc = pthread_mutex_lock(&mutexCT);  /* get lock */
if(rc != 0){
  perror("error getting connection table lock");
  return -1;
}

/* get socket to send message to */
rc = GetSocketFromConnectionTable (id, target, &socket);
switch (rc) {
  case 0:
    /* OK, go on... */
    break;
  case -1:
    fprintf(stderr, "error getting socket from connection table\n");
    /* release connection table lock */
    rc = pthread_mutex_unlock(&mutexCT);  /* release lock */
    if(rc != 0){
      perror("error releasing connection table lock");
      return -1;
    }
    return -1;
    break;
  case -2:
    /* OK, no connection, no reply ;) */
    /* release connection table lock */
    rc = pthread_mutex_unlock(&mutexCT); /* release lock */
    if(rc != 0){
      perror("error releasing connection table lock");
      return -1;
    }
    return 0;
    break;
}

/* release connection table lock */
rc = pthread_mutex_unlock(&mutexCT);  /* release lock */
```

```
  if(rc != 0){
    perror("error releasing connection table lock");
    return -1;
  }

#ifdef DEBUG
  {
    int  fileTemp;
    char fileName[30];
    char fileNumber[20];

    strcpy (fileName,"send");
    sprintf(fileNumber, "%d", fileCounterS++);
    strcat (fileName, fileNumber);

    fileTemp=open(fileName, O_CREAT | O_TRUNC | O_RDWR, 0666 );
    if(fileTemp < 0){
      perror("error creating file");
      return -1;
    }

    rc = write(fileTemp, hdr, sizeof(lnet_hdr_t));
    if(rc == -1){
      perror("error writing to debug file");
      return -1;
    }

    rc = close(fileTemp);
    if(rc == -1){
      perror("error closing debug file");
      return -1;
    }
  }
#endif

  /* pass on Lustre LNET hello */
  rc = SendBuffer(socket, hdr, sizeof(lnet_hdr_t));
  switch (rc) {
    case -1:
      fprintf(stderr, "Error sending Message.\n");
      return -1;
      break;
    case -2:
      fprintf(stderr, "peer closed connection.\n");
      return -1;
      break;
    case 0:
      /* OK, go on... */
      break;
  }

  return 0;
}


// ----------------------------------------------------------------------
// Checks a Lustre message and passes the message on
//
// returns: 0 on success / -1 if error occurs
// ----------------------------------------------------------------------
int CheckAndSendMessage ()
{
  int rc;
```

```
  int socket;
  __u32        id;
  __u32        target;
  __u32        transisMessageLength; /* length of the transis message */
  __u32        transisHeaderLength;  /* length of the transis message header */
  lnet_hdr_t   *hdr;                 /* pointer to Lustre message header */
  __u32        *hdrTran;             /* pointer to transis message header */

  /* set pointer to the structures in the buffer */
#ifndef TRANSIS_BYPASS
  hdrTran = (__u32 *)BufferFromTransis;
#else
  hdrTran = (__u32 *)BufferToTransis;
#endif
  hdr     = (lnet_hdr_t *)(hdrTran+4);

  /* get message data from the transis message header */
  transisHeaderLength  = 4*sizeof(__u32); /* length of the transis message header */
  transisMessageLength = *(hdrTran+1);    /* length of the entire transis message */
  id                   = *(hdrTran+2);    /* connection id */
  target               = *(hdrTran+3);    /* message target */

  /* adjust ip addresses in Lustre message header */
  if( 0 == strcmp(libcfs_nid2str(hdr->src_nid),CLIENT_ADDR) ||
      0 == strcmp(libcfs_nid2str(hdr->src_nid),LUSTRE_MDS_ADDR))
    change_string(&hdr->src_nid, INTERCEPTOR_ADDR);

  if(0 == strcmp(libcfs_nid2str(hdr->dest_nid), INTERCEPTOR_ADDR)) {
    if(target == MDS) /* message target is MDS */
      change_string(&hdr->dest_nid, LUSTRE_MDS_ADDR);
    else /* message target is Client */
      change_string(&hdr->dest_nid, CLIENT_ADDR);
  }

  /* get connection table lock */
  rc = pthread_mutex_lock(&mutexCT);  /* get lock */
  if(rc != 0){
    perror("error getting connection table lock");
    return -1;
  }

  /* get socket to send message to */
  rc = GetSocketFromConnectionTable (id, target, &socket);
  switch (rc) {
    case 0:
      /* OK, go on... */
      break;
    case -1:
      fprintf(stderr, "error getting socket from connection table\n");
      /* release connection table lock */
      rc = pthread_mutex_unlock(&mutexCT);  /* release lock */
      if(rc != 0){
        perror("error releasing connection table lock");
        return -1;
      }
      return -1;
      break;
    case -2:
      /* OK, no connection, no reply ;) */
      /* release connection table lock */
      rc = pthread_mutex_unlock(&mutexCT);  /* release lock */
      if(rc != 0){
        perror("error releasing connection table lock");
```

```
          return -1;
        }
        return 0;
        break;
    }

    /* release connection table lock */
    rc = pthread_mutex_unlock(&mutexCT);   /* release lock */
    if(rc != 0){
      perror("error releasing connection table lock");
      return -1;
    }

