Middleware for Transparent TCP Connection Migration

Masking Faulty TCP-based Services

by

Håvard Bakke and Torbjørn Meland

Masters Thesis in
Information and Communication Technology

Agder University College

Grimstad, June 2004

Abstract

Mission critical TCP-based services create a demand for robust and fault tolerant TCP communication. Sense Intellifield monitors drill operations on rig sites offshore. Critical TCP-based services need to be available 24 hours, 7 days a week, and the service providers need to tolerate server failure.

How to make TCP robust and fault tolerant without modifying existing infrastructure like existing client/server applications, services, TCP stacks, kernels, or operating systems is the motivation of this thesis.

We present a new middleware approach, first of its kind, to allow TCP-based services to survive server failure by migrating TCP connections from failed servers to replicated surviving servers. The approach is based on a proxy technique, which requires modifications to existing infrastructure. Our unique middleware approach is simple, practical, and can be built into existing infrastructure without modifying it. A middleware approach has never been used to implement the proxy based technique.

Experiments for validation of functionality and measurement of performance of the middleware prototype are conducted. The results show that our technique adds significant robustness and fault tolerance to TCP, without modifying existing infrastructure.

One of the consequences of using a middleware to make TCP communication robust and fault tolerant is added latency. Another consequence is that TCP communication can survive server failure, and mask it. Companies providing robust and fault tolerant TCP, is no longer dependant of third party hardware and/or software. By implementing our solution, they can gain economical advantages.

A main focus of this report is to present a prototype that demonstrates our technique and middleware approach. We present relevant background theory which has lead to the design architecture of a middleware approach to make TCP communication fault tolerant. Finally we conduct experiments to uncover the feasibility and performance of the prototype, followed by a discussion and conclusion.

Index Terms: Network reliability, TCP migration, Continuity of service, Middleware, Fault tolerance and failure masking.

Preface

This thesis is written for Sense Intellifield and is the final part of the Master of Science degree in Information and Communication Technology, also known as “Sivilingeniør i IKT” in Norway. The work has been carried out in the period between January and May 2004.

The thesis definition is based on a practical problem provided by Sense Intellifield which we try to solve with a theoretical approach.

The thesis is based on theory from the *Distributed Systems* courses IKT2335 and IKT2340, lectured by *cand. scient.* Ole-Christoffer Granmo, at Agder University College. Ole-Christoffer Granmo is also our theoretical supervisor.

Sense Intellifield, http://www.sensetech.no, is a Norwegian company that sells IT-solutions for the petroleum industry. They are sited in Kristiansand, Norway. Their main focus is to monitor drill and rig sites onshore and offshore.

Our practical/technical supervisor is the Vice President of Technology and Product Development Rune Skarbø, at Sense Intellifield.

Our gratitude to our supervisors and contributors is expressed in the *Acknowledgments* section at the end of this report.

Grimstad, May 2004

Håvard Bakke and Torbjørn Meland

Table of contents

Table of figures

1 Introduction

In this chapter we will give an introduction to how a middleware can be used to add significant robustness and fault tolerance to TCP. In section 1.1 we give a brief overview of the need for robust and fault tolerant TCP. In section 1.2 we present a real case where there is a lack of robustness and fault tolerance, including a description of requirements to possible solutions. In section 1.3 we give an overview of related work to add robustness and fault tolerance to TCP-based services. In section 1.4 we outline the final thesis definition to the problem we will solve. In section 1.5 we give an overview of our work, followed by section 1.6 which we give an outline of this report.

1.1 Background

Businesses around the globe are becoming more dependent of network services than ever before. Several TCP/IP-based services are mission-critical for these businesses and the safety of their employees. The need for continuous information flow and "round a clock" operation has arisen. High availability, by failover protection of TCP-based services, is the key issue to address continuous information flow and "round a clock" operation.

Today, numerous protocols for data communication like the Transmission Control Protocol (TCP) ensures reliable and efficient client to server communication [1]. TCP ensures delivery of packets unless a client, server, or network fails.

To prevent that a client, server, or network becomes a single point of failure, replication and redundancy is used.

Sense Intellifield has developed a system, SiteCom® Rig, for acquiring, distributing and managing rig site data. This involves transferring real-time and historical data between onshore and offshore drill- and well sites, onshore control centers, headquarter offices, remote export sites, etc.

The information flow is mission-critical for the safety and efficiency of drilling operations. It is therefore a high demand for the reliability and availability of the SiteCom® Rig services.

1.2 Sense Intellifield's service architecture

Sense Intellifield monitors several critical components during drilling operations. One of Sense's products, SiteCom® Rig, is used at well sites either onshore or offshore. Various drilling systems push their data to the SiteCom® Rig system which in turn stores data and distributes data in real-time to any number of visualization and monitoring applications. A separate alarm system is deployed to acquire information from the well site SiteCom® Rig system and analyze the data. If any abnormality is detected, alarms are generated and sent to emergency personnel. This information flow is critical for the safety and efficiency of drilling operations.

Fault tolerance is a key ingredient for continuous (24/7) operation and to ensure high availability for continuous information flow. Sense Intellifield has ensured fault tolerance by using load balancing and server redundancy. The SiteCom® Rig system is replicated and a load balancer redirects incoming requests. The architecture is shown in the following figure:

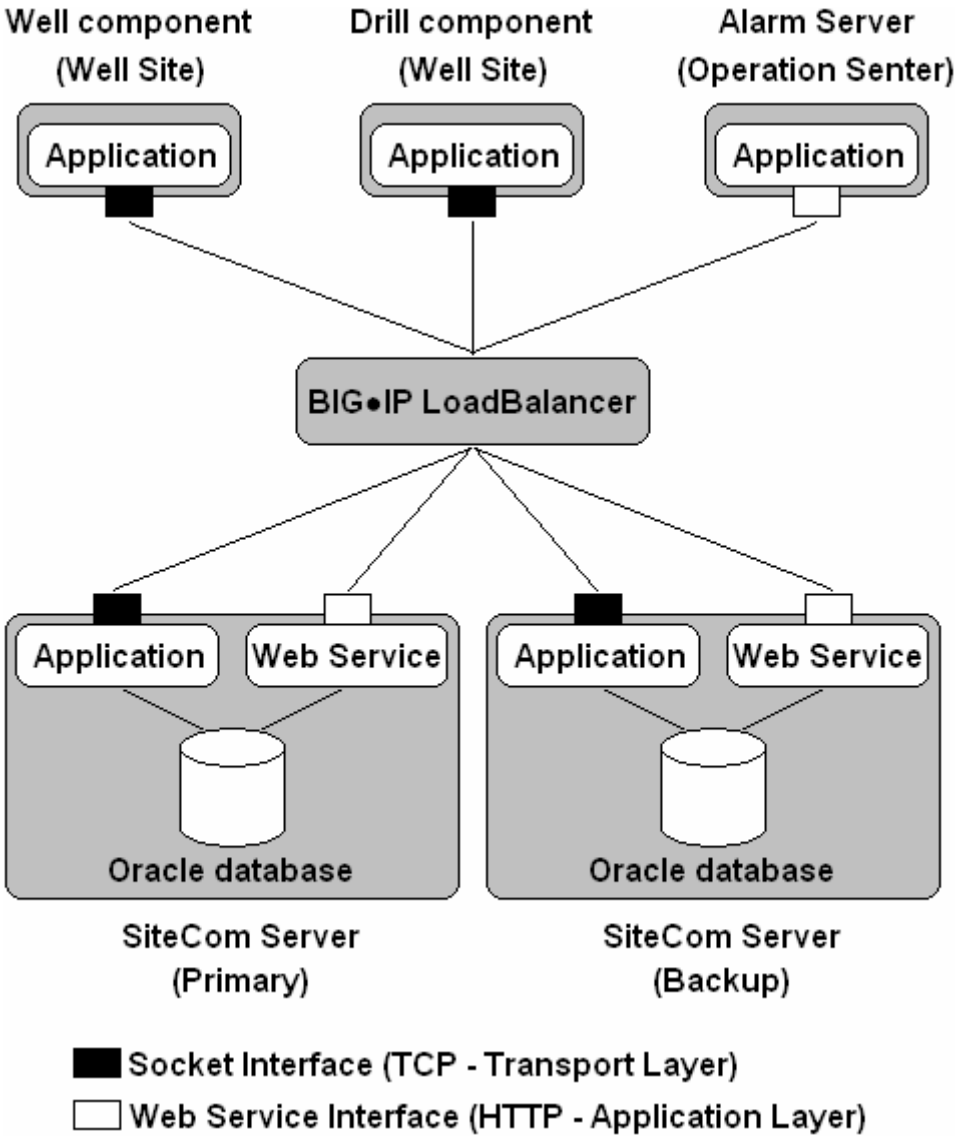

Figure 1: Sense's SiteCom® architecture

If a server halts due to failure, incoming requests are redirected to the backup SiteCom® Rig server. Established connections to the faulty server are broken and clients have to reconnect to the backup. The SiteCom® Rig servers are unavailable for a few seconds, due to the time the load balancer uses to detect failure and the time the backup server currently requires to initialize the services the primary server was providing. During this downtime, some information may be lost, but this loss is not critical.

The SiteCom® Rig server is crash-failure tolerant by redundancy and provides high availability to its TCP/IP based services. The solution is *N*-1 crash-failure tolerant, where *N* is the number of replicated server nodes. The load balancer is very reliable, but is still a single point of failure. It has the option of being replicated as well, ruling out the single point of failure.

However one of the important problems with the current architecture is that this solution is very expensive and it does not mask faults completely. If a server node fails, connections break. The client connected will detect the failure and may attempt to reconnect, but not all clients have an auto-reconnect feature. Some clients might fail due to a broken connection. This architecture do not migrate existing TCP connections to surviving replicated server nodes, and is therefore unable to mask failure.

Sense Intellifield monitor several drill sites and has to introduce a SiteCom® Rig system, as seen in Figure 1, at each rig. The solution used today illustrated in Figure 1 is too expensive. The F5 load balancer has a marked value of about 100000 NOK[1], and it has to come in pairs to rule out the load balancer as a single point of failure.

Sense has developed their services and server applications with the Microsoft .Net service architecture, which only runs on the Microsoft Windows operating system. To use this operating system, licenses have to be acquired. To limit the costs of acquiring expensive licenses, Sense Intellifield has decided to limit the number of servers to the minimum. Two are needed to rule out a single point of failure.

Sense Intellifield wants a solution consisting of two servers running on Microsoft Windows operating system. The servers have to be connected to standard network node like a hub or a switch to cut expenses and should not be dependant of third party load balancers or "black boxes". There is a need for a solution that does not require modifications to operating systems.

1.3 Related work

> *Our primary motivation is to provide a tool that enhances reliability, which can be simply attached to existing infrastructure without making any modifications to the server or the client. In addition, no modifications shall be applied to any operating systems, TCP/IP stacks, applications, services, or kernels.*

Today, there are several theoretical, commercial, and open source approaches to enhance the reliability of TCP and the availability of TCP-based services. Generally, the proposed solutions are tailored for specific purposes and they are not general. Most of the solutions are dependant of specific operating systems, third party hardware, or tailored applications.

There are many solutions that offer efficient TCP connection failover in clusters [47]. These solutions generally organize servers in a front-end and backend. The front-end

[1] Marked value and currency per 20 May 2004

usually is a server functioning as a coordinator. The backend is a server cluster consisting of replicated services. The coordinator distributes incoming requests to the cluster. If a server in the cluster fails, the coordinator migrate connections to surviving servers in the cluster. The failure is completely transparent to the clients, by introducing a new transport layer. By using a coordinator, one is vulnerable to single point of failure. This solution requires at least three servers, and server side applications and services have to be modified. This solution does not comply with the requirements of not modifying existing infrastructure, and having two servers as the maximum amount of machines.

A similar approach is used by fine-grained failover using connection migration [48], complies with the requirement of only having two servers, ensuring fault-tolerant and robust to TCP. The difference in this approach is that both servers have the same IP address and uses no coordinator, ruling out a single point of failure. The drawback is that the solution requires modifications to kernels in the operating system.

A solution complying with the requirement of only having two servers, ensuring fault-tolerant and robust to TCP, is called ST-TCP (Server fault-Tolerant TCP) [49]. The solution is based on an active backup server taping the Ethernet for information about the primary server's TCP state. This solution requires modification of the server side applications and kernels.

A different solution addresses link failures by adding a session layer protocol on top of TCP [50]. The solution is interesting due to its platform independence, link failure handling and low overhead (adding only 10% latency). If a link is broken, the session layer prevents the TCP connections from timing out. If the link is reestablished, the connection persists. The drawback is that this solution does not handle server failure.

There are not many theoretical solutions today that enhance the reliability and availability of TCP-based services without adding expensive hardware, and modifying client/server applications, operating systems, TCP/IP stacks, or kernels. Extensive search has currently revealed only two papers offering a solution to this problem; Jeebs and its extension Secure Jeebs [39, 40].

Jeebs is a client-transparent TCP migration system. It offers high availability by migrating connections from a failed server to a backup server. The Jeebs implementation consists of placing a "black box" on the server's subnet to monitor all TCP connections for the specified server host and services, detect loss of service, and recover TCP connection before clients' TCP stacks are aware of any difficulty. The system recovers from all combinations of Linux/Windows clients/servers, and has demonstrated seamless operation across server failures of many services, including HTTP, Telnet, SMTP and FTP. Jeebs can invoke a system call to place TCP state information (not application state) onto pristine sockets during connection recovery [39]. Note that this requires kernel modifications on the recovery system, ruling out this solution as applicable to Microsoft Windows or Mac operating systems. The Jeebs system requires the use of three servers; a primary server and a backup server, and a monitor/recovery server referred to as the "black box".

Secure Jeebs is a more advanced version of Jeebs. Secure Jeebs is based on Jeebs's client-transparent TCP migration techniques. It offers a migration system which migrate secure TCP-based services like SSH and HTTPS [40].

There are several commercial and open source solutions that enhance the reliability of TCP and the availability of TCP-based services. The solutions presented are mainly load balancers or modified servers enhancing reliable TCP.

Load balancing is a service that distributes load to a cluster of servers. This service is provided either in a hierarchical group or in a flat group (see chapter 3.2.1 for further details). Load balancing also provides higher availability by forwarding incoming request to available servers and services. If a server fails, connections are lost. But if clients try to reconnect, they will be forwarded to another available server in the cluster. These solutions do not provide connection migration transparent to the client, but ensures successful reconnection.

The BIG-IP® LoadBalancer Limited 520 is an example of a hierarchical load balancer, which is widely used in the commercial sector today [45]. Sense Intellifield is an example of a company that use a this kind of load balancer to increase the availability of their services.

Microsoft 2000 Server and Microsoft 2003 provides Network load balancing in their operating systems [46]. The servers in this solution are organized in flat groups (see chapter 3.2.1 for further details). The availability of web services is increased by distributing incoming request between the servers in the cluster. The members of the cluster operate on two IP addresses; one unique and one common virtual IP for the entire cluster. But if a server fails, handling a TCP connection, the failure is not transparent to the client. The client has to issue a reconnect to continue. Only HTTP failure is transparent. A single point of failure is detected in the documentation of the MS load balancer. A synchronization process is executed from a synchronization disc which is passed around as a token between the servers in the cluster. If the server containing the synchronization disc (token) crashes, the whole cluster fails.

The modified server solutions include modifications of existing open source operating systems, which are not applicable with Microsoft Windows, and Mac operating systems and applications. These solutions are platform dependant and not general. The High-Availability Linux Project is an example [44].

1.4 Thesis definition

The thesis provides a design for a fault tolerant, failure masking and replicated middleware for (N-1)-way failover in a cluster with N replicated server nodes. The middleware will focus on migrating open connections from a failed server node to surviving server nodes without breaking the connections with the clients. The final definition of the project is:

The students will design a middleware for migrating TCP connections, from a failing primary server to a replicated backup server, without breaking the connections. The migration will be completely transparent to the client and thus the failure is masked.

The thesis title is:

Middleware for Transparent TCP Connection Migration

The thesis subtitle is:

Masking Faulty TCP-based Services.

The following requirement is given:

The IP-stack must not be modified and there should be no modifications to the client or server application.

And finally:

If possible, a prototype will be made, demonstrating some of the concepts involved.

1.5 Our work

We will present a new concept of a middleware approach to enhance the availability and robustness of TCP-based services. Our concept is unique because it does not require any modifications to existing infrastructure. This means that no operating system, client/server applications, services, TCP/IP stacks, or kernels has to be modified. The middleware can simply be introduced to an existing architecture and increase the robustness and availability of TCP-based services.

Our concept is based on an existing proxy technique [39]. The technique was introduced in a system called Jeebs, but it was not implemented as a middleware and it required modifications to Linux kernels. In other words, the proxy technique requires modifications to existing infrastructure. A middleware approach implementing the proxy technique has never been done.

We also present a prototype solution for fault tolerant TCP, which demonstrates our concept. The solution is a middleware application, installed as a *black box* on the backup server on the local area network (LAN).

The prototype has to be true to the internet philosophy, where the intelligence is supposed to be implemented in the computers, while keeping the network nodes as dumb as possible. We want to regard the prototype as a network node, even though it is a middleware running on a backup server. Therefore it is important to let the TCP/IP stack take care of the communication as much as possible. This way we can rely on the transmission control protocol to provide reliable client to server communication. The

prototype will add robustness to TCP by increasing the availability of the TCP-based services residing on the server.

If the concept is realized by the prototype it could in practice, or at least in principle, make Sense Intellifield's SiteCom® Rig system less dependant of third party load balancer, as seen in Figure 1.