#ifdef DEBUG
    {
      int   fileTemp;
      char  fileName[30];
      char  fileNumber[20];

      strcpy  (fileName,"send");
      sprintf(fileNumber, "%d", fileCounterS++);
      strcat  (fileName, fileNumber);

      fileTemp=open(fileName, O_CREAT | O_TRUNC | O_RDWR, 0666 );
      if(fileTemp < 0){
        perror("error creating file");
        return -1;
      }

      rc = write(fileTemp, hdr, transisMessageLength-transisHeaderLength);
      if(rc == -1){
        perror("error writing to debug file");
        return -1;
      }

      rc = close(fileTemp);
      if(rc == -1){
        perror("error closing debug file");
        return -1;
      }
    }
#endif

    /* pass on complete Lustre message */
    rc = SendBuffer(socket, hdr, transisMessageLength-transisHeaderLength);
    switch (rc) {
      case -1:
        fprintf(stderr, "Error sending Message.\n");
        return -1;
        break;
      case -2:
        fprintf(stderr, "peer closed connection.\n");
        return -1;
        break;
      case 0:
        /* OK, go on... */
        break;
    }

    return 0;
}
```

```
// ---------------------------------------------------------------------------
// Lustre Code ...
// ---------------------------------------------------------------------------
char * libcfs_nid2str (lnet_nid_t nid)
{
  __u32 addr = LNET_NIDADDR(nid);

  snprintf(ipString, LNET_NIDSTR_SIZE, "%u.%u.%u.%u",
           ((unsigned int)addr >> 24) & 0xff, ((unsigned int)addr >> 16) & 0xff,
           ((unsigned int)addr >> 8) & 0xff, (unsigned int)addr & 0xff);

  return ipString;
}

int libcfs_ip_str2addr (char *str, int nob, __u32 *addr)
{
  int  a;
  int  b;
  int  c;
  int  d;
  int  n = nob;     /* XscanfX */

  /* numeric IP? */
  if (sscanf(str, "%u.%u.%u.%u%n", &a, &b, &c, &d, &n) >= 4 &&
      n == nob &&
      (a & ~0xff) == 0 && (b & ~0xff) == 0 &&
      (c & ~0xff) == 0 && (d & ~0xff) == 0) {
    *addr = ((a<<24)|(b<<16)|(c<<8)|d);
    return 1;
  }
  return 0;
}

void change_string (lnet_nid_t *nid, char *str)
{
  __u32 *addrp;
  __u32 addr = LNET_NIDADDR(*nid);
  __u32 net  = LNET_NIDNET (*nid);

  addrp = &addr;
  libcfs_ip_str2addr(str, strlen(str), addrp);
  *nid  = LNET_MKNID(net,addr);
}

int lnet_accept_magic (__u32 magic, __u32 constant)
{
  return (magic == constant || magic == __swab32(constant));
}


// ---------------------------------------------------------------------------
// End of file
// ---------------------------------------------------------------------------
```

A.1.6 lustreMessageAdjust.h

```
// ---------------------------------------------------------------------------
// Lustre High Availability Daemon
//
// lustreMessageAdjust.h --header file--
//
```

```
// version 0.52rev
//
// by Matthias Weber
// -----------------------------------------------------------------------------


#ifndef LUSTREMESSAGEADJUST_H

#include <sys/uio.h>
#include <sys/types.h>
#include <stdio.h>
#include <stddef.h>
#include <fcntl.h>


// -----------------------------------------------------------------------------
// Lustre Data
// -----------------------------------------------------------------------------

#ifndef __KERNEL__
/* Userpace byte flipping */
# include <endian.h>
# include <byteswap.h>
# define __swab16(x) bswap_16(x)
# define __swab32(x) bswap_32(x)
# define __swab64(x) bswap_64(x)
# define __swab16s(x) do {*(x) = bswap_16(*(x));} while (0)
# define __swab32s(x) do {*(x) = bswap_32(*(x));} while (0)
# define __swab64s(x) do {*(x) = bswap_64(*(x));} while (0)
# if __BYTE_ORDER == __LITTLE_ENDIAN
#  define le16_to_cpu(x) (x)
#  define cpu_to_le16(x) (x)
#  define le32_to_cpu(x) (x)
#  define cpu_to_le32(x) (x)
#  define le64_to_cpu(x) (x)
#  define cpu_to_le64(x) (x)
# else
#  if __BYTE_ORDER == __BIG_ENDIAN
#   define le16_to_cpu(x) bswap_16(x)
#   define cpu_to_le16(x) bswap_16(x)
#   define le32_to_cpu(x) bswap_32(x)
#   define cpu_to_le32(x) bswap_32(x)
#   define le64_to_cpu(x) bswap_64(x)
#   define cpu_to_le64(x) bswap_64(x)
#  else
#   error "Unknown byte order"
#  endif /* __BIG_ENDIAN */
# endif /* __LITTLE_ENDIAN */
#endif /* ! __KERNEL__ */

typedef char                __s8;
typedef unsigned char       __u8;
typedef unsigned short      __u16;
typedef unsigned long       __u32;
typedef unsigned long long  __u64;
typedef __u64               lnet_nid_t;
typedef __u32               lnet_pid_t;

#define LNET_NID_ANY           ((lnet_nid_t) -1)
#define LNET_NIDSTR_SIZE       32 /* size of each one (see below for usage) */
#define LNET_NIDADDR(nid)      ((__u32)((nid) & 0xffffffff))
#define LNET_NIDNET(nid)       ((__u32)(((nid) >> 32)) & 0xffffffff)
#define LNET_MKNID(net,addr)   ((((__u64)(net))<<32)|((__u64)(addr)))
```

```c

#define WIRE_ATTR                  __attribute__((packed))

#define LNET_PROTO_TCP_MAGIC            0xeebc0ded
#define LNET_PROTO_TCP_VERSION_MAJOR    1
#define LNET_PROTO_TCP_VERSION_MINOR    0
#define LNET_PROTO_ACCEPTOR_MAGIC       0xacce7100
#define LNET_PROTO_ACCEPTOR_VERSION     1

typedef enum {
  LNET_MSG_ACK = 0,
  LNET_MSG_PUT,
  LNET_MSG_GET,
  LNET_MSG_REPLY,
  LNET_MSG_HELLO,
} lnet_msg_type_t;

/* The wire handle's interface cookie only matches one network interface in
 * one epoch (i.e. new cookie when the interface restarts or the node
 * reboots).  The object cookie only matches one object on that interface
 * during that object's lifetime (i.e. no cookie re-use). */
typedef struct {
  __u64 wh_interface_cookie;
  __u64 wh_object_cookie;
} WIRE_ATTR lnet_handle_wire_t;

/* The variant fields of the portals message header are aligned on an 8
 * byte boundary in the message header.  Note that all types used in these
 * wire structs MUST be fixed size and the smaller types are placed at the
 * end. */
typedef struct lnet_ack {
  lnet_handle_wire_t  dst_wmd;
  __u64               match_bits;
  __u32               mlength;
} WIRE_ATTR lnet_ack_t;

typedef struct lnet_put {
  lnet_handle_wire_t  ack_wmd;
  __u64               match_bits;
  __u64               hdr_data;
  __u32               ptl_index;
  __u32               offset;
} WIRE_ATTR lnet_put_t;

typedef struct lnet_get {
  lnet_handle_wire_t  return_wmd;
  __u64               match_bits;
  __u32               ptl_index;
  __u32               src_offset;
  __u32               sink_length;
} WIRE_ATTR lnet_get_t;

typedef struct lnet_reply {
  lnet_handle_wire_t  dst_wmd;
} WIRE_ATTR lnet_reply_t;

typedef struct lnet_hello {
  __u64 incarnation;
  __u32 type;
} WIRE_ATTR lnet_hello_t;

typedef struct {
  lnet_nid_t  dest_nid;
```

```
    lnet_nid_t   src_nid;
    lnet_pid_t   dest_pid;
    lnet_pid_t   src_pid;
    __u32        type;               /* lnet_msg_type_t */
    __u32        payload_length;     /* payload data to follow */
    /*<------__u64 aligned------->*/
    union {
      lnet_ack_t    ack;
      lnet_put_t    put;
      lnet_get_t    get;
      lnet_reply_t  reply;
      lnet_hello_t  hello;
    } msg;
} WIRE_ATTR lnet_hdr_t;

typedef struct {
    __u32    magic;                  /* LNET_PROTO_TCP_MAGIC */
    __u16    version_major;          /* increment on incompatible change */
    __u16    version_minor;          /* increment on compatible change */
} WIRE_ATTR lnet_magicversion_t;

typedef struct {
    __u32    acr_magic;       /* PTL_ACCEPTOR_PROTO_MAGIC */
    __u32    acr_version;     /* protocol version */
    __u64    acr_nid;         /* target NID */
} lnet_acceptor_connreq_t;


// -----------------------------------------------------------------------
// Interceptor Data
// -----------------------------------------------------------------------

// Defines
#ifdef INTERCEPTOR_CLIENT
#define INTERCEPTOR_ADDR      "10.0.0.12"
#define LUSTRE_MDS_ADDR       "10.0.0.10"
#elif  INTERCEPTOR_CLIENT_ALONE
#define INTERCEPTOR_ADDR      "10.0.0.12"
#define LUSTRE_MDS_ADDR       "10.0.0.5"
#else
#define INTERCEPTOR_ADDR      "10.0.0.10"
#define LUSTRE_MDS_ADDR       "10.0.0.5"
#endif
#define CLIENT_ADDR           "10.0.0.1"
#define LUSTRE_SERVER_PORT    988
#define LUSTRE_MIN_ACC_PORT   512
#define LUSTRE_MAX_ACC_PORT   1023
#define MESSAGE_BUFFER_SIZE   4168  /* Lustre message size: 4096(payload) + 72(header) */
#define BLOCK                 1     /* 1 blocking / 0 non-blocking communication */


// Prototypes
int   CheckAndSendAcceptorRequest  ();
int   CheckAndSendLNETHello        ();
int   CheckAndSendMessage          ();
// Lustre prototypes
char * libcfs_nid2str              (lnet_nid_t nid);
int   libcfs_ip_str2addr           (char *str, int nob, __u32 *addr);
void  change_string                (lnet_nid_t *nid, char *str);
int   lnet_accept_magic            (__u32 magic, __u32 constant);

#endif

```

```

// -----------------------------------------------------------------------------
// End of file
// -----------------------------------------------------------------------------
```

A.1.7 Makefile