The intention is that if our prototype is introduced to the SiteCom® Rig system, the third party BIG-IP® LoadBalancer Limited 520 can replaced by a pair of hubs or switches with tapping capability. The prototype can simply be installed on the backup server, as seen below in Figure 2.

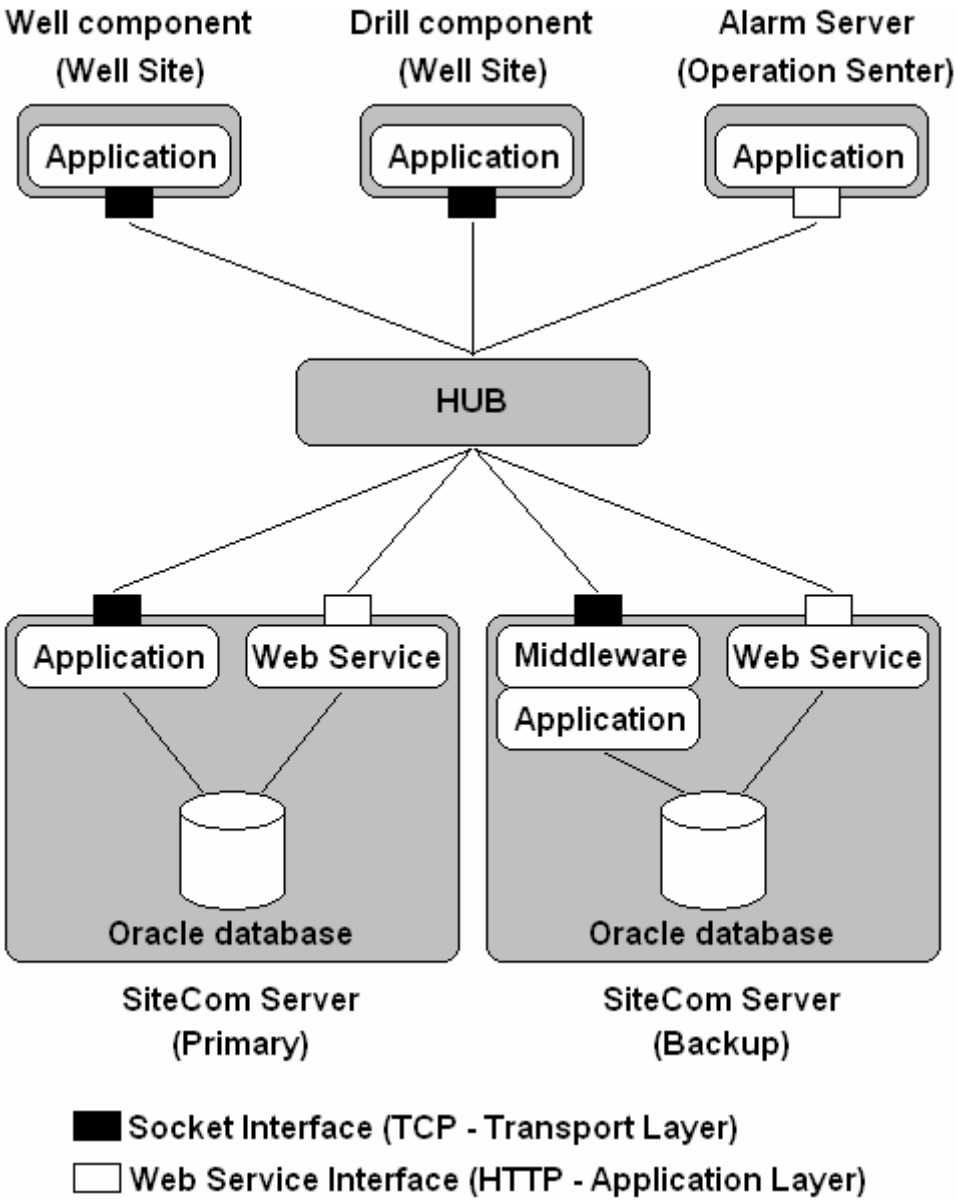

Figure 2: Supplement Sense's SiteCom® architecture with a middleware

The intention of our solution is to simply install a middleware to existing infrastructure without being dependant of expensive third party devices, or being dependant to specific operating systems. The concept of our middleware prototype is to simply install a platform independent middleware to enhance the reliability, robustness and availability of TCP and TCP-based services.

1.6 Report outline

The outline of this report is as follows; Chapter 2 contains a “need to know” presentation of the Transmission Control Protocol. It presents the basic of TCP, allowing the reader to keep track of the terminologies in the report. In chapter 3 and 4 we present relevant theory about replication and fault tolerance. Keep in mind that in chapter 2, 3, and 4 we give a brief overview of theory to give the reader a foundation to comprehend the contents of chapter 5. In chapter 5 we describe the architecture of the middleware prototype for transparent TCP connection migration which masks faulty TCP-based services. The modules and algorithms are described in this chapter. In chapter 6 we present the results of conducted experiments on the prototype, followed by a discussion of the results in chapter 7. The conclusion of the thesis is presented in the last and final chapter 8. The source code for the prototype is presented in *Appendix A* [51].

2 The Transmission Control Protocol

In this chapter we give a general overview of the transmission control protocol [1]. In chapter 5 an algorithm, translating sequence and acknowledgement numbers from a TCP connection to another, is presented. In this chapter we try to give the reader tools to comprehend the architecture of the prototype presented in this report. The literature in this chapter is a summary of the TCP RFC [1] and general computer engineering curriculum [36].

2.1 Introduction

This chapter will give an overview of the Transmission Control Protocol (TCP), and a more in depth view of the following issues; Connection establishment, connection termination, sequence and acknowledgement numbers, and the sliding window protocol. These subjects are especially important to understand this report.

The TCP protocol is on top of the IP protocol and below the application protocols, on the transport layer, seen in Figure 3. The TCP protocol has several purposes; creating process-to-process communication, creating flow and error control, and managing the connection setup and termination mechanisms [36].

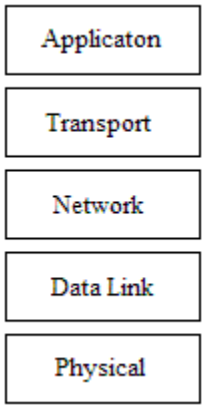

Figure 3: ISO 5 layer model

Each process is uniquely identified by the IP address and a port number. Where the IP address identifies a specific host on the internet, and the port number identifies a specific process on a host. This makes it possible to map a TCP connection to an application process.

The flow and error control is achieved through a combination of acknowledge packets, time-out and retransmission.

2.2 The TCP/IP data units

A data unit, when we are talking about the Internet Protocol (IP), is called a packet. When two hosts communicate over the internet, using the IP protocol, they need an IP

address each. The address is assigned by their internet service provider. The data sent over the internet will then be divided into small packets of data and a header is added. This header contains information which the routers on the internet use to forward the packets to the right destination. The header also contains a checksum and other information. The checksum is an ones complement checksum which functions as a simple error detection mechanism for the IP packets.

The IP packets are not ordered, and this creates a problem when you want to send data over the internet with the IP protocol, because the packets can arrive in a different order than they were sent in. This is where the TCP protocol comes in. TCP creates a virtual connection over the IP protocol using sequence and acknowledgment numbers. In addition TCP contains port numbers which makes it possible to identify the two communicating processes on the hosts. The data unit used by the TCP protocol is called a segment. The segment has a header containing the sequence and acknowledgment numbers, port numbers, flags, and other information. The TCP segment is again wrapped in an IP packet before it is sent over the internet to the remote host.

2.3 Segmentation and acknowledgement

TCP is a connection-oriented protocol, and the protocol is responsible for segmenting the data stream received from the higher layers. This must be done in order for the TCP layer to send the data stream over the packet switched IP network. The segments have to be numbered in order for the TCP layer at the receiving end to be able to put the stream back together again.

The sequence numbers are selected starting at a cryptographically secure random number. This is called the initial sequence number or ISN. The sequence number is the first bye carried in the segment. If the ISN was 1 in the first segment which contained 1000 bytes of data, the sequence number of the next segment will be 1001.

On the receiving end the TCP layer has the responsibility for putting together the segments in a correct order and with error free segments. TCP use one buffer to send packets and one to receive packets. The buffers are where the implementation of the flow and error control mechanisms is implemented. The receiver responds to the sender by sending an acknowledgement.

An acknowledgement is a receipt to the sender. The receipt confirms the proper reception of all segments up to some point. The acknowledgement indicates the next byte expected from the sender, and the acknowledgement number can also be used to request a retransmit of a lost segment. The acknowledgement number is cumulative, which means that adding one to the last byte received. This is then the acknowledgement number [36].

Because a TCP connection is a full duplex connection, the acknowledgement can be piggybacked with next segment going out. This way the overhead caused by sending acknowledgement packages is minimal.

2.4 Connection establishment and termination

When a computer wants to establish a new TCP connection this process is called the Three-way handshake, and connection termination is performed through a process called the Four-way handshake. Both the connection establishment and connection termination are controlled by setting the bits in the flags field of the TCP header. The flags are URG, ACK, PSH, RST, SYN, and FIN. The SYN flag is used to initiate a TCP connection, synchronizing the sequence numbers during connection establishment. The ACK flag confirms the proper reception of a segment, or all segment up to some point. The ACK flag indicates that the acknowledgment field in the TCP header is valid. The PSH flag indicates that the segment contains data, and not just the header. FIN indicates to the receiver that the remote host wants to terminate the connection, with the four way handshake. This is similar to the RST flag which causes the TCP connection to be aborted with out performing the four way handshake. Finally we have the URG flag, which is not in use anymore.

TCP supports full duplex communication. As a result of this, the sender has to get a confirmation from the receiver after a connection setup, and before any data can be sent.

2.4.1 Three-way handshake

The connection establishment in TCP is done through a three-way handshake, as seen in Figure 1. The connection establishment is initiated from a client, but before the client can connect to another host, the server, it has to start accepting connections on a port. This is called passive open. When the client tries to connect, it requests an active open. If the server accepts the connection from the client the three-way handshake begins [36].

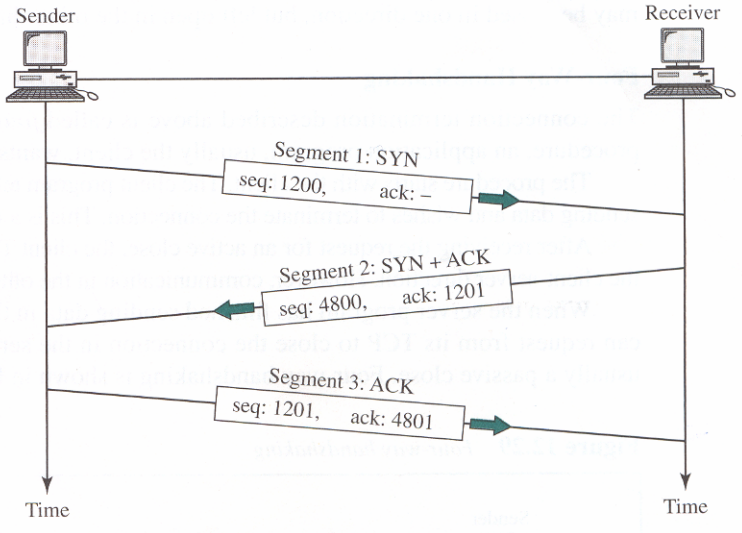

Figure 4: Three-way handshake

The client sends the first segment, with the SYN flag set. This segment contains information about the source and destination port numbers. These identify which processes will be communicating. The segment also contains the clients initial sequence

number, which is used for numbering the segments from the client. This segment may also define the max segment size here. This segment can be seen in Figure 4, marked as Segment 1.

Secondly, the server sends a segment with the SYN+ACK flags set. This segment acknowledges the first segment from the client, and is used as the initialization segment for the server. This segment contains the client window size, and may also define the server's maximum segment size and the window scale factor. The segment sequence number is initiated with the servers initial sequence number. This segment is marked Segment 2 in Figure 4.

Thirdly, the client sends a segment with the ACK flag set. With this segment the client acknowledges the receipt of the previous segment. The segment also defines the server window size. This segment may contain data from the client to the server. When this segment is received the three-way handshake is complete. This segment is marked Segment 3 in Figure 4.

2.4.2 Four-way handshake

Connection termination can be initiated both from the client and the server side, as seen in Figure 5. When one end close the connection the other end can continue to send data until it decide to close the connection as well. This means we have to use a four-way handshake to close the connection in both directions [36].

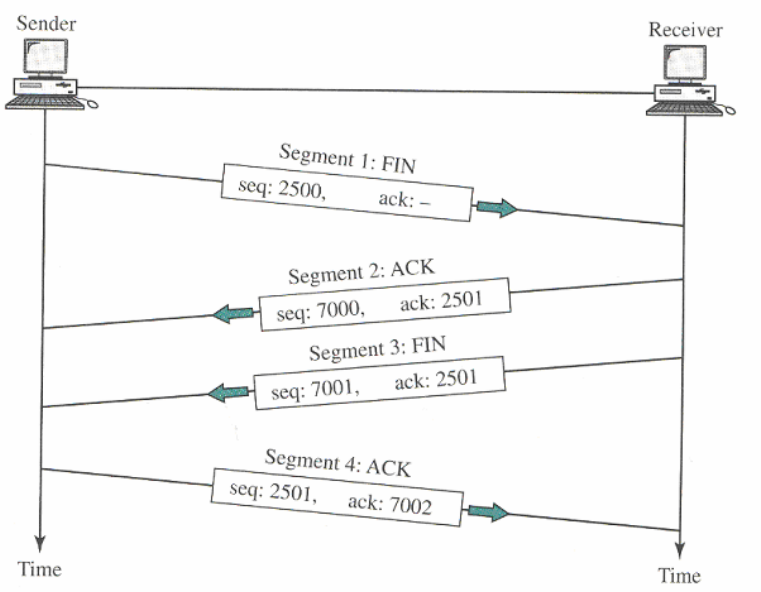

Figure 5: Four-way handshake

First, if we assume the connection termination is initiated by the client, the client sends a segment with the FIN flag set, as seen in Figure 5 (Segment 1). This is called an active close [36].

Secondly the server acknowledges the receipt of the previous segment with a segment where the ACK flag is set (Segment 2). The acknowledgement number is one plus the sequence number of the previous segment. When the client receives this packet it closes the connection from the client to the server, while the connection from the server to the client is kept open until the server request a connection termination.

When the server is finished sending all the data, the third step is initiated. The server sends a segment with the FIN flag set to the client (Segment 3). This is called a passive close [36].

In the last step the client sends a segment to the server with the ACK flag set (Segment 4), and the acknowledgement number set to one plus the sequence number of the FIN segment received from the server. This completes the four-way handshake and the connection is now terminated in both directions.

2.4.3 Reset

In addition to the four-way handshake, the connection may also be reset. This is not a graceful connection termination, and is used when some kind of abnormality occurs, or when one end of the connection wishes to abort the connection. A connection reset is carried out when one end of the connection send a segment with the RST flag set.

2.5 Sliding Window Protocol

TCP provide flow and error control trough a mechanism called the Sliding Window Protocol. This protocol control how much data can be sent before an acknowledgement is received, and help us avoid network congestion. This protocol is implemented at the sender and the receiver. A window is a portion of the send or receive buffer. At the receiving side the TCP protocol implements the protocol to only accept TCP segments when they lie within the current window. The window slide forward when it has received the lower bytes in the buffer correctly. And in this way start to accept higher bytes. An acknowledgement segment is sent when the receiver has verified the correct reception of a segment. If the sender doesn’t receive the acknowledgement within a predefined time, a retransmission is triggered. Eventually the segments will be received or a connection reset is performed. The sender and receiver are able to inform each other of the window size, making it possible to throttle the send/receive speed.

3 FAULT TOLERANCE

In this chapter we give a general overview of common fault tolerance terms. In this chapter we try to give the reader tools to comprehend the concept of the middleware approach to fault tolerant TCP-based services, which is presented in this report. The literature in this chapter is a summary of basic fault tolerance in distributed systems, which can be found in general computer science curriculum [37].

3.1 Background

It is important to design distributed systems in such a way that it can automatically recover from partial failures without seriously affecting the overall performance. When failures occur, the distributed system should be able to operate acceptable while repairs are being made. This is called **process resilience**. **Reliable multicasting** guarantees that message transmission to a collection of processes is executed successfully. It is necessary to keep processes synchronized. To recover from failures, one can save states to recover back to a functional state.

There are four basic terms in fault tolerance, called dependability: availability, reliability, safety and maintainability [2].

> ***Availability*** *means that services are available to the users at any given time.*
>
> ***Reliability*** *defines the uptime of a system.* ***Safety*** *means that failure does not lead to catastrophic events.*
>
> ***Maintainability*** *defines how easy a failed system can be repaired.*

All kinds of failure may occur, but they can be classified. The following schema is describes different classes of failure [3, 4]:

Type of failure	**Description**
Crash failure	A server halts, but is working correctly until it halts.
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times.

To make a system fault tolerant, the best thing to do is to hide the occurrence of failures from other processes. The key technique for **masking faults** is to use **redundancy**. Three kinds are possible: information redundancy, time redundancy, and physical redundancy [5]. Hamming code is an example of information redundancy. Extra information is added to handle noise. Transactions are an example of time redundancy. If a transaction aborts, it is redone.

Physical redundancy can be done in both hardware and software. If some crash, others resume the operations. **TMR (Triple Modular Redundancy)** is an example presented in Figure 6. Let us consider A, B, and C as network nodes. If A sends a message to C, it has to go through B (a). If B fails, A and C cannot communicate. A voter has three input channels, and needs two incoming messages to generate an output (b).