```
## Makefile to create the HA components for Lustre
## Written by Matthias Weber
##
## usage:
##  three targets to build:
##      interceptor_mds (default) (possible flag: CPPFLAGS+=-DTRANSIS_BYPASS)
##      interceptor_client (with flags: CPPFLAGS+=-DINTERCEPTOR_CLIENT
##                                      CPPFLAGS+=-DTRANSIS_BYPASS)
##      fake_mds (with flag CPPFLAGS+=-DFAKE_MDS)
##
##  additional option:
##      debug mode: CPPFLAGS+=-DDEBUG
##
##  for cleanup:
##      clean (deletes all object files and executables)
##      clean_objects (deletes all object files)
##      clean_debug_files (deletes the files created in debug mode)
##
##  CPPFLAGS:
##      DEBUG                       - enable debug mode
##      INTERCEPTOR_CLIENT          - switch ip addresses to client (use of MDS
##                                                             interceptor as MDS)
##      INTERCEPTOR_CLIENT_ALONE - switch ip addresses to client (use of Lustre
##                                                                  MDS directly)
##      FAKE_MDS                    - just work as transis client and don't use real MDS
##      TRANSIS_BYPASS              - no use of transis
##
## example:
##      make interceptor_client -e CPPFLAGS+=-DDEBUG CPPFLAGS+=-DINTERCEPTOR_CLIENT
##                                                   CPPFLAGS+=-DTRANSIS_BYPASS


## Compiler
CC = gcc

## Transis directory
BASEDIR=/usr/src/transis

## Transis include directories
INCLUDEDIR=$(BASEDIR)/include/
LIBDIR=$(BASEDIR)/bin/LINUX/

## Transis flags
TRANSISLIBS=-L$(LIBDIR) -ltransis

## Compiler flags
CFLAGS=-I$(INCLUDEDIR) -Wall

## lpthread flags
LPTHREAD=-lpthread

## the objects
OBJECTS = lustreHAdaemon.o lustreMessageAdjust.o transis.o

```

```
all: interceptor_mds

interceptor_mds: clean_objects $(OBJECTS)
	@echo "building Lustre MDS Interceptor..."
	@$(CC) -o lustre_MDS_Interceptor $(OBJECTS) $(TRANSISLIBS) $(LPTHREAD)
	@echo "done"

interceptor_client: clean_objects $(OBJECTS)
	@echo "building Lustre Client Interceptor..."
	@$(CC) -o lustre_CLIENT_Interceptor $(OBJECTS) $(TRANSISLIBS) $(LPTHREAD)
	@echo "done"

fake_mds: clean_objects $(OBJECTS)
	@echo "building Lustre Fake MDS..."
	@$(CC) -o lustre_Fake_MDS $(OBJECTS) $(TRANSISLIBS) $(LPTHREAD)
	@echo "done"

clean:
	@echo "cleaning all executables and object files..."
	@/bin/rm -f lustre_Fake_MDS lustre_CLIENT_Interceptor lustre_MDS_Interceptor *.o
	@echo "done"

clean_objects:
	@echo "cleaning object files..."
	@/bin/rm -f *.o
	@echo "done"

clean_debug_files:
	@echo "cleaning debug files..."
	@/bin/rm -f send* recv* TR*
	@echo "done"

%.o: %.c
	@echo "compiling file..."
	@$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@
	@echo "done"
```

A.2 Benchmark Program Source Code

A.2.1 benchmarkProgram.c

```
// ---------------------------------------------------------------------------
// Benchmark Programm for the
//     Lustre High Availability Daemon
//
// benchmarkProgram.c  --source file--
//
// version 1.0
//
// by Matthias Weber
// ---------------------------------------------------------------------------


#include "benchmarkProgram.h"


// Globals
__u64        NumberOfFiles;
__u64        NumberOfTests;
int          *FileDescriptorArray;
char         **FileNameArray;
time_data_t *timeData;


// ---------------------------------------------------------------------------
// sets up the needed values to perform the tests
//
// returns: 0 on success / -1 if error occurs
// ---------------------------------------------------------------------------
int Set_Up_Values ()
{
  __u64 i;
  char  fileNumber[20];

  /* allocate memory to hold results of test runs */
  timeData = (time_data_t *) malloc(NumberOfTests * sizeof(time_data_t));
  if(timeData == NULL)
    return -1;

  /* allocate memory to hold file name and descriptor */
  FileDescriptorArray = (int *) malloc(NumberOfFiles * sizeof(int));
  if(FileDescriptorArray == NULL)
    return -1;

  FileNameArray = (char **) malloc(NumberOfFiles * sizeof(char *));
  if(FileNameArray == NULL)
    return -1;

  for(i=0; i<NumberOfFiles; i++){
    FileNameArray[i] = (char *) malloc(30 * sizeof(char));
    if(FileNameArray[i] == NULL)
      return -1;
  }

  /* create file names */
  for(i=0; i<NumberOfFiles; i++){
    strcpy (&FileNameArray[i][0], "/mnt/lustre/LTEST");
```

```
    sprintf(fileNumber, "%lld", i);
    strcat (&FileNameArray[i][0], fileNumber);
  }

  return 0;
}


// ---------------------------------------------------------------
// creates the specified number of files and measures time needed
// to do so
//
// returns: 0 on success / -1 if error occurs
// ---------------------------------------------------------------
int Test_Open (__u64 run_number)
{
  __u64 i;
  int   rc;
  struct timezone tz;
  struct timeval  time_before;
  struct timeval  time_after;
  time_data_t   *time;

  time = &timeData[run_number];