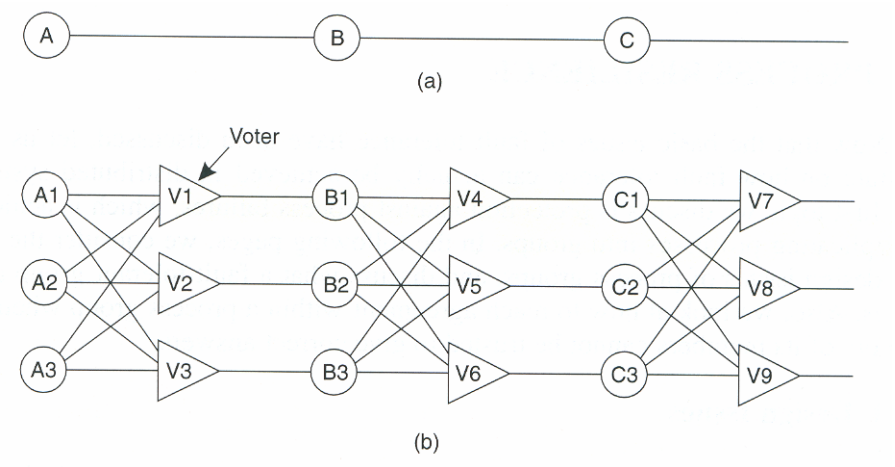

Figure 6: Triple modular redundancy

3.2 Process Resilience

Process resilience means protection against process failures in distributed systems. Processes can be organized into groups. When a message is sent to the group itself, all members of the group receive it [6]. If a process fails, someone else in the group can take over.

3.2.1 Flat Groups versus Hierarchical Groups

The flat group, of processes, has no single point of failure, but decisions take more time. The hierarchical group has a coordinator and therefore a single point of failure. The advantage is that decision making is more efficient, and the coordinator can dedicate tasks to processes that are more suited for the specific task.

3.2.2 Failure Masking and Replication

Having a group of identical processes allows us to mask one or more faulty processes in that group. This way one can replace a single (vulnerable) process with a (fault tolerant) group. There are two ways to approach such replication: by means of primary-based protocols, or through replicated-write protocols [37]. **Primary-based replication** is a hierarchical group of processes with a primary coordinator. The coordinator controls all write operations to the other processes (primary-backup protocol). **Replicated-write protocols** are organized in flat groups. There is no single point of failure. This protocol is used in the form of active replication [7] and quorum-based protocols [8, 9]. To implement active replication, totally-ordered multicasting is needed [10]. A system is said to be k fault tolerant if it can survive faults in *k* components and still meet its specifications. Having $k + 1$ processes is enough to provide *k* fault tolerance.

3.2.3 Agreement in Faulty Systems

This has to be taken in account if processes lie about their result. It takes $2k + 1$ processes to be k fault tolerant. This problem is called **Byzantine generals problem**. We will not describe this in further details, because this does not apply to our case.

3.3 Reliable Client-Server Communication

A communication channel may exhibit crash, omission, timing and arbitrary failures. Arbitrary failures may occur in the form of duplicate messages, resulting from the fact that in a computer network messages may be buffered for a relatively long time, and are reinjected into the network after the original sender has already issued a retransmission [11].

3.3.1 Point-to-Point Communication

This is done by using a reliable transport protocol, such as TCP. TCP masks errors and retransmits messages. Only crash failures are not masked.

3.3.2 RPC Semantics in Presence of Failures

There are five classes of failures that can occur in RPC systems, as follows:

1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

The solutions from these failures are presented, as follows:

1. Client Cannot Locate the Server

Having the error raise an exception can prevent crashes.

2. Lost Request Messages

If a request message is truly lost, a timer can be used to retransmit the message. If no replies arrive at the client, error 1 occurs. If two identical requests arrive at the server, error 4 occurs. The problem with this solution is that the client does not know if the server is slow or if the message was lost.

3. Server Crashes

There are several combinations of server crashes. The following scenario describes these combinations of failure: *A client sends a request for printing some text to a server. The server crashes.* A table containing the crash combinations and solutions for these errors is presented, as follows:

M	=	Completion message
P	=	Print the text
C	=	Crash

Client	Server					
	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(MP)
Always	DUP	OK	OK	DUP	DUP	DUP
Never	DUP	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

4. Lost Reply Messages
If a reply message is lost the client times out because no ACK is received. The client will then retransmit the request, thinking the request message was lost. The server might be slow and receive both the original request message and the retransmitted one. To avoid answering both messages, sequence number can distinguish original messages from retransmitted ones.

5. Client Crashes
There are several problems with a client crashing after it has sent a message. See page 380 for further details [37]. This problem does not apply to our case.

3.4 Reliable Group Communication

It is very important to guarantee reliable multicasting. All members in a process group must receive the message.

3.4.1 Basic Reliable-Multicasting Schemes

During multicasts, a receiver can miss a message. By sequencing the multicast messages the receiver can detect lost messages. If the last received message is 23 and it receives message 25, it can acknowledge to the sender that it has missed message 24. If the sender has a history buffer of all sent messages, it can simply retransmit message 24 [12]. One problem with the reliable multicast scheme is that it cannot support large number of receivers.

3.4.2 Scalability in Reliable Multicasting

If only negative ACKs are returned to the sender, this scheme can handle more clients than the solution represented above [13]. The problem is that the sender of multicast messages has to store all messages in a history buffer. The buffer has to delete messages so it will not be overflowed. Some clients might not receive certain messages at all. Different schemes are compared [14].

A **nonhierarchical feedback control** is a scalable solution that tries to reduce the number of feedback messages that are returned to the sender. The Scalable Reliable Multicast (SRM) [15] is based on **feedback suppression**. See pages 383-385 for further details.

A **hierarchical feedback control** is used to achieve scalability for very large groups. It divides a network in to smaller groups, so that a coordinator in each group can use any reliable multicast schemes to retransmit messages in their sub groups [16]. The multicast messages are only sent to the coordinators.

3.4.3 Atomic Multicast

Atomic multicast is supposed to guarantee that a message is delivered to either all processes or to none at all. Generally it is also required that all messages are delivered in the same order to all processes.

4 CONSISTENCY AND REPLICATION

In this chapter we give a general overview of common terms regarding consistency and replication. The literature in this chapter is a summary of theory on consistency and replication in distributed systems, which can be found in general computer science curriculum [37]. It is recommended to have basic knowledge about the topic, when we describe Sense's SiteCom and SiteCom® Rig architecture. For example, as we described in chapter 1.2 , a primary and backup server is replicated to tolerate failure, and the data sources are being kept consistent by Oracle applications.

4.1 Background

Reliability and performance are the primary reasons for replication. A reliability example: imagine there are three copies of a database, and every insert, update and select query is performed on each copy. We can safeguard ourselves against a single, failing insert or update operation, by considering the value that is returned by at least two copies as being the correct one.

4.2 Synchronous Replication

A consistency type called synchronous replication provides tight consistency [17]. The key idea is to update all replicas in one single atomic operation or transaction. If one replica is updated, all replicas should be updated before any new operations takes place. Difficulties arise because all replicas have to be synchronized and an agreement has to be made to decide when exactly an update has to be performed locally. The use of global ordering (Lamport's timestamps) or a coordinator to assign such an order has to be included. The gain is a scalable and improved performance solution, but the cost is global synchronization that affects the performance.

4.3 Data-centric Consistency Models

Traditionally, consistency has always been discussed in the context of read and write operations on shared data, available by means of (distributed) shared memory, a (distributed) shared database, or a (distributed) file system [37]. A broader term can be used: **data store**. It can be physically distributed across multiple machines. Copies or replicas of the data store can be accessed by different processes. A process can do it locally or nearby.

4.3.1 Strict Consistency

Basic idea: *Any read on data item x returns a value corresponding to the result of the most recent write on x.* This model assumes the existence of absolute global time and is therefore impossible to implement.

4.3.2 Linearizability Consistency

The closest you get to strict consistency. A global available clock is used to order the operations with timestamps [18]. Every operation is also executed in the sequential order. The drawbacks are loosely synchronized clocks that will not guarantee absolute time ordering and the expensive implementation [19].

4.3.3 Sequential Consistency

Basic idea of sequential consistency: *The result of any execution is the same as if the (read and write) operations by all processes in the data store where executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program* [20]. The sequence of write operations is not ordered in absolute global time, but every processor reads the same write sequence. Every read is guarantied to read the most resent write. Easy to implement (widely used), but poor performance.

4.3.4 Causal Consistency

Causal related events are being kept consistent [21]. For example: A causal related write has to be read by all other processes enforcing consistency. Concurrent writes are operations that are not causal related. These writes does not enforce consistency. No global order is implemented. To implement this consistency model, one needs to keep track of which processes have read which writes. One must also keep track of which operation is depended on which other operation. This can be implemented by the use of vector timestamps.

4.3.5 FIFO Consistency

All writes are concurrent. Only write operations from one process are ordered. This model is easy to implement and a process number and a sequence number for each write operation is necessary to ensure FIFO ordering for a processor's write operations [22].

4.3.6 Weak Consistency

This model uses synchronization variables to update replicas. When a process has accessed a critical section and performed write operations, all the replicas are synchronized. This model is most useful when isolated accesses to shared data are rare, with most accesses coming in clusters (many accesses in a short period, then none for a long time). One limits only the time when consistency holds, rather than limiting the form of consistency [23].

4.3.7 Release Consistency

Eager release consistency uses two variables for synchronization: acquire and release [24]. When a process acquires data, the local data is brought up to date by reading other replicas. When a release is done, the local copy's changes are being propagated to other replicas. If acquire and release occur in pairs, the result of any execution will be no different than a sequentially consistent data store. Operations on shared data are made atomic. In *Eager release consistency* no updates are propagated after a release [25]. Nothing happens when a release occurs. While acquiring, a process determines if it has all the data it needs. The result is reduced network traffic.

4.3.8 Entry Consistency

Entry consistency is similar to the release consistency model [26, 27]. The difference is that it requires each ordinary shared data item to be associated with some synchronization

variable such as a lock or barrier. This increases the amount of parallelism. Programming this model is more complicated and error prone.

4.4 Client-centric Consistency Models

Data stores exposed to a majority of read operations and few updates does not necessarily need strong consistency models. Since write-write conflicts are easy to resolve and most operations are read operations very weak consistency models are sufficient. Many inconsistencies can be hidden easily by applying eventual consistency.

Eventual consistency is suited for data stores that tolerate a high degree of inconsistency. If no updates take place for a long time, all replicas will gradually become consistent. All replicas converge toward identical copies of each other. It is cheap to implement. This model works fine if clients always access the same replica. Problems occur when mobile clients access different replicas. Eventual consistency solves this problem. If a mobile process performs updates on one local copy and accesses another, all previous updates will be performed on the new local copy before further updates occur [28, 29].

4.4.1 Monotonic Reads

This model quarantines that a client reading data x on one local copy L_1, it will read the same result x or a more recent version of x on another local copy L_2. This is ensured by propagating every update on L_1 to L_2 before the same client starts to work on L_2.

4.4.2 Monotonic Writes

When a write is done on a local copy x, all preceding write operation on any copy of x is performed first. The local copy x is being brought up to date before any write operation is performed [29].

4.4.3 Read Your Writes

A write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place [30]. Solves problems, like for example, web-browsers not show updated pages.

4.4.4 Writes Follow Reads

Any successive write operation by a process on a data item x will be performed on a copy of x that is up to date with the value most recently read by that process [29].

4.5 Distribution Protocols

Protocols for distributing updates to replicas will be discussed in this chapter.

4.5.1 Replica Placement

There are three main types of replica placement categories to be considered when placing replicas in a distributed system [31]. *Permanent replicas* are widely used in distribution of web sites [32] and databases [33, 34]. Some of these techniques are known as mirroring, shared-nothing strategies and federated databases. These techniques spread replicas on a limited number of machines. *Server-Initiated replicas* dynamically push

replicas close to clients to enhance performance [35]. *Client-Initiated Replicas* is used to improve access time by caching replicas at clients. They are responsible for managing the update of the replica.

4.5.2 Update Propagation

What should be propagated during updates? Three basic approaches: 1) Propagate only a notification of update, 2) Transfer data from one copy to another, and 3) Propagate the update operation to other copies. Issues with *pull versus push protocols* and *unicasting versus multicasting* is discussed on page 331-334 [37].

4.5.3 Epidemic Protocols

This protocol, one assumes that all updates for a specific data item are initiated at a single server. Two methods are essential: *anti-entropy* and *rumor spreading* (gossiping). The *anti-entropy* method is very similar to disease spreading and uses either a push or pull to exchange updates, or push and pull. It is easy to spread updates and hard to delete copies.

5 MigWare architecture

In these sections we give an overview of the MigWare middleware's architecture. MigWare consists of four modules which we described in detail in this chapter. A detailed explanation of the algorithm used to recover TCP connection is also given. It explains how MigWare modifies the sequence and acknowledgment numbers in the TCP header. Some of the restrictions in the current implementation of MigWare are also described. The source code for the MigWare middleware prototype can be found in the appendix [51].

5.1 Overview

MigWare is developed as a proof of concept study to demonstrate the feasibility of developing a middleware solution to recover and migrate TCP connections, and to measure its performance. An important premise for this middleware is to avoid modification to the TCP/IP stack. The architecture we have devised consists of four modules as seen in Figure 7, namely a Sniffer module, a Parser module, a Connection Tracker module, and a Monitor module.

The Sniffer module has the responsibility of detecting all traffic on the local network and buffering this traffic in terms of raw packets. The Parser module processes the buffered raw packets in order to extract the data fields that are relevant for tracking TCP connections. Furthermore, packets that are not related to TCP traffic are thrown away for performance reasons. The extracted data fields are in turn passed to the Connection Tracker module. The Connection Tracker module maintains information about current TCP connections to the Primary Server based on the information provided by the Parser module. In essence, the Connection Tracker module identifies which parsed packets belong to which TCP connection and updates the tracked state of the respective TCP connections accordingly.

Independently of the other modules, the Monitor module monitors the Primary Server continuously in order to detect whether the server has crashed. When a crash is detected, the other modules are notified, so that they can take appropriate recovery actions. The purpose of these recovery actions is to migrate the TCP connections of the Primary Server to the Backup Server, based on the connection state maintained by the Connection Tracker module.

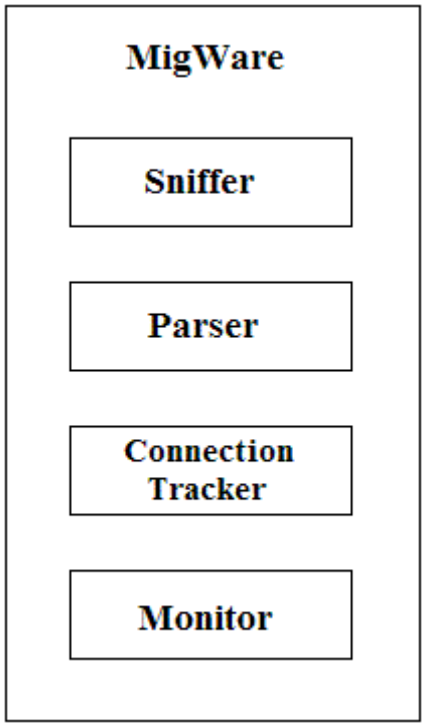

Figure 7: MigWare modules

Our design is one of three possible designs patterns for a fault tolerant recovery system; Standalone, Integrated, and Proxy [39]. The design seen in Figure 7, which is used by MigWare, is based on the Proxy design. This design does not require any modification to the service we intend to replicate, and therefore no access to the source code is necessary. This design is implemented as a middleware, which translates packets to or from the client, and maps them to the new connection. While the primary server is running, the middleware is able to log the necessary TCP state information. When the primary server crashes, the middleware can act as a packet forwarder for the connections originally intended for the primary server. It pipelines packets from the client to the backup server. An example of this communication can be seen in Figure 8. In addition MigWare lets the TCP/IP-stack take care of the communication as much as possible. MigWare is true to the internet philosophy, where the intelligence is supposed to be implemented in the computers, while keeping the network nodes as dumb as possible. We regard MigWare as a network node, even though it is a middleware running on the backup server.

The Standalone design uses two components. All the recovered connections are serviced by a standalone software application, which is a modified version of the standard service [39]. A standard unmodified service handles all the new connections to the service. This implementation requires two services running on the backup server, and access to the source code of the service. This might not be possible because the source code for a service might not be available. An implementation of the Standalone design might involve modification to the kernel and to the standard service. This would require access to the operating systems source code, in addition to the services source code, but might give the recovered connections a performance boost. An implementation without kernel

modifications would involve using a similar process as the one used by MigWare, which is described in detail in the following sections.

The Integrated design relies on modifying the service to support recovery of connections, as well as accepting new connections. This might also involve modification of the client software, because the modifications to the service might require protocol update to support connection recovery [39]. This implementation requires only one service on the backup server, but access to the source code is required to modify the standard application to recover connections. An implementation of the Integrated design might involve modifications to the kernel and the service in much the same way as the Standalone design. The kernel modification would give the recovered connections a performance boost, but require access to the kernel source. This is the category in which we would place Jeebs [39], because all the connections to the server are serviced by one application. It could be argued, since Jeebs use an additional computer to monitor the TCP connections and replay the connection to the backup server when the primary fails, it fall into the Standalone category. The Integrated design can also be implemented without kernel modification, but this would involve using a similar process as the one used by MigWare. This process is described in detail in the following sections.