  /* get time before test */
  rc = gettimeofday(&time_before, &tz);
  if(rc == -1)
    return -1;

  for(i=0; i<NumberOfFiles; i++){
    /* create file */
    FileDescriptorArray[i] = open(&FileNameArray[i][0],
                                  O_CREAT | O_TRUNC | O_RDWR, 0666 );
    if(FileDescriptorArray[i] == -1) {
      perror("open");
      return -1;
    }
    /* close file */
    rc = close(FileDescriptorArray[i]);
    if(rc == -1) {
      perror("close");
      return -1;
    }
  }

  /* get time after test */
  rc = gettimeofday(&time_after, &tz);
  if(rc == -1)
    return -1;

  /* get difference */
  time->open_usec = ((time_after.tv_sec*1000000) + time_after.tv_usec) -
                    ((time_before.tv_sec*1000000) + time_before.tv_usec);

  return 0;
}


// ---------------------------------------------------------------
// reades the file status (metadata) of the created files
// and measures time needed to do so
//
```

```c
// returns: 0 on success / -1 if error occurs
// -----------------------------------------------------------------
int Test_Stat (__u64 run_number)
{
  __u64   i;
  int     rc;
  struct stat      file_status;
  struct timezone tz;
  struct timeval   time_before;
  struct timeval   time_after;
  time_data_t      *time;

  time = &timeData[run_number];

  /* get time before test */
  rc = gettimeofday(&time_before, &tz);
  if(rc == -1)
    return -1;

  for(i=0; i<NumberOfFiles; i++){
    /* open file */
    FileDescriptorArray[i] = open(&FileNameArray[i][0], O_RDWR, 0666 );
    if(FileDescriptorArray[i] == -1) {
      perror("open");
      return -1;
    }
    /* read file */
    rc = fstat(FileDescriptorArray[i], &file_status);
    if(rc == -1) {
      perror("fstat");
      return -1;
    }
    /* close file */
    rc = close(FileDescriptorArray[i]);
    if(rc == -1){
      perror("close") ;
      return -1;
    }
  }

  /* get time after test */
  rc = gettimeofday(&time_after, &tz);
  if(rc == -1)
    return -1;

  /* get difference */
  time->read_usec = ((time_after.tv_sec*1000000) + time_after.tv_usec) -
                    ((time_before.tv_sec*1000000) + time_before.tv_usec);

  return 0;
}


// -----------------------------------------------------------------
// deletes the created files and measures time needed to do so
//
// returns: 0 on success / -1 if error occurs
// -----------------------------------------------------------------
int Test_Delete (__u64 run_number)
{
  __u64   i;
  int     rc;
  struct timezone tz;
```

```
  struct timeval   time_before;
  struct timeval   time_after;
  time_data_t      *time;

  time = &timeData[run_number];

  /* get time before test */
  rc = gettimeofday(&time_before, &tz);
  if(rc == -1)
    return -1;

  for(i=0; i<NumberOfFiles; i++){
    rc = unlink(&FileNameArray[i][0]);
    if(rc == -1) {
      perror("unlink");
      return -1;
    }
  }

  /* get time after test */
  rc = gettimeofday(&time_after, &tz);
  if(rc == -1)
    return -1;

  /* get difference */
  time->delete_usec = ((time_after.tv_sec*1000000) + time_after.tv_usec) -
                      ((time_before.tv_sec*1000000) + time_before.tv_usec);

  return 0;
}


// -------------------------------------------------------------------
// Prints the result of the benchmark test on the screen
// -------------------------------------------------------------------
void Print_Test_Results ()
{
  __u64  i;
  double open_time   = 0;
  double read_time   = 0;
  double delete_time = 0;
  double open_Temp   = 0;
  double read_Temp   = 0;
  double delete_Temp = 0;
  double open_Operations;
  double read_Operations;
  double delete_Operations;
  double open_StandardDeviation;
  double read_StandardDeviation;
  double delete_StandardDeviation;

  time_data_t *time;

  /* add up time */
  for(i=0; i<NumberOfTests; i++){
    time = &timeData[i];

    open_time   += time->open_usec;
    read_time   += time->read_usec;
    delete_time += time->delete_usec;
  }

  /* calculate mean value */
```

```
  open_time    = open_time   / (double) NumberOfTests;
  read_time    = read_time   / (double) NumberOfTests;
  delete_time = delete_time / (double) NumberOfTests;

  /* print mean value */
  printf("--     Mean Time taken for Operations     --\n");
  printf("- Time taken for create: %12.3lf usec -\n", open_time);
  printf("- Time taken for read:   %12.3lf usec -\n", read_time);
  printf("- Time taken for delete: %12.3lf usec -\n", delete_time);
  printf("---------------------------------------------\n\n");

  /* calculate performed operations per sec */
  open_Operations   = (double) NumberOfFiles / (open_time   / 1000000.0);
  read_Operations   = (double) NumberOfFiles / (read_time   / 1000000.0);
  delete_Operations = (double) NumberOfFiles / (delete_time / 1000000.0);