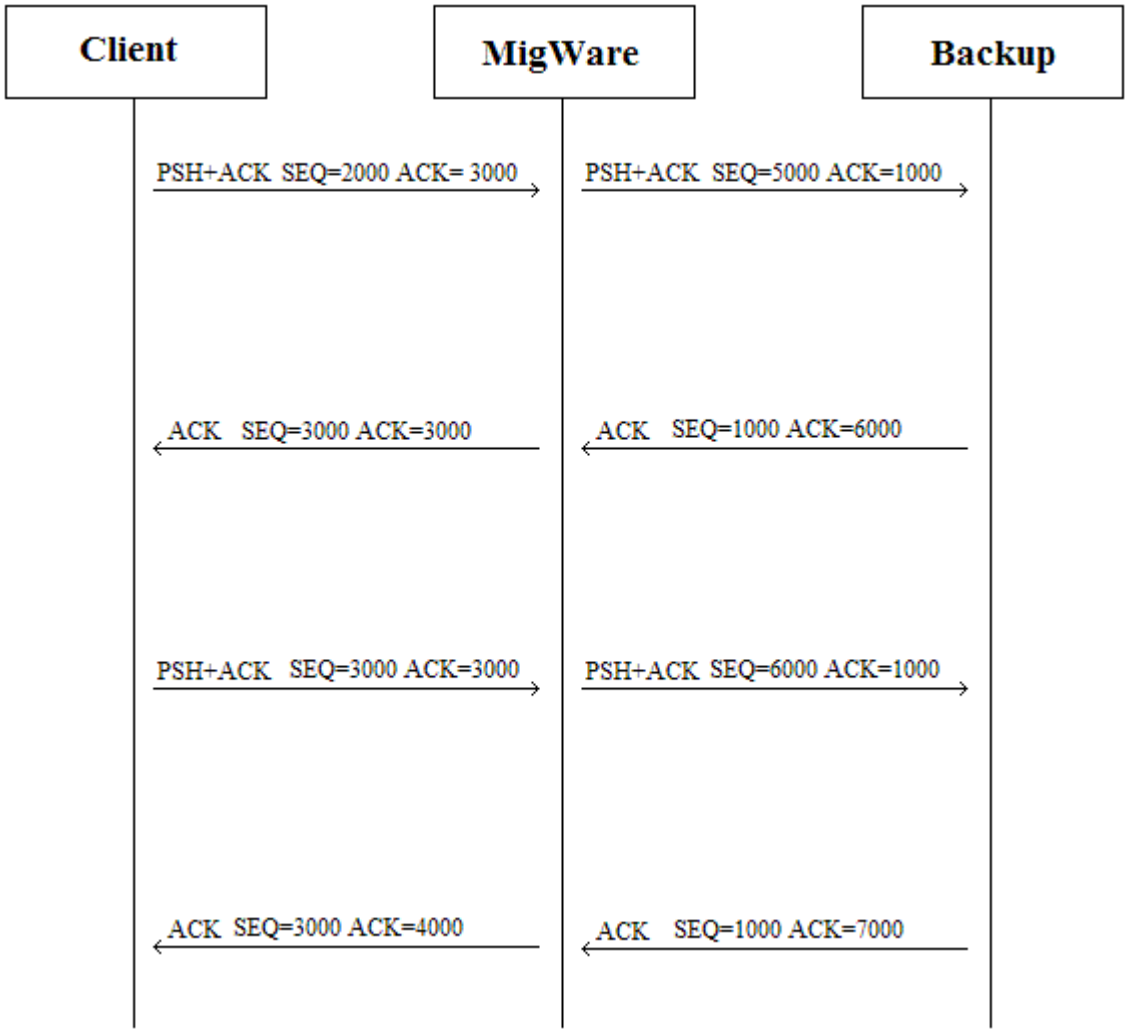

Figure 8: Sequence and acknowledgement number mapping

In Figure 8, we can see how the client sends a packet with a sequence and acknowledgement number. MigWare detects this packet and converts the sequence and acknowledgement numbers based on the sequence and acknowledgement previously collected from the connection. In order to map the packets to the new connection, the IP addresses, port numbers, checksums, and MAC addresses need to be modified as well. It then sends the modified packet to the backup server. A connection uses a different sequence numbers in each direction. When a TCP connection is created, the sequence and acknowledgement numbers are randomly generated as discussed in the Transmission Control Protocol section. It is necessary to monitor the packets in both directions to successfully modify the packets in both directions. While MigWare monitors the primary server's connections for these TCP/IP header fields, it is in the Normal state. The forwarding of packets, seen in Figure 8 is performed while in the Proxy state, seen in Figure 9.

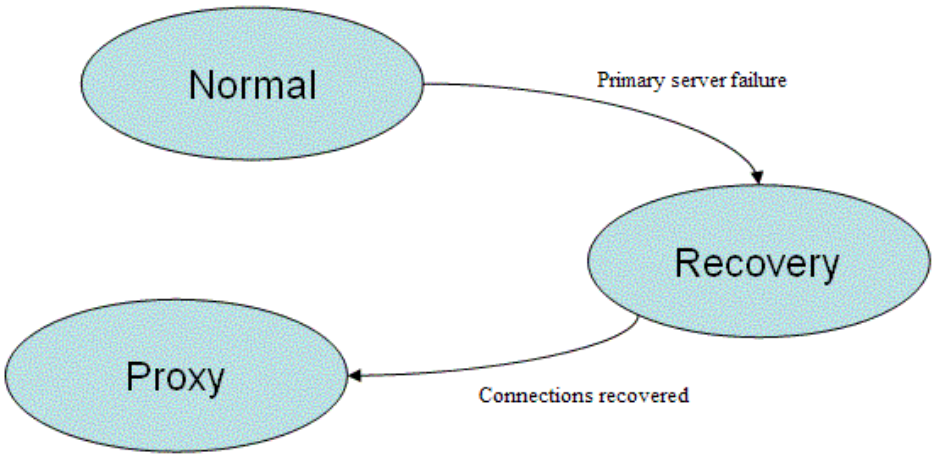

Figure 9: MigWare state chart

As seen in Figure 9, MigWare operates in three states. Normal state is when the primary server is running and no failure has been detected by MigWare. A problem can be when the primary server crashes, when a TCP service on the primary server crashes or when the primary server looses its network connection. In this state MigWare does not actively participate in the communication, but monitors the traffic between the primary server and the clients.

The second state is the Recovery state. MigWare makes a transition to this state when a monitor reports a failure with the primary servers operation. When MigWare is in this state, all the connections to the primary server are recovered, but only those who have been detected. This involves setting up new connections, with similar properties, between MigWare and the backup server.

When all the connections have been recovered, MigWare transitions to the third and final stage. In the Proxy stage, MigWare acts as a proxy server. All packets from the clients to the primary server are sniffed by MigWare and modified, before they are sent to the replicated service on the backup server. The same is true when the backup server is sending packets to MigWare.

5.2 Sequence and Acknowledgement number algorithm

While MigWare is in the Proxy state, it operates on two TCP connections for each recovered connection. MigWare monitors the initial crashed connection and forwards data to the new recovered connection between itself and the backup server. MigWare uses four key sequence numbers collected from the two connections three-way handshake, and from the last retransmission of the crashed connection. Using these four sequence numbers, it is possible to map a sequence and an acknowledgement number from the crashed TCP connection, to the new recovered TCP connection, from MigWare to the backup server. The same is possible in the reverse direction.

While MigWare is running in Normal state, it continuously listens for TCP segments with the SYN flag set. This flag indicates a client wants to establish a new connection. When

MigWare detects a segment with the SYN flag sent to the primary server, a new connection object is created. A connection object is MigWare's representation of a TCP connection between a client and the primary server. The primary server will reply, to the SYN segment, with a TCP segment with the SYN+ACK flags set, indicating that the server accepts the connection. Finally, the client sends a TCP segment with the ACK flag send, acknowledging the proper reception of the SYN+ACK segment. This sequence can be seen in Figure 10.

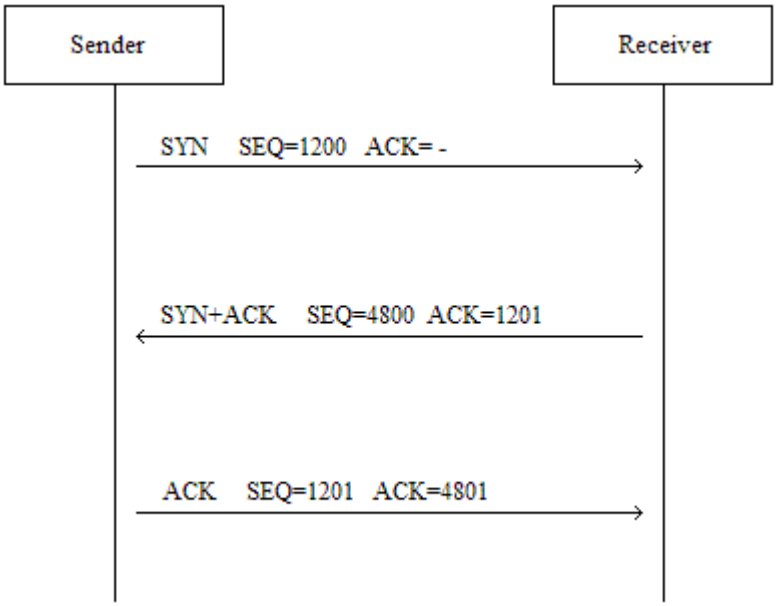

Figure 10: A three-way handshake/connection establishment

A new connection is created, and MigWare registers the initial sequence numbers in both directions. The source and destination IP addresses uniquely identifies a connection together with the source and destination port numbers. This is used as an identifier for that connection.

MigWare also registers the MAC address where the segment originates from. This enables us to physically send the packet on the Ethernet, as discussed in the previous chapter. The client continues to send packets on the connection, while MigWare keeps track of the last sequence and acknowledgement numbers.

When MigWare detects a failure with the primary server, it enters the Recovery state, and the information previously collected is used. With the information collected and stored in the connection object, it is possible to construct a SYN segment. In Figure 11 below, we can see a scenario where a client is communicating with the primary server. MigWare detects the connection between the client and the primary server.

In Figure 11 we can see how MigWare detects the connection establishment between the client and the primary server (1). This communication from the client is originally intended for the primary server, but it is also received by MigWare. The SYN+ACK

segment in the connection setup (1) is marked with a red label (MCS). This is a label for the sequence number this packet has, which is used later.

The client continues to push data to the server, up to the point where the primary server stops responding (2), and the client starts to retransmit the packet. The sequence number of the last retransmit is labeled (CMS). MigWare will detect that the primary server has crashed, and transitions into the Recovery state. In (3) we can se how MigWare uses the CMS, also called MBS, to construct a SYN segment. The backup server does not communicate directly with MigWare. All packages sent from MigWare are addressed from the primary server, as discussed above. The SYN+ACK segment received from the backup (3) is labeled (BMS). Using the MBS and BMS MigWare is able to respond correctly with an ACK segment. MigWare has successfully recovered the connection, and is now ready to start transmitting the data segments (4) originally sent to the primary server.

Using the four sequence numbers we have labeled MCS, CMS, MBS, and BMS we are able to calculate the difference between the sequence numbers on both TCP connections. One side is the TCP connection between the client and the primary server, which is mapped to the primary server to backup server connection. The formula for calculating these two differences is as follows:

$$D1 = MBS - CMS$$
$$D2 = BMS - MCS$$

Using these two differences we are able to formulate four formulas for converting the sequence and acknowledgement numbers between the two TCP connections. The equations are as follows:

$$ClientToBackupSeq = Seq + D1$$
$$ClientToBackupAck = Ack + D2$$
$$BackupToClientSeq = Seq - D2$$
$$BackupToClientAck = Ack - D1$$

Because the segment and acknowledgement numbers are 32bit long, the above equations need to take this into account. When the number becomes larger than 32bit the number starts at zero again. This is also the case when the number becomes less than zero, when the number wrap around to the maximum value represented by 32bit.

For example if MigWare receive a segment from the client to the primary server, and want to map this segment to the primary server to backup server connection, we use the ClientToBackupSeq formula. In this formula the Seq is the original sequence number of the segment before we modify it. The same is done when converting the acknowledgement number, but using the ClientToBackupAck formula and the original acknowledgement number.

Continuing with the example in Figure 11, we see the segment retransmitted in (2) is modified and forwarded to the backup server (4). The ACK segment which the backup server responds with is mapped to the primary server to client connection (5). The sequence and acknowledgement numbers are converted using the BackupToClientSeq and BackupToClientAck formulas.

Figure 11 shows how the response from the backup, the ACK segment (4), because MigWare first has to recover the connection. When the client has received the ACK segment the client can again start to transfer new data to the server (5). The mapping MigWare now does is much faster than the first segment, where the connection has to be recovered first.

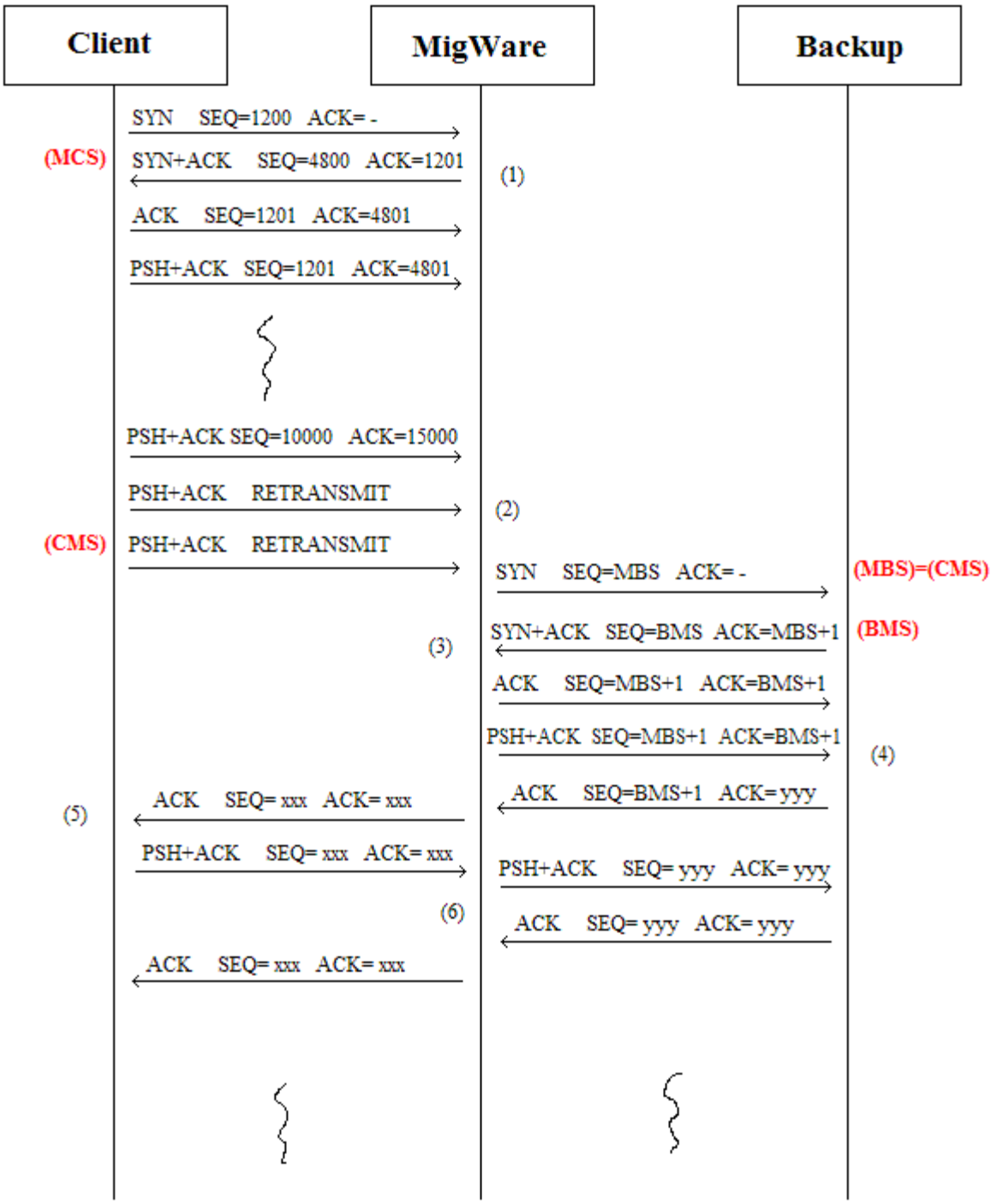

Figure 11: A scenario where the primary server crashes

This figure illustrates the translation of sequence and acknowledgement number between two connections.

5.3 Modules

This section contains a more detailed description of the four modules in MigWare. The architecture consists of four modules as seen in Figure 7, namely a Sniffer module, a Parser module, a Connection Tracker module, and a Monitor module. The Sniffer module has the responsibility of detecting all traffic on the local network and buffering this traffic in terms of raw packets. The Parser module processes the buffered raw packets in order to extract the data fields that are relevant for tracking TCP connections. Furthermore, packets that are not related to TCP traffic are thrown away for performance reasons. The extracted data fields are in turn passed to the Connection Tracker module. The Connection Tracker module maintains information about current TCP connections to the Primary Server based on the information provided by the Parser module. In essence, the Connection Tracker module identifies which parsed packets belong to which TCP connection and updates the tracked state of the respective TCP connections accordingly. Independently of the other modules, the Monitor module monitors the Primary Server continuously in order to detect whether the server has crashed. When a crash is detected, the other modules are notified, so that they can take appropriate recovery actions.

5.3.1 Monitor implementation

In order for MigWare to detect a server failure we have implemented a monitor. Because this is only a prototype the monitor is relatively simple. It monitors the server for crashes through sending a series of ICMP ping packet at a 500ms interval. When the server fails to respond to these ping packets, MigWare will make a transition to the Recovery state and initiate the recovery procedure.