  /* print operations per sec */
  printf("-- Operations per second --\n");
  printf("- create: %10.3lf /sec -\n",   open_Operations);
  printf("- read:   %10.3lf /sec -\n",   read_Operations);
  printf("- delete: %10.3lf /sec -\n", delete_Operations);
  printf("---------------------------\n\n");

  /* print time needed for one operation */
  printf("--        Mean Time        --\n");
  printf("--    for one Operation    --\n");
  printf("- create: %10.3lf msec -\n",   open_time/((double)NumberOfFiles*1000.0));
  printf("- read:   %10.3lf msec -\n",   read_time/((double)NumberOfFiles*1000.0));
  printf("- delete: %10.3lf msec -\n", delete_time/((double)NumberOfFiles*1000.0));
  printf("---------------------------\n\n");

  /* calculate standard deviation */
  for(i=0; i<NumberOfTests; i++){
    time = &timeData[i];

    open_Temp   += pow(time->open_usec - open_time, 2.0);
    read_Temp   += pow(time->read_usec - read_time, 2.0);
    delete_Temp += pow(time->delete_usec - delete_time, 2.0);
  }
  open_StandardDeviation   = sqrt(open_Temp   /(NumberOfTests-1));
  read_StandardDeviation   = sqrt(read_Temp   /(NumberOfTests-1));
  delete_StandardDeviation = sqrt(delete_Temp/(NumberOfTests-1));

  /* print standard deviation */
  printf("-- Standard Deviation --\n");
  printf("--    of Test Series   --\n");
  printf("- create: %12.3lf -\n", open_StandardDeviation);
  printf("- read:   %12.3lf -\n", read_StandardDeviation);
  printf("- delete: %12.3lf -\n", delete_StandardDeviation);
  printf("------------------------\n\n");
}


// ---------------------------------------------------------------------------
// Runns one test series
//
// returns: 0 on success / -1 if error occurs
// ---------------------------------------------------------------------------
int Run_One_Test (__u64 run_number)
{
  int rc;

  rc = Test_Open(run_number); /* create files */
```

```c
  if(rc == -1) {
    fprintf(stderr, "error, creating files\n");
    return -1;
  }

  rc = Test_Stat(run_number); /* read metadata */
  if(rc == -1) {
    fprintf(stderr, "error, reading metadata\n");
    return -1;
  }

  rc = Test_Delete(run_number); /* delete files */
  if(rc == -1) {
    fprintf(stderr, "error, deleting files\n");
    return -1;
  }

  return 0;
}


// ----------------------------------------------------------------------
// Application main entry point
// ----------------------------------------------------------------------
int main ( int argc, char *argv[] )
{
  __u64 i;
  int   rc;

  /* check for parameters */
  if(argc != 3){
    printf("usage:  benchmark number_of_files number_of_tests\n");
    printf("example: benchmark 1024 1\n");
    exit(-1);
  }

  NumberOfFiles = atoll(argv[1]);
  NumberOfTests = atoll(argv[2]);

  printf("Number of files to use for testing: %lld\n", NumberOfFiles);
  printf("Number of tests to run: %lld\n", NumberOfTests);

  /* set up values */
  printf("setting up values... ");
  rc = Set_Up_Values();
  if(rc == -1){
    fprintf(stderr, "error Set_Up_Values\n");
    free(FileDescriptorArray);
    free(FileNameArray);
    exit(-1);
  }
  printf("done\n");

  /* run test series */
  printf("doing test runns...\n");
  for(i=0; i<NumberOfTests; i++){
    rc = Run_One_Test(i);
    if(rc == -1){
      fprintf(stderr, "error in test run %lld\n", i);
      free(FileDescriptorArray);
      free(FileNameArray);
      exit(-1);
    }
```

```
    printf(".");
  }
  printf("\ndone\n");

  /* print results */
  printf("\nTest Results:\n\n");
  Print_Test_Results();

  /* free memory and exit */
  free(FileDescriptorArray);
  free(FileNameArray);
  exit(1);
}


// ------------------------------------------------------------------------
// End of file
// ------------------------------------------------------------------------
```

A.2.2 benchmarkProgram.h

```
// ------------------------------------------------------------------------
// Benchmark Programm for the
//     Lustre High Availability Daemon
//
// benchmarkProgram.h  --header file--
//
// version 1.0
//
// by Matthias Weber
// ------------------------------------------------------------------------


// Includes
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <math.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>


// Defines
typedef unsigned long       __u32;
typedef unsigned long long  __u64;
typedef struct {
  __u64 open_usec;
  __u64 read_usec;
  __u64 delete_usec;
} time_data_t;


// Prototypes
int  Set_Up_Values       ();
int  Test_Open           (__u64 run_number);
int  Test_Stat           (__u64 run_number);
int  Test_Delete         (__u64 run_number);
int  Run_One_Test        (__u64 run_number);
```

```
void Print_Test_Results ();


// ---------------------------------------------------------------------------
// End of file
// ---------------------------------------------------------------------------
```

A.3 Lustre XML Config File