In a production system one would have several monitors, which would monitor several properties and services in order to detect a server crash. It would also be able to detect imminent server crashes, and in some cases take pre-emptive measures.

Because both the primary and the backup server are located on the same network node (hub/switch), it is sufficient to send ICMP packets to detect server crashes. We have discovered that since the packet loss is very low between our two servers, we can say that the server has failed already after one ICMP packet fails to give a response. But this might have to be adjusted in a production environment. MigWare sends ICMP packets at a 500ms interval, and the packets have a 500ms timeout. This gives us a 500-1000ms delay before we can detect a server crash. We are also not able to detect anything other than server crashes, because this has not been the main focus. Other methods of monitoring the health status of a service have been covered elsewhere in the literature [41], [42], [43].

When a server crash is detected by the monitor, all the modules listening will be notified. MigWare will also transition into the Recovery state, and initiate the recovery procedure.

5.3.2 Sniffer implementation

The sniffer module has the responsibility of collecting information about all the TCP connections running on the local network. Using a library called WinPcap [38] the sniffer is able to tap into Ethernet wire and collect all the traffic reaching the network interface. Because of different network topologies different methods has to be employed in order for all traffic to reach the network interface and the sniffer. All the collected packets are buffered, before being processed.

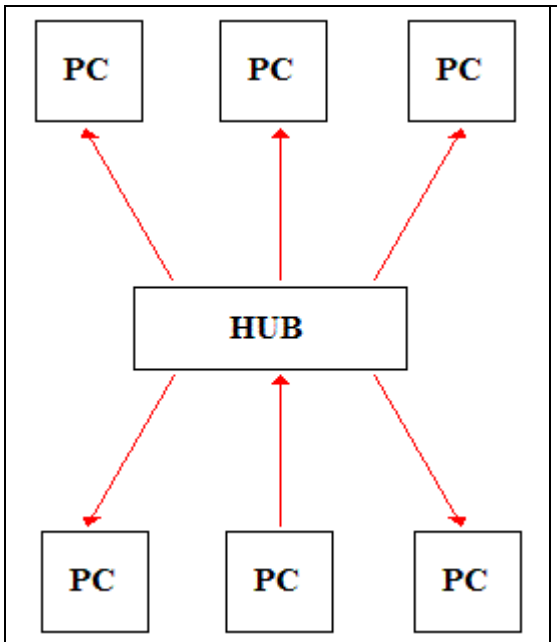

Figure 12: A Hub with bus architecture

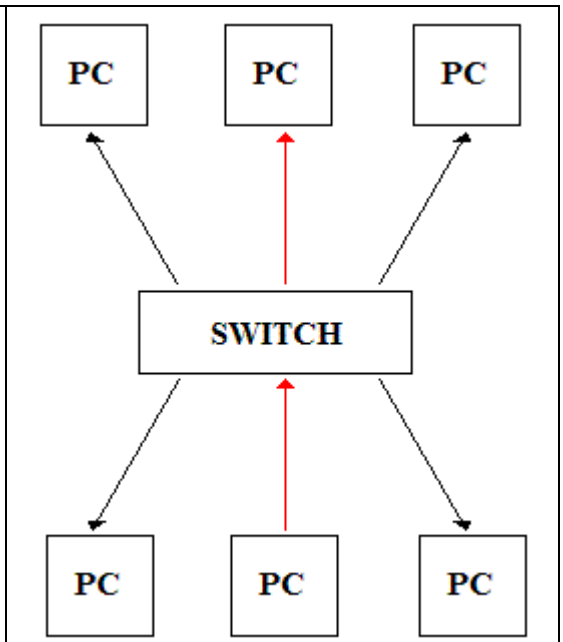

Figure 13: A switch with star architecture

Figure 12 is an example of two hosts communicating over a bus. A hub is a network device based on the bus architecture, where any data transmitted from one host to a second host is broadcast to all the hosts connected to the hub, as indicated by the red arrows. This means a sniffer on any of the hosts connected to the hub can monitor all traffic to and from a second host connected to the hub.

Figure 13 show two hosts communicating over a switch. A switch is based on the star architecture, where traffic to a specific host is only transmitted on the port the receiver is connected to, as indicated by the red arrows. This means a sniffer on one of the four other hosts not participating in the communication cannot monitor the traffic. The network traffic would not reach their network adapter.

In order for MigWare to be able to track and recover the TCP connections to the primary server, the backup server running MigWare, need to receive the same packets as the primary server. An occasional packet might be lost, because of network failure, but we have discovered that this is not a significant problem. The amount of packets lost is so low it will not affect MigWare's operation. For a packet loss to affect MigWare's

operation, one of the three segments sent in the three-way handshake has to be lost. We have observed the probability for this to happen is very low.

MigWare needs to detect all the packets sent to the primary server. This can be accomplished in several ways. The easiest way, and the way we use in our test environment, is to use a network hub. A network hub broadcast all the traffic to all the ports simultaneous. This enables us to sniff all the packets sent to and from the primary server and its clients. This is what we have relied on during the development of the prototype.

One other solution to this problem is to use a high end managed switch. Normal switches unicast the traffic between the ports. This would create a problem for MigWare, because MigWare would not be able monitor the packet stream between the primary server and its clients. Almost all high end managed switches has the ability to forward all traffic on one port to a second port, thus solving our problem.

A third solution, which we have explored, is to send the packets to a multicast MAC address. This solution would demand support for multicast MAC addressing in the network switch and by Windows. These features are not supported by Windows, and would thus require modifications to the network stack on the client and server. We have tried to broadcast the traffic, but it generated too much traffic on all the ports. We do not want to overflow the switch.

To get raw access to the Ethernet we use a library called WinPcap. WinPcap is a kernel-level packet filter, which give us direct network access under windows. This enables us to circumvent the TCP/IP stack. This is necessary because we need access to all packets detected by the network adapter. Otherwise the TCP/IP stack would filter away some packets, and would not give us access to all the information contained in the headers. Then MigWare would not be able to replicate an exact copy of the data stream, which is necessary when mapping the TCP connection between the primary server and the client, to the recovered connection between MigWare and the backup server.

Using normal system primitives, like sockets, one would only have access to the traffic intended for the host itself. But because we use WinPcap we are able to set the network interface in a promiscuous mode. When the network interface is in promiscuous mode, WinPcap is able to sniff all the traffic the network interface detects. And because we use a network node with broadcast capability, all traffic to the primary server is now sniffed by the WinPcap library running on the backup server. This means all traffic for the primary servers port on the hub is also sent to the backup server’s port.

5.3.3 Parser implementation

The parser takes the raw data collected by the sniffer and identifies the IP packets. Using the information in the packets header fields, the parser can discard all non TCP packets. It also look at the IP packets source and destination fields and filter away all packets other than the ones addressed to or from the primary server, the clients, and the backup server.

The remaining packets are forwarded to the Connection tracker. The information contained in the packet headers has been identified by the sniffer, and is also forwarded. The parser avoids moving the payload of the packets, thus limiting the load on the CPU. MigWare is almost unaffected by the packet size, and it is the number of packets which affects the CPU load.

Using the information contained in the packet headers, the Connection tracker can decide which connection a packet belongs to.

5.3.4 Connection tracker implementation

The Connection tracker uses the TCP/IP header fields to identify which connection a packet belongs to. A TCP connection and its packets can be uniquely identified by the source address, destination address, source port, and destination port. The Connection tracker stores the information from the TCP/IP headers. This information is needed for connection recovery. Eventually MigWare will detect a primary server crash, and the connection tracker will use the stored information to create a three-way handshake to the backup server, recovering the connections to the primary server.

The connection tracker receives packets from the parser. All traffic not related to the primary server, backup server or the clients has already been filtered away. The connection tracker has to decide what to do with the incoming packet. Initially MigWare starts in the Normal state. In the normal state MigWare listens for new connections to the primary server, and keeps track of some information needed to recover a connection. This information includes the TCP sequence numbers, TCP acknowledgement numbers, IP source and destination port numbers, source and destination IP addresses, source and destination MAC addresses, and the sequence and acknowledgement number of the last segment retransmitted by the client. When this information has been collected, the packet is discarded.

Eventually the primary server will crash, and MigWare is notified about this by one of its monitors. This triggers the transition to the Recovery state, and the initiation of the recovery procedure. All the necessary information about the existing TCP connections between the clients and the primary server has been tracked. Using this information we can create fake SYN segments to initiate the creation of a new TCP connection between MigWare and the backup server.

All packets received while MigWare is in the Recovery state are buffered, and because MigWare uses two buffers while in Recovery mode we are able to inspect the packets without loosing them. MigWare uses FIFO buffers to improve performance. When reading a packet inserted into a FIFO buffer, the packet will be removed from the buffer. While in Recovery mode, MigWare needs to inspect the packets looking for incoming SYN+ACK segments. This requires the use of one additional FIFO buffer to prevent packet loss, inserting all the inspected packets into the second FIFO buffer. All the packets inspected while in the Recovery mode need to be forwarded when MigWare makes the transition to the Proxy mode. This allows us to detect the proper reception of a SYN+ACK segment from the backup server, and respond with a fake ACK segment.

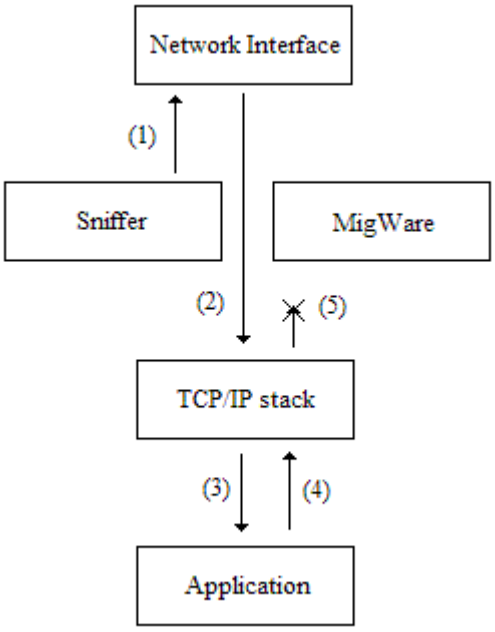

Figure 14: A scenario when MigWare cannot detect packets

Figure 14 show a possible solution for establishing a recovered connection between MigWare and the replicated service on the backup server. This solution is used by many similar solutions, as mentioned in chapter, but does not work when using a middleware solution

When MigWare is in the Recovery state, it establishes a connection to the backup server. Figure 14 shows an example where MigWare is running on the backup server and sending the SYN segment from MigWare to the backup (1). This segment has the backup server as both source and destination address. The source address is the backup server because MigWare is running on the backup server, and the destination address is the backup server because the packet is intended for the redundant service running on the backup server. The network interface will detect that the segment is addressed to the local host, and forwards the packet to the TCP/IP stack of the backup (2). The TCP/IP stack tries to establish a connection with the application on the backup server (3). When the application accepts the connection request (4), the TCP/IP stack tries to send a SYN+ACK segment back to the source of the request, as part of the three-way handshake. Because the segment's destination is the backup server, the segment will never actually be sent on the wire (5). Instead the segment is forwarded directly to the destination port on the local host, by the TCP/IP stack. The SYN segment was generated by MigWare and not by a socket, and therefore the segment cannot be received through one. MigWare needs to receive all its segments through the sniffer, and the sniffer only detects packets actually sent on the wire. This is why MigWare uses a more complicated system to communicate with the backup server.

A solution to the above problem might be to send the SYN segment to a random host other than the backup servers. This would cause the packet to be transmitted on the wire and detected by the sniffer. The only problem with this is that the address might already be in use.

In order to send a SYN segment MigWare needs the IP address, port number, and the MAC address of the destination host. When a computer transmits an IP packet on the Ethernet, the IP packet needs to be wrapped in an Ethernet frame. This frame contains the source and destination MAC addresses. The Ethernet frame encapsulates the IP packet, and transmits the packet between two computers connected to the same local area network. Then the TCP/IP stack of the receiver will forward the packet over a different network if needed. Initially the MAC addresses of the other hosts on the local area network are unknown. If the computer already has transmitted a packet to the destination, the MAC address has been saved to the senders ARP cache, and is accessible for later use. The problem is when no MAC address for the destination exists in the ARP cache. Then the computer needs to perform an ARP request. This means broadcasting a special frame requesting the MAC address of a host with a specific IP address. This frame is transmitted with a predefined broadcast MAC address as the destination. The host will send an ARP reply with the MAC address. The come when the computer tries to transmit the packet to the host which does not exist. A MAC address for the destination host does not exist in the ARP cache, so an ARP request has to be performed before sending. If the IP address does not exist, then there would be no reply to the ARP request, and the sender would not have any where to send the packet. Then the packet will never be sent. This is the reason why MigWare has to use the solution seen in Figure 15.

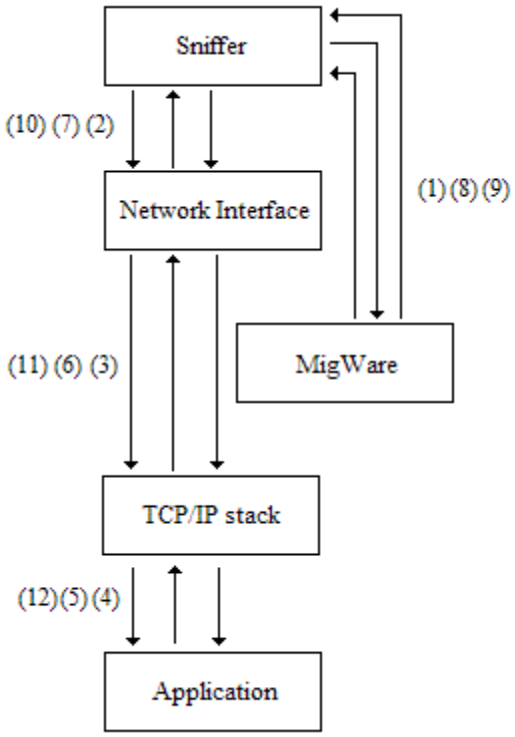

Figure 15: A scenario where MigWare recover a connection

We found a solution, enabling us to drop the implementation of at ARP cache injector. In stead of sending the segment to a random host, we send the segment to the primary host. We have already determined that the primary host is down, so there would not be any response. In addition the ARP cache on the backup server contains an entry with the MAC address of the primary host.

This solution is shown in Figure 15, and it exploits the fact that the backup server already has the MAC address of the primary host in the ARP cache. By pretending that the backup server is communicating with the primary server, MigWare can detect the communication and modify the packets before transmitting them to the client.

In Figure 15 MigWare start by informing the sniffer of a SYN segment it want to send (1). The sniffer sends this segment to the network interface (2). The segment is addressed from the primary server which is down, but whose MAC address still exist in the ARP cache on the local computer. As before, the network interface will detect that the segment is addressed for the local host, and forward it to the TCP/IP stack (3). The TCP/IP stack will try to setup a connection with the application (4), and if the application accepts the connection (5), the TCP/IP stack will send a SYN+ACK segment to the network interface (6). This segment is addressed from the local host to the primary server. The network interface will transmit this segment, as it is addressed for the primary host (7). When the segment is transmitted, it is also sniffed by the sniffer, and MigWare is informed of the SYN+ACK segment (8). Using the information contained in the SYN+ACK segment, MigWare is able to construct an ACK segment as a response to the SYN+ACK segment.

The primary server is down, and no reply will be sent from this server. MigWare instructs the sniffer to send the segment (9). As before, the segment's source address is the primary server, but the destination is addressed to the local host. When the sniffer sends the segment (10), the network interface determines that the segment is addressed for the local host and forwards it to the TCP/IP stack (11). The TCP/IP stack expects this ACK segment as a response to the previous SYN+ACK segment, and completes the connection setup with the application (12).

This solution has one drawback. While the backup server is running in Recovery or Proxy state the primary server cannot be put online again. When the primary server come online, the packets sent to the primary server will reach its destination. But because the primary server does not know any of the connections anymore, it will reply with a segment with the RST flag set. This will abort the TCP connection.

The sending of TCP segments to the primary server address can be detected by MigWare. It can respond as if the primary server was running. The backup server acts as if it is communicating with the primary server.

When all the connections have been recovered, MigWare transitions to the final state, the Proxy state. All communication from the client to the primary server will now be detected. The packets will be modified, and sent to the backup server, with the primary server as its source. The backup server will respond by sending its packets to the primary server. MigWare will detect this traffic, modify it, and send it to the client with he primary server as the source.

All the necessary information to modify packets and map them to their connection has already been registered by the connection tracker. Both the IP packet, MAC frame and the TCP segment need to be modified. The MAC frame needs new source and destination addresses. The TCP segment needs new sequence and acknowledgement numbers, source and destination port numbers, while the IP segment needs new source and destination IP addresses. Finally we need to calculate a new checksum for the modified IP packet.

MigWare only translates the incoming packets, it does not need to keep track of which packets have been acknowledged and which have not. In essence, MigWare lets the TCP/IP stack take care of the communication as much as possible. MigWare is true to the internet philosophy, where the intelligence is supposed to be implemented in the computers, while keeping the network nodes as dumb as possible. We regard MigWare as a network node, even though it is a middleware running on the backup server.

5.4 Restrictions

When the primary server has crashed MigWare will recover the existing connections, but any new connection attempts to the primary server will fail. This feature can easily be implemented, but was deemed unnecessary in the prototype.

When the primary server has crashed and the backup server running MigWare has taken over, the primary server cannot be put online again with the same IP address while

MigWare is running in Proxy mode. This would cause all the recovered connections to be aborted. The only way to get the primary server online again is to use a controlled shutdown of all the connections recovered, and establish new connections to the primary server again. At the same time MigWare needs to be restarted. But a controlled shutdown would not be as problematic as an uncontrolled one.