```xml
<?xml version='1.0' encoding='UTF-8'?>
<lustre version='2003070801' mtime='1169142788'>
  <ldlm name='ldlm' uuid='ldlm_UUID'/>
  <node uuid='mds1_UUID' name='mds1'>
    <profile_ref uuidref='PROFILE_mds1_UUID'/>
    <network uuid='NET_mds1_tcp_UUID' nettype='tcp' name='NET_mds1_tcp'>
      <nid>mds1</nid>
      <clusterid>0</clusterid>
      <port>988</port>
    </network>
  </node>
  <profile uuid='PROFILE_mds1_UUID' name='PROFILE_mds1'>
    <ldlm_ref uuidref='ldlm_UUID'/>
    <network_ref uuidref='NET_mds1_tcp_UUID'/>
    <mdsdev_ref uuidref='MDD_mds1_mds1_UUID'/>
  </profile>
  <node uuid='ost1_UUID' name='ost1'>
    <profile_ref uuidref='PROFILE_ost1_UUID'/>
    <network uuid='NET_ost1_tcp_UUID' nettype='tcp' name='NET_ost1_tcp'>
      <nid>ost1</nid>
      <clusterid>0</clusterid>
      <port>988</port>
    </network>
  </node>
  <profile uuid='PROFILE_ost1_UUID' name='PROFILE_ost1'>
    <ldlm_ref uuidref='ldlm_UUID'/>
    <network_ref uuidref='NET_ost1_tcp_UUID'/>
    <osd_ref uuidref='OSD_ost1_ost1_UUID'/>
    <osd_ref uuidref='OSD_ost2_ost1_UUID'/>
  </profile>
  <node uuid='usr1_UUID' name='usr1'>
    <profile_ref uuidref='PROFILE_usr1_UUID'/>
    <network uuid='NET_usr1_tcp_UUID' nettype='tcp' name='NET_usr1_tcp'>
      <nid>usr1</nid>
      <clusterid>0</clusterid>
      <port>988</port>
    </network>
  </node>
  <profile uuid='PROFILE_usr1_UUID' name='PROFILE_usr1'>
    <ldlm_ref uuidref='ldlm_UUID'/>
    <network_ref uuidref='NET_usr1_tcp_UUID'/>
    <mountpoint_ref uuidref='MNT_usr1_UUID'/>
  </profile>
  <mds uuid='mds1_UUID_2' name='mds1'>
    <active_ref uuidref='MDD_mds1_mds1_UUID'/>
    <lovconfig_ref uuidref='LVCFG_lov1_UUID'/>
    <filesystem_ref uuidref='FS_fsname_UUID'/>
  </mds>
  <mdsdev uuid='MDD_mds1_mds1_UUID' name='MDD_mds1_mds1'>
    <fstype>ldiskfs</fstype>
    <devpath>/lustretest/mds-mds1</devpath>
    <autoformat>no</autoformat>
    <devsize>500000</devsize>
    <journalsize>0</journalsize>
    <inodesize>0</inodesize>
    <node_ref uuidref='mds1_UUID'/>
    <target_ref uuidref='mds1_UUID_2'/>
  </mdsdev>
  <lov stripesize='1048576' stripecount='0' stripepattern='0'
                                               uuid='lov1_UUID' name='lov1'>
```

```
    <mds_ref uuidref='mds1_UUID_2'/>
    <obd_ref uuidref='ost1_UUID_2'/>
    <obd_ref uuidref='ost2_UUID'/>
  </lov>
  <lovconfig uuid='LVCFG_lov1_UUID' name='LVCFG_lov1'>
    <lov_ref uuidref='lov1_UUID'/>
  </lovconfig>
  <ost uuid='ost1_UUID_2' name='ost1'>
    <active_ref uuidref='OSD_ost1_ost1_UUID'/>
  </ost>
  <osd osdtype='obdfilter' uuid='OSD_ost1_ost1_UUID' name='OSD_ost1_ost1'>
    <target_ref uuidref='ost1_UUID_2'/>
    <node_ref uuidref='ost1_UUID'/>
    <fstype>ldiskfs</fstype>
    <devpath>/lustretest/ost1</devpath>
    <autoformat>no</autoformat>
    <devsize>1000000</devsize>
    <journalsize>0</journalsize>
    <inodesize>0</inodesize>
  </osd>
  <ost uuid='ost2_UUID' name='ost2'>
    <active_ref uuidref='OSD_ost2_ost1_UUID'/>
  </ost>
  <osd osdtype='obdfilter' uuid='OSD_ost2_ost1_UUID' name='OSD_ost2_ost1'>
    <target_ref uuidref='ost2_UUID'/>
    <node_ref uuidref='ost1_UUID'/>
    <fstype>ldiskfs</fstype>
    <devpath>/lustretest/ost2</devpath>
    <autoformat>no</autoformat>
    <devsize>1000000</devsize>
    <journalsize>0</journalsize>
    <inodesize>0</inodesize>
  </osd>
  <filesystem uuid='FS_fsname_UUID' name='FS_fsname'>
    <mds_ref uuidref='mds1_UUID_2'/>
    <obd_ref uuidref='lov1_UUID'/>
  </filesystem>
  <mountpoint uuid='MNT_usr1_UUID' name='MNT_usr1'>
    <filesystem_ref uuidref='FS_fsname_UUID'/>
    <path>/mnt/lustre</path>
  </mountpoint>
</lustre>
```

A.4 User Manuals

A.4.1 Benchmark Program

The benchmark program can be build easily from the sources, provided in Section A.2, with the following command:

```
gcc -lm -o benchmarkProgram benchmarkProgram.c
```

The use of the program is straightforward. The program needs two parameters to determine how the test run should be performed. The first parameter gives the number of files to use for one test run. The second parameter tells the program how many test runs to perform.

A command for an example test may look like this:

```
./benchmarkProgram 1024 10
```

The program always uses the `/mnt/lustre/` directory for testing. The above given command starts the benchmark program. It performs one test run with three individual tests. The program creates, reads the metadata of, and deletes 1024 files in the mentioned directory. The times needed to perform each of the tests are taken.

The second parameter tells the program to repeat this test run 10 times. After all test runs are completed, the mean time needed to perform one test is calculated from all test runs. Also the standard derivation of the test series is calculated in order to evaluate the error of the test.

The result of the example test is given below:

```
Number of files to use for testing: 1024
Number of tests to run: 10
setting up values... done
doing test runns...
..........
done
```

```
Test Results:

--     Mean Time taken for Operations     --
- Time taken for create:    46457.700 usec -
- Time taken for read:       2213.200 usec -
- Time taken for delete:     4732.100 usec -
--------------------------------------------

-- Operations per second --
- create:  22041.556 /sec -
- read:   462678.475 /sec -
- delete: 216394.413 /sec -
---------------------------

--       Mean Time       --
--   for one Operation   --
- create:      0.045 msec -
- read:        0.002 msec -
- delete:      0.005 msec -
---------------------------

-- Standard Deviation --
--   of Test Series   --
- create:    33795.633 -
- read:         90.385 -
- delete:      213.347 -
------------------------
```

A.4.2 Lustre HA Prototypes

Due to the lack of complete HA functionality a user manual cannot be provided for the prototypes. What is described in this section is how to setup the machines in order to replicate the results of this project.

First step is to setup a network with five nodes. All nodes need to run Fedora Core 4 as operating system.

Lustre needs to be installed on all nodes. The test runs in the project have been done with Lustre version 1.4.8, build from source against a prepatched kernel provided by Lustre. The two following source packages of Lustre version 1.4.8 for the Red Hat kernel 2.6 include the needed data and can be downloaded from Lustre[1].

The prepatched kernel source package:
`kernel-source-2.6.9-42.0.3.EL_lustre.1.4.8.i686.rpm`
The Lustre source package:
`lustre-source-1.4.8-2.6.9_42.0.3.EL_lustre.1.4.8smp.i686.rpm`

The installed source trees can be found in the following directory:
`/usr/src/`

Now, the kernel source tree needs to be configured and installed. The following commands must be performed in the kernel source directory.

clean the source tree:
`make distclean`

copy config file into source tree:
`` cp /boot/config-`uname -r` .config ``

configure the kernel:
`make oldconfig || make menuconfig`

build the kernel and install the kernel modules:
`make oldconfig dep bzImage modules modules_install install`

[1] Lustre download: `http://www.clusterfs.com/download.html`

modify the boot menu in order to reboot with the new kernel:
`vi /boot/grub/menu.lst`

Now, the machine needs to be rebooted with the new kernel.

After this step Lustre can be built. This is done with the following two commands called from the Lustre source directory:

```
./configure -with-linux=/your/patched/kernel/sources
make rpms
```

If run successfully, Lustre builds rpm packages and places them in the following directory:
`/usr/src/redhat/RPMS/i386/`

To install Lustre on the system, two packages from this directory need to be installed. The Lustre package itself and the Lustre kernel modules.

After the installation of Lustre the prototypes must be built. This can be done with the source code and the makefile provided in Section A.1. How to build the different components required for the tests is described in the makefile.

Figure 3.9 gives and overview of the needed prototype components and the network address setup. On the client (`USR1`) and the first MDS (`MDS1`) node IP aliasing must be used to establish the two IP addresses.

Lustre must be configured with help three XML files. One XML file for each component of the file system. How to create and configure these XML files is described in Section 3.1.

For proper functionality of the prototypes the group communication system Transis needs to be downloaded[2] and built from source. This can be easily done with the `make` command called in the source directory.

Transis needs to know the addresses of all possible group members. A plain text file called `config`, only including all IP addresses of the interceptors of the MDS group must be created in the directory of the Transis daemon executable.

Now, all components needed are installed and configured. Last thing to do, in order to

[2]Transis download: `http://www.cs.huji.ac.il/labs/transis/software.html`

replicate the results, is to start the test setup. This process requires several steps.

Fist, the Transis daemon has to be started on all relevant nodes.

Then all for the test required prototype components need to be started. This is done by just starting the built executable.

Last, Lustre can be started. This is done in three steps. Therefore, the following commands have to be performed on the respective nodes in the directory in which the XML file lies.

First, the OSTs are started:
`lconf -reformat -node ost config_OST.xml`

Then, the MDS is started:
`lconf -reformat -node mds config_MDS.xml`

At last, the client can be started:
`lconf -node usr config_USR.xml`

If no errors occur, the test setup is up and running. To use the file system or to perform tests, the benchmark program described in Section A.4.1 can be used.

In order to shutdown Lustre, the following commands must be used on the respective nodes in the given order.

First the OSTs are stopped:
`lconf -cleanup -node ost config_OST.xml`

Then the MDS is shutdown:
`lconf -cleanup -node mds config_MDS.xml`

Last, the client is unmounted:
`lconf -cleanup -node usr config_USR.xml`

After Lustre has exited, the prototype components and the Transis daemon can be stopped.

List of Figures

List of Tables