It is possible, but unlikely, that the packets sent to the primary server, can get lost on the way to the backup server. This is not a problem as long as both primary and backup server does not receive the packets, but only when the primary receives them and the backup does not. The vulnerable point is the connection establishment. If these packets do not reach the backup server, MigWare will not be able to detect new connections. Other packets can be lost without causing too much interference with MigWare's operation. We have observed that the amount of packet loss is too low to cause any problems for the normal operation, and that the probability the lost packets will be one of the connection establishment packets is much lower.

6 Experiments

In this chapter we test the performance and robustness of the MigWare prototype. Only the setup and results of the experiments are presented. You will find a discussion of the results in the next chapter. The experiment tests are set-up for validation of functionality and the measurement of performance.

6.1 Background

To test the middleware in a realistic environment, we have tried to create the same environment and use the same parameters as in Sense's SiteCom® Rig system, described in section 1.2 *Sense Intellifield's service architecture*, which is the motivation of the conducted experiments.

6.2 Experiment setup

In this section we give a description of the environment where the experiments shall be conducted. The research environment, which is simulating a general scenario where an error occurs, is detailed in this section. The experiment tests are set-up for validation of functionality and the measurement of performance, of the MigWare prototype.

6.2.1 Configuration of the network environment

Three identical servers are connected to a 10 MBit hub. All the servers use Microsoft Windows 2003 Server Enterprise Edition as operating system. The servers contain the following components: Intel® Pentium® 4 CPU 2.80 GHz, with 1.49 GB memory and 10/100/1000Mbit Broadcom NetXtreme Gigabit Ethernet for hp.

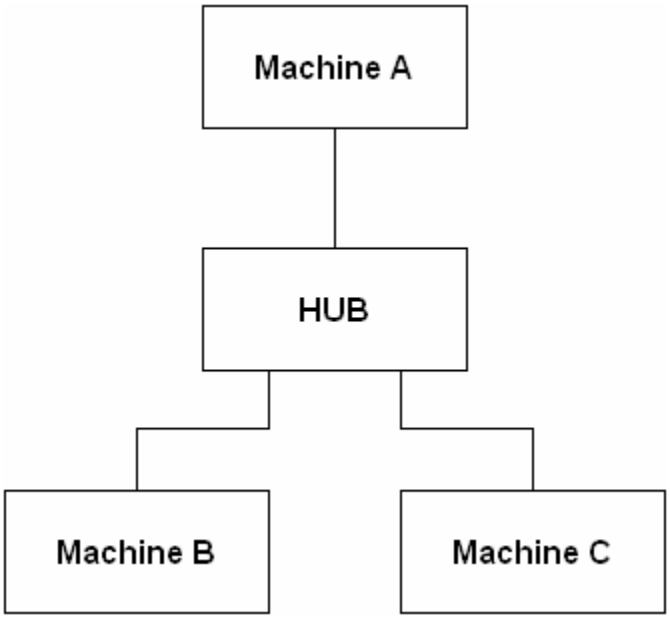

Figure 16: Experiment setup

6.2.2 Scenario

The scenario for the experiments is the same as for Sense's SiteCom® Rig system, seen in chapter 1.2 *Sense Intellifield's service architecture*. Several drill components push their data to one server, through TCP socket connections. The data pushing clients send PSH+ACK packets. These packets contain numbers indicating the current status of the drill components. For example a drilling device updates the server with its temperature once or twice per second. The server only receives data and does not reply with any data, only acknowledging received data with ACK packets.

Let us call the three machines in Figure 16 for Machine A, Machine B and Machine C. Machine A acts as the client containing a client application pushing data every 500ms per connection. Machine B is the primary server that contains a server application receiving data from client connections on Machine A. Machine C is the backup server containing a replica of the application running on server B and the MigWare Middleware application.

All the machines are connected to a hub. A hub broadcasts incoming messages to all its ports. We assume that the machines connected to the hub receive the messages simultaneously. If Machine B receives a packet, the assumption is that Machine C receives the packet at the same time.

6.2.3 Simulating Sense SiteCom® Rig traffic

To create a scenario of normal TCP traffic a socket connection is established between a client and a server. The client continuously sends integer values to the server. The server does not respond to the client, but prints the numbers on screen. This is a typical PUSH scenario where the server only acknowledges received data from the client. This is the case at Sense's SiteCom® Rig. To gather sufficient amount of packet data we assume that the client has to send 40 integer values over each connections. This results in about 80 packets including PSH+ACK and ACK packets.

The client data pusher application on Machine A sends numbers to the data receiver application on Machine B. The data receiver prints the data on screen. Initially the client data pusher sends 40 numbers through a socket connection to the data receiver. This results in 80 packets. These packets have an average packet length of 57 bytes. We have chosen to use the short packet length, because it is very close to Sense's SiteCom® case where drill components push digits in real-time to the SiteCom® server. The 80 packets sent are categorized as normal TCP traffic and is the basis for measuring the latency of TCP (the time it takes to receive and respond to a packet).

When a packet is received at the TCP/IP stack, an acknowledgement is issued to the sender. The time from a packet is received to it is acknowledged can be interpreted as the latency of TCP.

6.2.4 Simulating error

An established TCP connection can in theory be permanent. If a sent packet is acknowledged within a certain time, before a timeout is issued, the connection will persist. If the sender does not receive an acknowledgement, it retransmits

unacknowledged packets. If still no acknowledgement is received, a timeout occurs and the connection is broken. Timeouts differ from two to nine minutes [39]. Repeated retransmissions may trigger timeouts as well [1]. To break a connection one can physically break the link between a client and a server.

We simulate errors as follows. During normal TCP communication between the client, Machine A, and the server, Machine B, the MigWare middleware application monitors the connection. The replicated backup application, Machine C, is ready to receive connection requests (is on stand by). After about 80 packets have been exchanged between the client and the server per connection, the network cable to the primary server, Machine B, is disconnected.

6.2.5 Performance measurement

The figure below illustrates normal TCP communication. A client sends a PSH+ACK packet containing data and it is acknowledge by the server when received. We measure the performance as follows: The time from a PSH+ACK packet is received by the server, to the server acknowledges it with an ACK packet. A network interface card (NIC) timestamps incoming data (PSH+ACK packets) and outgoing acknowledgement (ACK packets). These timestamps can be used to measure the latency of TCP message exchange.

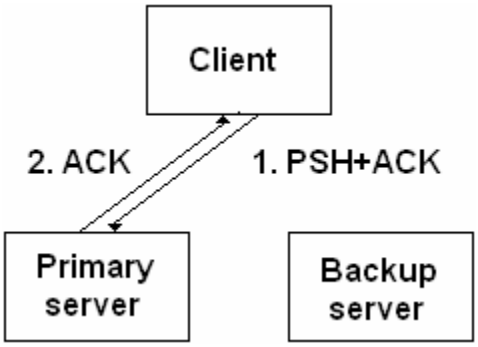

Figure 17: TCP latency

To measure the performance of the MigWare middleware application, four experiments have been executed. 1) How much time TCP use to respond to a received packet (TCP latency). 2) The time it takes to recover *N* connections (Time to recover connections). 3) How much time the middleware use to respond to a packet sent from the client (Total proxy time). 4) And finally the latency of the middleware during proxy mode (Proxy latency).

6.2.6 Robustness measurement

The robustness and reliability of the MigWare has to be measured. The functionality of MigWare has to be verified. The experiments should verify TCP connections persist even if a server failure occurs, and they should be migrated to surviving backup servers, completely transparent to the clients.

We conduct the experiments by checking how many connections are active before we simulate error; see the simulating error chapter above for more details, and how many connections are active after a recovery. If no connections break before or after a recovery is initiated, the robustness test is evaluated as successful because no errors have occurred. The robustness test shall be tested on the prototype handling one to 200 TCP connections at the same time.

6.3 Experiment results

In this section we present the results of conducted experiments to measure the performance and robustness. The experiment tests are set-up for validation of functionality and the measurement of performance, of the MigWare prototype. The experiment setup is described in the previous section, but how the experiments are conducted and the results are presented in the following subsections. At the end of this section, a summary of the results is listed and illustrated in section 6.3.6. A discussion of the results is available in chapter 7.

6.3.1 TCP latency

To capture the added latency of the middleware, one must measure the time it takes to detect a packet and respond to it during proxy mode. Proxy mode is when the middleware is in the state where it detects packets intended for the downed primary server, and forwards those packets to the backup server. It is therefore important to know the latency of normal TCP traffic, which will be measured in this subchapter. To measure the latency of TCP we measure the time it takes for a PSH+ACK packet, t_0 in Figure 18, to be acknowledged by the server with an ACK packet, t_1 in Figure 18. The MigWare middleware application read packets from the network in promiscuous and stores the timestamps of the packets. We calculate the TCP latency by subtracting the timestamp of the ACK with the timestamp of the PSH+ACK, as seen in Figure 17.

Figure 18: Time horizon for acknowledging TCP packets

One thing that has not been tested is an increased packet payload. The packets sent during our experiments have had an average packet length of 57 bytes. TCP packets can have a packet length up to 1460 bytes due to the restrictions of their transmission medium (in this case Ethernet). The reason why we did not test heavier payload is that the middleware does not use or inspect data. Only the first 50 bytes are of interest. Another reason is that small payload is close to Sense's case, described in chapter 1.2 *Sense Intellifield's service architecture*.

As mentioned the payload of a packet can be a possible parameter for testing the efficiency of the middleware, but was discarded due its distance from Sense's case.

Another parameter is packets per second. According to Sense's case, each drill component pushes data once or twice per second. Therefore we have taken no interest in increasing the number of packets sent per connection (the client sends two packet per second). Instead we increase the number of connections which again increases the number of packets sent per second.

For one connection, four packets are sent per second, two in each direction. Two PSH+ACK packets sent by the client and two ACK packets sent by the server. For 200 connections, about 800 packets are sent per second. As mentioned a packet has an average size of 57 bytes in this scenario. This means that 45600 bytes is sent per second or 364800 bits per second (365 kbits/s).

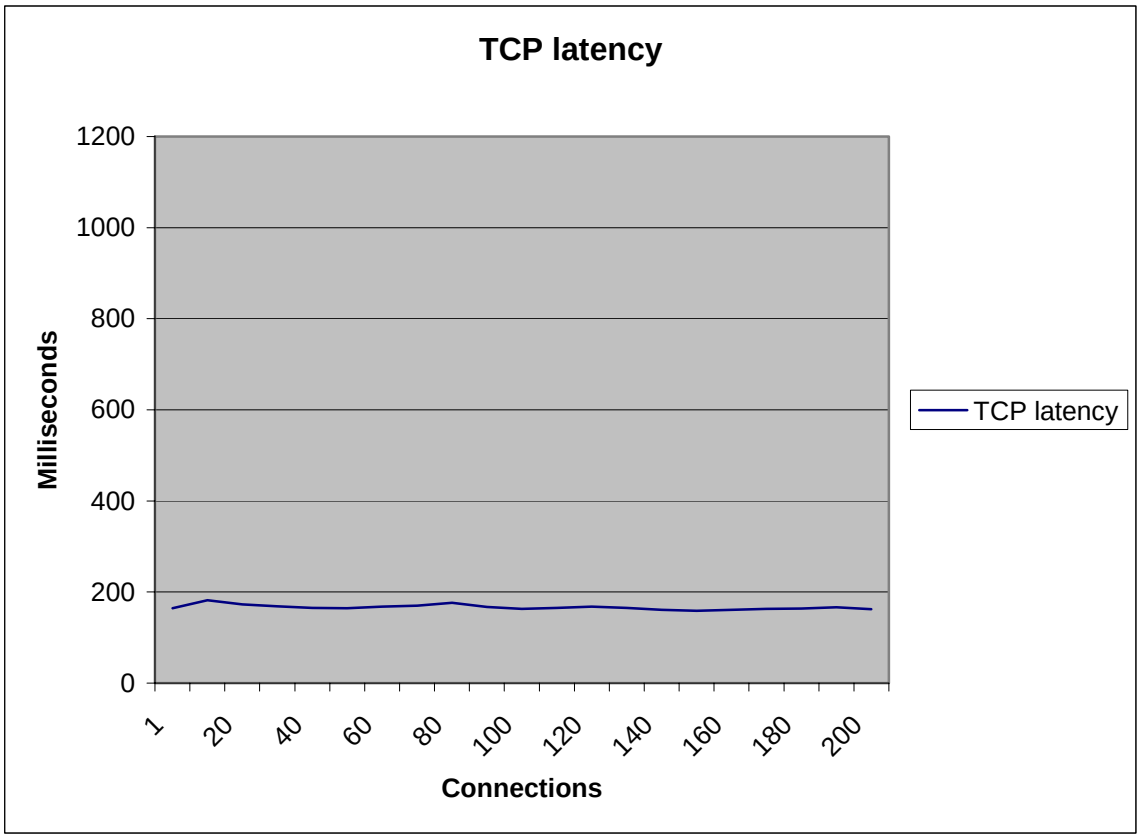

Figure 19: The time the TCP/IP stack use to respond to a PSH+ACK packet

The graph above illustrates the collected data. The X-axis represents the number of active client connections to the server. The connections are normal TCP socket connections. The clients send PSH+ACK data packets and the server responds with ACK packets. The Y-axis represents the number of milliseconds it takes before a PSH+ACK packet is responded to by an ACK packet. Each Y-value represents the average time between a PSH+ACK packet is responded to with an ACK packet per connection. The average is

calculated based on each connection sending about 40 PSH+ACK packets and 40 ACK packets.

The average time of all the connections was 175 milliseconds. The highest average time is 182 milliseconds and is observed at 10 connections. The lowest average time is 158 and is observed at 150 connections. Our observation is that the TCP latency is just below 200 milliseconds and is almost constant.

The time measurement executed in this experiment is quite accurate, because network interface card timestamps received packets. One just has to subtract one UNIX time stamp from another. Both timestamps are created by the same clock. The time measure accuracy is not affected by CPU load.

6.3.2 Time to recover connections

As remember from chapter 5.3.1 *Monitor implementation*, that if the monitor system in MigWare fails to ping the primary server, an alarm is sounded and the recovery of the active connections is initiated. The monitor system pings the primary server once per 500 milliseconds.

When recovery is initiated, all the active connections have to be migrated to the backup server. The figure below illustrates the procedure. To recover a connection, a new one has to be created with the backup server. (1) MigWare sends SYN packets to the replica server application on the backup server, pretending that the crashed primary server is the client. (2) The server application responds through the TCP/IP stack with SYN+ACK packets. (3) The backup server must acknowledge these packets with ACK packets. (P) The client does not know of these events and keeps sending packets. When the handshakes are complete and all the connections are established, the recovery is complete. (P) The MigWare detects the client's packet, rewrites it and forwards it to the backup server. A recovery is initiated when a ping fails and is complete when a new connection is established for each existing connection to the primary server.

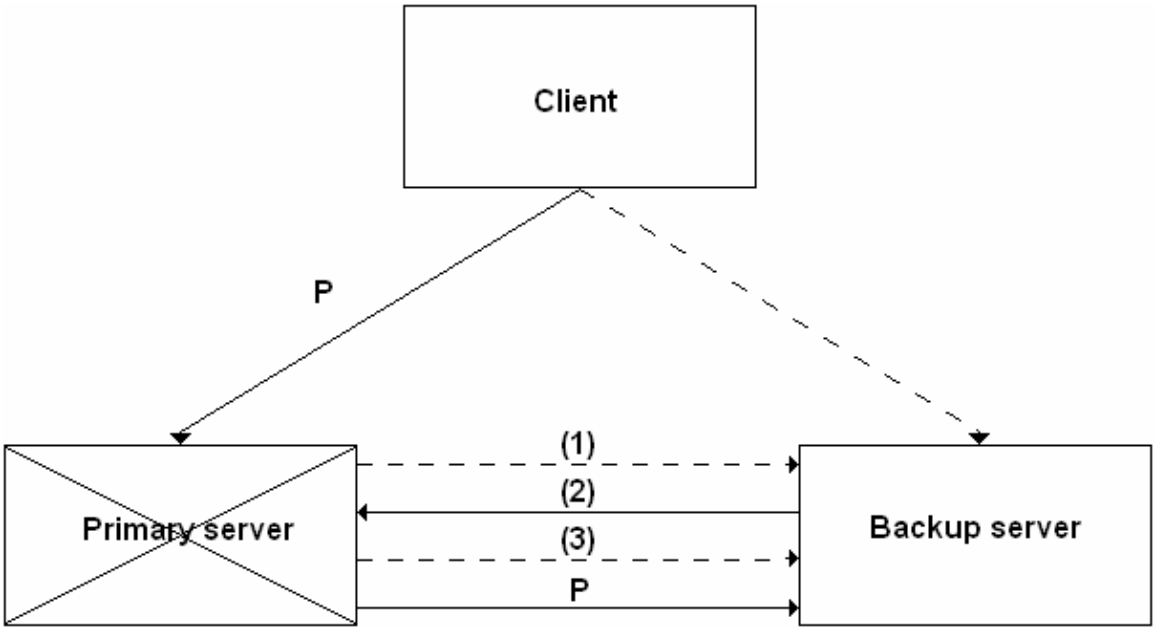

Figure 20: Recovery of one connection

The figure above illustrates a recovery. When the MigWare detects that the primary server is crashed, a new handshake is initiated with the backup server. The time it takes to go through (1), (2), and (3) for each connection is the recovery time. After the recovery packets can be forwarded to the backup server. A measurement of the recovery performance is presented in the following graph:

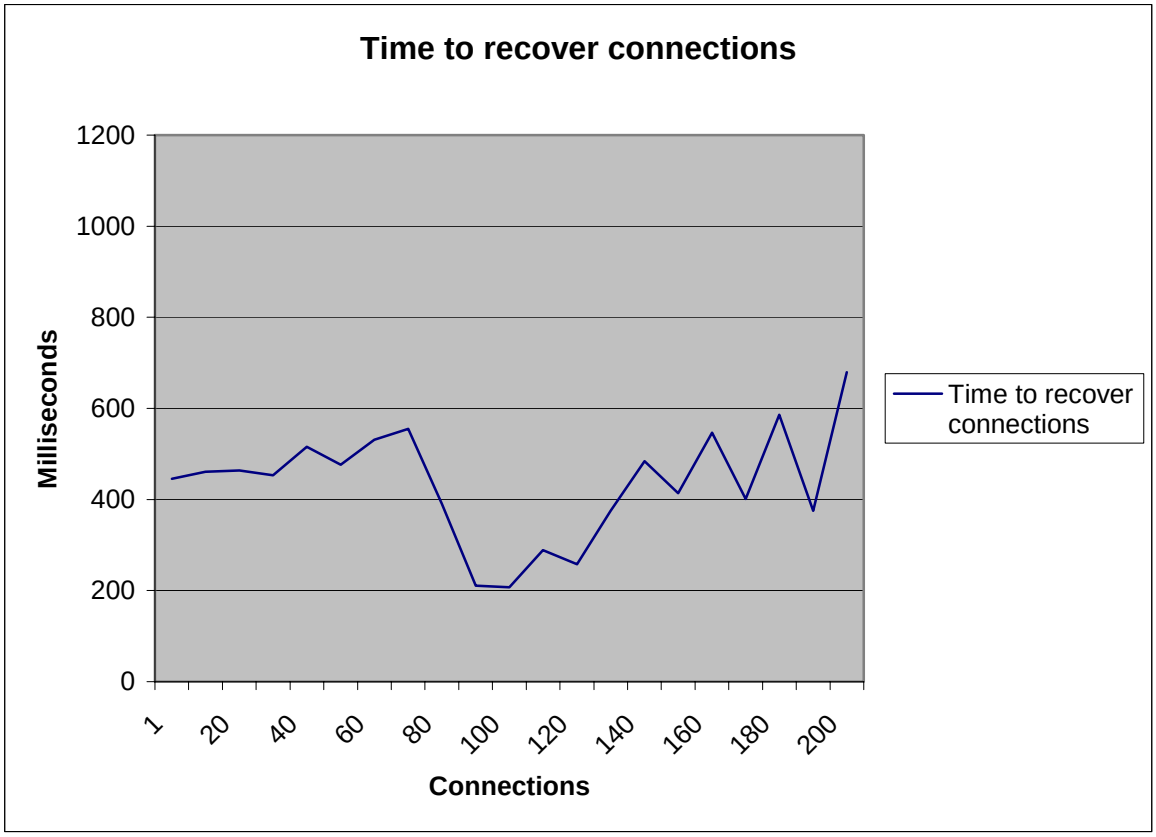

Figure 21: The time MigWare use to recover connections

The highest average recovery time is 680 milliseconds for 200 connections and the lowest average recovery time is 207 milliseconds for 100 connections. The total average recovery time for all the connections is 456 milliseconds, about half a second.

There is a broad variance between the recovery time and the number of connections. We will come back to this in the next chapter.

6.3.3 Total proxy time

To be able to measure the overall latency of the MigWare middleware application, one can measure the time it takes for the system to respond to packets sent from clients. This measurement follows the same principles as the measurement of the TCP latency.

The middleware solutions packet rewriting procedure is described in the figure below. The middleware sniffs packets in promiscuous mode in the sub network. A hub connects the machines and broadcasts packets. (1) During proxy mode, the connection tracker component detects a packet sent form the client intended for the primary server. (2) MigWare rewrites the packet and sends it to backup server pretending that the crashed server is the source. (3) The TCP/IP stack acknowledges the received packet. (4) MigWare detects that packet on the network and rewrites it and sends it to the client, pretending that it is the primary server who acknowledges the sent packet.

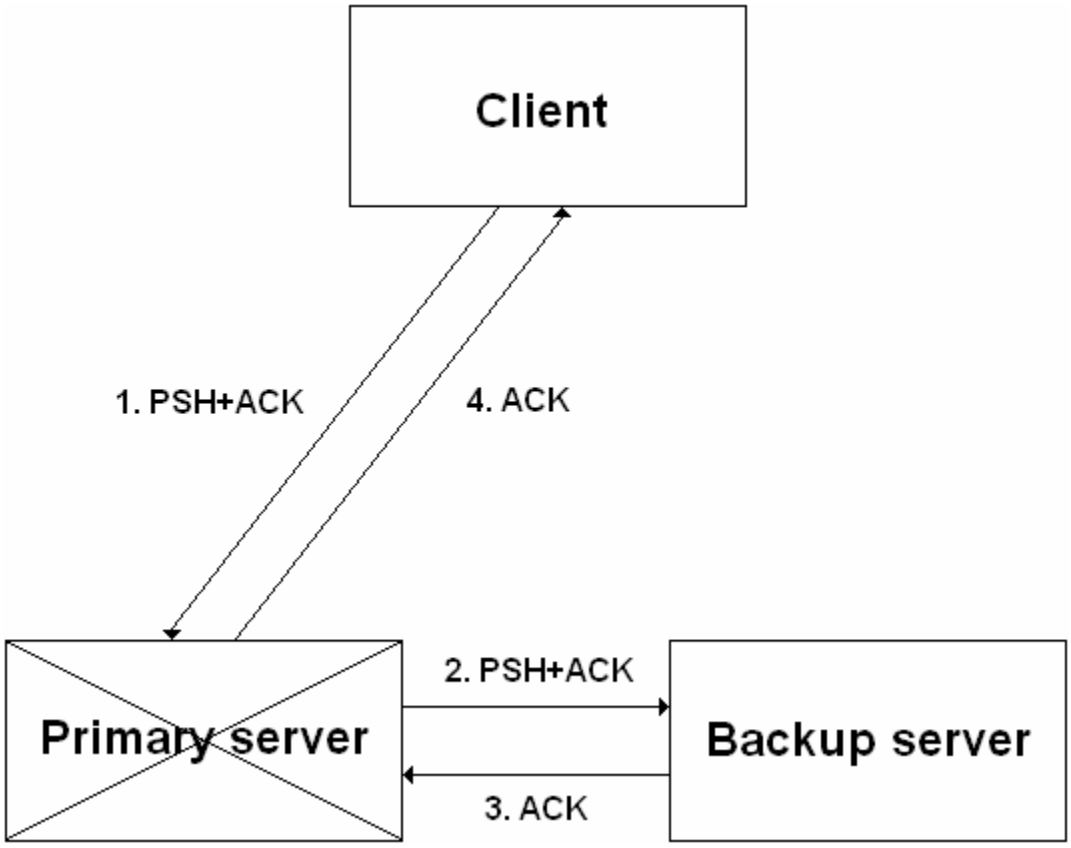

Figure 22: MigWare's message exchange

The time it takes for MigWare to detect a PSH+ACK packet from a client (1), to the MigWare responds to it with an ACK packet (4) is the variable in this experiment. With these results, one can measure the latency of the middleware by subtracting the total proxy time with the TCP latency. We will come back to this later in this chapter.

For now we will measure the total time from the client sends a packet, t_0 in , to it receives an acknowledge from the backup server through MigWare, t_4 in, still believing that the response comes from the primary server.

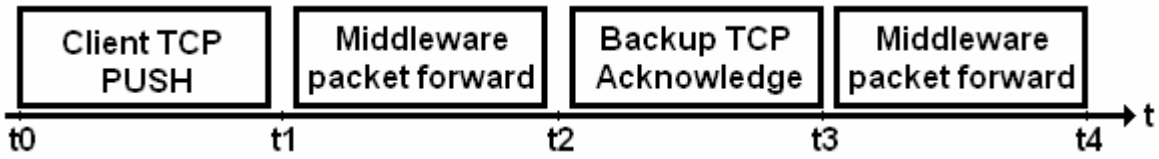

Figure 23: The total time horizon for the MigWare in proxy mode

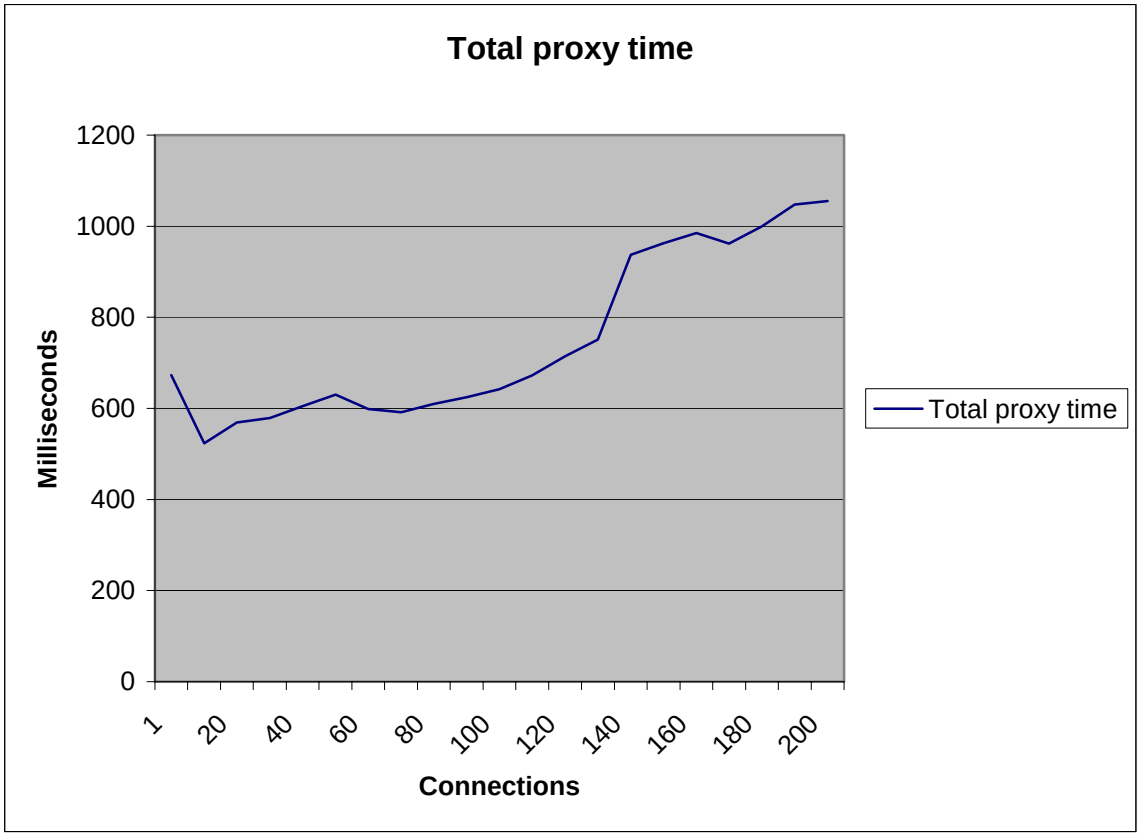

Figure 24: The time the MigWare use to rewrite and forward packets

The highest average total time is 1055 milliseconds when handling 200 connections and the lowest average time is 523 milliseconds when handling 10 connections. The average total time for all connections is 787 milliseconds, or ¾ of a second.

6.3.4 Proxy latency

This experiment measures the latency of the MigWare Middleware application. The latency represents the time delay added to normal TCP traffic. To find the delay, one must use the following equation:

'Proxy latency' = 'Proxy total time' – 'TCP latency'

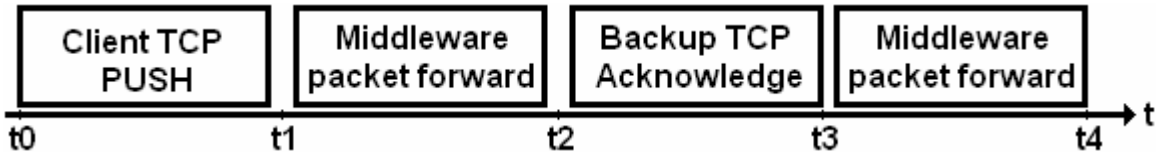

Figure 25: The time horizon for the MigWare in proxy mode

'Proxy latency' = (t_1 to t_4) – ((t_1, t_2) + (t_3, t_4))

The equation is illustrated in the figure above. The client sends a PSH+ACK packet intended for the primary server; t_0 to t_1 in Figure 25, not knowing it has crashed. The middleware detects the packet, rewrites it and forwards it to the backup, t_1 to t_2 in Figure 25. The backup server responds to the received packet by acknowledging, t_2 to t_3 in Figure 25. The middleware detects the packet again, rewrites it and forwards it to the backup, t_3 to t_4 in Figure 25.

The proxy latency illustrates the time added due to the middleware's overhead and can be seen in Figure 25 as t_1 to t_2 and t_3 to t_4.

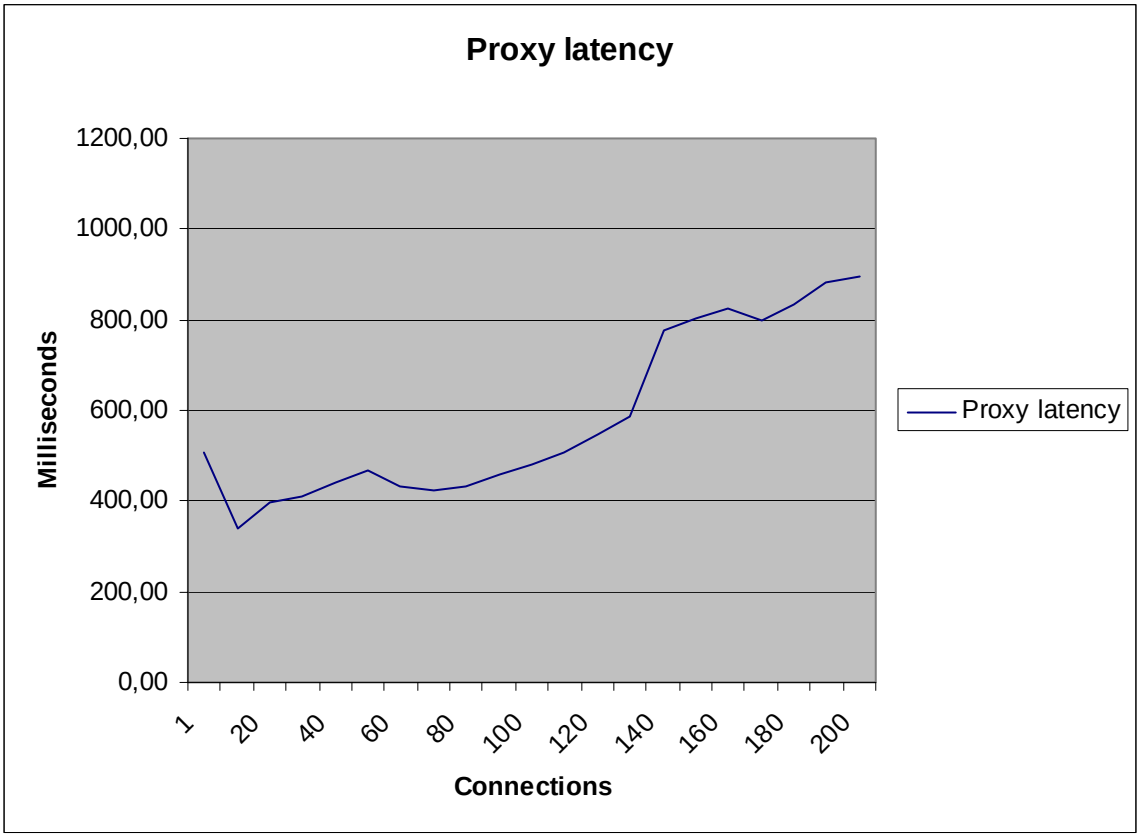

Figure 26: the latency of the MigWare Middleware Application

The highest average value is 893 milliseconds at 200 connections and the lowest value is 396 milliseconds at 20 connections. The total average latency for all connections is 612 milliseconds, just above half a second.

6.3.5 Robustness test

No connections broke before or after the error simulation. No errors have been detected during the simulation of Sense's SiteCom® Rig traffic, during connection recovery, or during proxy mode.

6.3.6 Summary

This is a summary of all the conducted experiments represented in the same graph in the same scale.

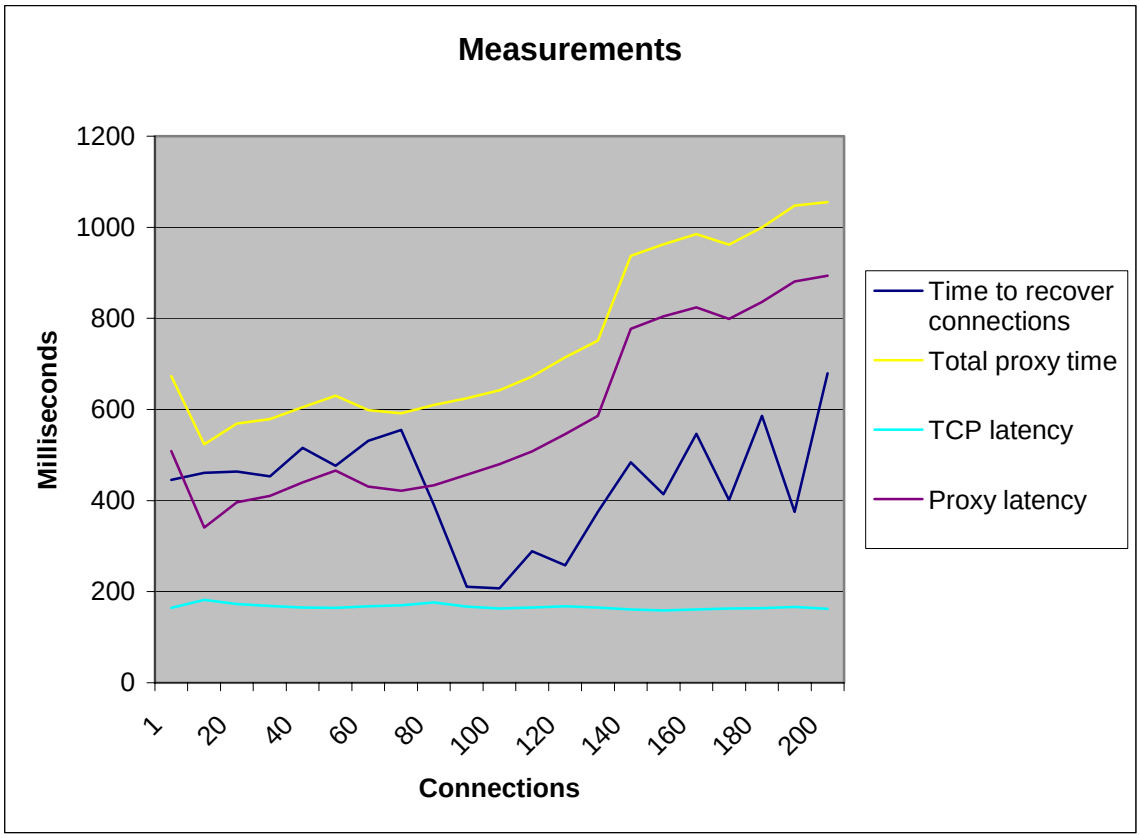

Figure 27: The experiment results

The total average recovery time for all the connections is 456 milliseconds, about half a second. The average total time for all connections is 787 milliseconds, or ¾ of a second. The average time of all the connections was 175 milliseconds, just below 200 milliseconds. The total average middleware latency for all connections is 612 milliseconds. No errors occurred during the experiments.

7 Discussion

In section 7.2 we present a discussion of the experiment results in chapter 6, and in section 7.3 we compare our prototype to similar existing one. In section 7.4 critical remarks to the experiments and the results is addressed. In chapter 7.4 we present a recomandation for further work.

7.1 Introduction

This section discusses the experiment results and compares the MigWare middleware application to Jeebs, due to the similarity of the two solutions [39]. Both migrates TCP connections without modifying the client or server application. The difference in Jeebs and MigWare is that Jeebs has to make changes to the backup server's kernel by modifying the TCP/IP stack. MigWare does not make any changes to any operating systems, kernels, daemons, TCP/IP stacks or applications. The MigWare middleware and WinPcap has to be installed on the backup server. Therefore, MigWare is an add-on solution.

A comparison between the two solutions will present their strength and weaknesses. At the end of this chapter, a recommendation for when to use which solution is available.

7.2 Experiment results

In chapter 6, four experiments were conducted; **Time to recover connections** in section 6.3.2, **Total proxy time** in section 6.3.3, **TCP latency** in section 6.3.1, and **Proxy latency** in section 6.3.4. Only two of the experiments will be discussed because they are the most relevant. The total proxy time and the TCP latency are conducted to calculate the proxy latency.

7.2.1 Time to recover connections

The average recovery time for all connections is 456 milliseconds, just below half a second. Comparing this with the Jeebs system, this is a dramatically improvement [39]. Jeebs has an average recovery time of 120000 milliseconds (2 minutes) for 100 connections. MigWare has an average recovery time of 207 milliseconds (1/5 of a second) for 100 connections.

The results vary. A probable reason is that the accuracy of the measurements is not adequate. We have used the computer's clock to capture the time of recovery initiation and finalization (The C# static `DateTime.Now` object). During heavy CPU load the time retrieval might be delayed, in contrary to the use of UNIX timestamps. Another reason is the use of threads without thread pooling. However we can say that the recovery time is below 678 milliseconds, because the maximum recovery time is 678 for 200 connections which is the maximum number of connections. The rest is below 678 milliseconds.

Why is there such a big difference in the results? One reason might be the architecture. The Jeebs's monitor is much more advanced; monitoring several services instead of just

pinging the server to detect if it has crashed or not. It takes much more time to verify that a single service has crashed than checking if a ping succeeds or not. MigWare has a much simpler monitoring system than Jeebs and has therefore less latency. One of MigWare's weaknesses is that if a service fails on the server, but not the server itself, than MigWare will not migrate any connections. MigWare is the most efficient recovery system if the server accidentally crash or is disconnected.

Another reason why Jeebs cannot match MigWare in recovery time is that Jeebs stores state information and critical packets in a database on a black box monitor [39]. During a recovery, Jeebs has to play back all the packets in the same order to recover a connection and its state. This adds more latency.

7.2.2 Proxy latency

When a recovery has been issued, the middleware rewrites packets to "fool" the TCP protocol to prevent the connections to break/time-out. The time it takes to detect a packet sent from the client to the primary server, and to send a reply from the backup server to the client is measured. This time measurement can be used to calculate the latency added by the middleware.

The total average latency for all connections is 612 milliseconds, just above half a second. This is severe penalty comparing it to the latency of normal TCP latency which is below 200 milliseconds, according to our measurements. The MigWare adds on average 612 milliseconds in latency to the TCP protocol.

The time measurement executed in this experiment is quite accurate in contrary to the recovery time measurement, because the network interface card timestamps received packets. One just has to subtract one UNIX time stamp from another. Both timestamps are created by the same clock.

Comparing this to Jeebs, who has made changes to the kernel by modifying the TCP/IP stack, the MigWare latency is enormous. Once Jeebs has recovered a connection, no latency is added to the TCP.

7.2.3 Robustness testing

No errors are detected during the experiment runs. This does not prove the real robustness of the prototype, but it does not falsify the claim that MigWare is robust and reliable.

If no TCP connections break before or after a server failure, one can conclude that the failure is masked and that the clients do not detect the failure. If this is the case, this is an evidence of the added robustness to TCP by using the MigWare middleware. The functionality of MigWare has been verified. TCP connections persist even if a server failure occurs. The connection is simply migrated to a surviving backup server, completely transparent to the clients.

Further robustness testing could have been applied to validate the logic of the prototype. The robustness tests conducted are quite simple and do not prove that the prototype is flawless, but it shows robustness to some extent.

7.3 When to use Jeebs and MigWare

Jeebs and MigWare are two different solutions to the same problem. Both are intended to increase the availability of TCP-based services and to mask failure. The two solutions have quite different performance.

Jeebs recovers states and completely migrate connections without adding any latency to the TCP. The penalty is that there has to be made changes to the operating system of the backup server, forcing the backup server to run on Linux. For Sense who runs Microsoft .Net applications that are dependant of the Windows operating system, it is impossible to adapt to this solution. For more general purposes, Jeebs is superb, recovering the states of TCP-services without adding any latency to the TCP.

MigWare offers an add-on solution providing fast recovery without modifying any existing operating system, kernel, TCP/IP stack, daemon or applications. The cost of the solution is added latency to the TCP due to the packet rewriting and forwarding. If this is the case, like in Sense's case, or there is a need for fast recovery, this is the way to go. The prototype is developed with .Net to run on the Windows platform, but it can be developed platform independent as well. The MigWare prototype does not support application state recovery, but it has not ruled it out.

When it comes to a comparison of robustness, we come up short. Jeebs presented no robustness data or robustness experiment for comparison. Jeebs has proven to migrate 120 connections without failure, while MigWare has successfully migrated 200 connections. With success, we mean that no errors were reported during experiments.

7.4 Further work

The performance of the prototype is adequate, but can be improved. Efficient use of threads and pointers can improve the performance. Instead of using WinPcap to sniff packets in promiscuous mode, miniport drivers can be developed to improve the efficiency. Keep in mind that this is a prototype demonstrating the concept of middleware connection migration.

The middleware solution can be developed platform independent, using for example C++ or Java. MigWare is developed in C# .Net so it could be adopted by Sense who runs everything on the Windows platform. The prototype is developed for the Microsoft Windows platform, but the technique and architecture is platform independent.

There has been no use of formal methods to verify the prototype. It can verify the prototype's algorithms and provide quality assurance.

7.5 Critcal remarks

Are the results well-founded? The results are well-founded due to the simulation of a real case, the SiteCom® Rig system. This case is also quite common in the IT-industry. The experiments conducted are very close to the simulated reality.

Are the results general? The results are general because of the simulation of a common push case. One can critique the results because it lacks variance of the packet lengths. However, several PUSH cases do not send large amount of data.

Are the results useful? The results are useful, because they give an accurate indication of the performance of an application level middleware solution. The results can be used to decide whether or not to implement this solution to increase the availability of TCP-based services

8 Conclusion

Until TCP-based migration solutions are available on the hundreds of millions of existing systems, there will remain a need for client-transparent migration. The MigWare middleware system demonstrates how our technique can be deployed in a simple manner, without requiring changes to existing site architecture. The simplicity and immediate applicability of the technique proposed in this report make it attractive for adoption in a number of cases.

Results

Empirical data show that a middleware for TCP connection migration is possible and feasible. This is demonstrated by the MigWare prototype which migrate TCP connections from a primary server to a backup server without the client ever knowing about it. The primary server was disconnected from the network while active connections were processing real-time data. All the connections were migrated to the backup server containing a replica of the TCP-based service on the primary server, completely transparent to the clients.

The middleware is simply installed on a backup server where it monitors the connections to the primary server. No modifications to the operating systems, TCP/IP stacks, kernels, client application or server application is needed. MigWare is true to the internet philosophy, where the intelligence is supposed to be implemented in the computers, while keeping the network nodes as dumb as possible. We regard MigWare as a network node, even though it is a middleware running on the backup server.

No errors were detected during the experiments and no connections broke before or after a recovery. Empirical data verifies the functionality of the prototype. The prototype proves to add robustness to TCP by detecting server failure and migrate connections, completely transparent to the client. The time to migrate active connections from a crashed server to a backup server is satisfactory. The result is better than similar solutions, using only 456 milliseconds in average to recover up to 200 connections. The performance of the prototype is adequate, adding a latency of 612 milliseconds in average and handling up to 200 connections simultaneous.

Consequences

MigWare can replace expensive third-party load balancers and clustering solutions that ensure high availability of TCP-based services. In addition MigWare adds robustness to TCP and, unlike many third-party solutions, connections persists when servers fail. This can give companies economical advantages by not being dependant of expensive, limiting, and existing third-party solutions.

Further work

No attempts to migrate application states were discussed or researched, and no effort was issued to reduce the latency or overhead of the prototype. However, there are many cases that do not need to include state migration, like Sense's SiteCom® Rig system. Simple

measures can be taken to improve the performance; like thread pooling, miniport driver implementation, and the use of pointers.

Critical Remarks

The results are well-founded due to the simulation of a real case, the SiteCom® Rig system. This case is also quite common in the IT-industry. The experiments conducted are very close to the simulated reality. The results are general because of the simulation of a common push case. One can critique the results because it lacks variance in the packet lengths. However, several PUSH cases do not send large amount of data. The results are useful, because they give an accurate indication of the performance of an application level middleware solution. The results can be used to decide whether or not to implement this solution to increase the availability of TCP-based services.

Acknowledgements

We would like to thank Rune Skarbø, at Sense Intellifield, for introducing us to the load balancing challenge which led us to our master thesis. We also would like to thank *cand. scient.* Ole-Christoffer Granmo, at Agder University College, for his theoretical guidance and advisory of our report. We are also grateful for the feedback on our report form *the head of study*, at the department of Information and Communication technology at Agder University College, Stein Bergsmark. We are also in great gratitude to the Agder University College library for their service. We would like to thank them for contacting Professor Andrew Burt, at the University of Denver, and giving us access to his submitted Jeebs paper [39].

References

[1] IETF RFC793, "Transmission Control Protocol (TCP)"

[2] Kopetz, H., Verissimo, P.: "Real Time and Dependability Concepts." In Mullender, S. (ed.), *Distributed Systems*, pp. 411-446. Wokingham: Addison-Wesley, 2nd ed., 1993.

[3] Cristian, F.: "Understanding Fault-Tolerant Distributed Systems." *Commun. ACM*, vol. 34, no. 2, pp. 56-78, Feb. 1991.

[4] Hadziliacos, V., Toueng, S.: "Fault-Tolerant Broadcasts and Related Problems." In Mullender, S. (ed.), *Distributed Systems*, pp. 97-145. Wokingham: Addison-Wesley, 2nd ed., 1993.

[5] Johnson, B.: "An Introduction to the Design and Analysis of Fault-Tolerant Systems." In Pradhan, D.K. (ed.), *Fault-Tolerant Computer System Design*, pp. 1-87. Upper Saddle River, NJ: Prentice Hall, 1995.

[6] Guerraoui, R., Schiper, A.: "Software-Based Replication for Fault Tolerance." *IEEE Computer*, vol. 30, no. 4, pp. 68-74, Apr. 1997.

[7] Schneider, F.: "Implementing Fault-Tolerant Services using State Machine Approach: A Tutorial." *ACM Comput. Surv*., vol. 22, no. 4, pp. 299-320, Dec. 1990.

[8] Thomas, R.: "A Majority Consensus Approach to Concurrancy Control for Multiple Copy Databases." *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180-209, June 1979.

[9] Gifford, D.: "Weighted Voting for Replicated Data." *Proc. Seventh Symp. Operating Systems Principles*. ACM, 1979. pp. 150-162. Cited on page 343.

[10] Rodrigues, L., Fonseca, H., Verissimo, P.: "Totally Ordered Multicast in Large-Scale Systems." *Proc. 16th Int'l Conf. on Distributed Computing Systems*. IEEE, 1996. pp. 503-510.

[11] Tanenbaum, A.: *Computer Networks*. Englewood Cliffs, NJ: Prentice Hall, 3rd ed., 1996.

[12] Chang, J., Maxemchunk, N.: "Reliable Broadcast Protocols." *ACM Trans. Comp. Syst*., vol. 2, no. 3, pp. 251-273, Aug. 1984.

[13] Townsley, D., Kurose, J., Pingali, S.: "A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols." *IEEE J. Selected Areas Commun*., vol. 15, no. 3, pp. 398-407, Apr. 1997.

[14] Levine, B., Garcia-Luna-Aceves, J.: "A Comparison of Reliable Multicast Protocols." *ACM Multimedia Systems Journal*, vol. 6, no. 5, pp. 334-348, 1998.

[15] Floyd, S., Jacobson, V., McCanne, S., Liu, C.-G., Zhang, L.: "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing." *IEEE/ACM Trans. Netw.*, vol. 5, no. 6, pp. 784, 803, Dec. 1997.

[16] Hofmann, M.: "A Generic Concept for Large-Scale Multicast." In Plattner, B. (ed.), *Broadband Communications – Networks, Services, Applications, Future Directions*, vol. 1044 of *Lect. Notes Comp. Sc.*, pp. 95-106. Berlin: Springer-Verlag, 1996.

[17] Buretta, M.: *Data Replication: Tools and Techniques for Managing Distributed Information*. New York: John Wiley, 1997.

[18] Herlihy, M., Vinonski, J.: "Linearizability: A Correctness Condition for Concurrent Objects." *ACM Trans. Prog. Lang. Syst.*, vol. 12, no. 3, pp. 463-492, July 1991.

[19] Attiya, H., Welch, J.: "Sequential Consistency versus Linearizability." *ACM Trans. Comp. Syst.*, vol. 12, no. 2, pp. 91-122, May 1994.

[20] Lamport, L.: "How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs." *IEEE Trans. Comp.*, vol. C-29, no. 9, pp. 690-691, Sept. 1979.

[21] Hutto, P., Ahamad, M.: "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories." *Proc. Tenth Int'l Conf. on Distributed Computing Systems*. IEEE, 1990. pp. 302-311.

[22] Lipton, R., Sandberg, J.: "PRAM: A Scalable Shared Memory." Technical Report CS-TR-180-88, Princeton University, Sept. 1988.

[23] Dubois, M., Scheurich, C., Briggs, F.: "Adaptive Leases: A strong Consitency Mechanism for the World Wide Web." *Proc. 19th INFOCOM Conf.* IEEE, 2000. pp. 834-843.

[24] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gaupta, A., Hennessy, J., "Memory Multiprocessors." *Proc. 17th Ann. Int'l Symp. On Computer Architecture.* ACM, 1990. pp. 15-26.

[25] Keleher, P., Cox, A., Zwaenepoel, W.: "Lazy Release Consistency." *Proc. 19th Ann. Int'l Symp. On Computer Architecture.* ACM, 1992. pp. 13-21.

[26] Bershad, B., Zekauskas, M., Sawdon, W.: "The Midway Distributed Shared Memory System." *Proc. COMPON*. IEEE, 1993. pp. 528-537.

[27] Bershad, B., Zekauskas, M.: "Midway: Shared Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors." Technical Report CMU-CS-91-170, Carnegie Mellon University, Sept. 1991.

[28] Terry, D., Demers, A., Petersen, K., Spreitzer, M., Theimer, M., Welsh, B.: "Session Guarantees for Weakly Consistence Replicated Data." *Proc. Third Int'l Conf. on Parallel and Distributed Information Systems*. IEEE, 1994. pp. 140-149.

[29] Terry, D., Petersen, K., Spreitzer, M., Theimer, M.: "The Case of Non-transparent Replication: Examples from Bayou." *IEEE Data Engineering*, vol. 21, no. 4, pp. 12-20, Dec. 1998.

[30] Birrell, A, Nelson, B.: "Implementing Location Independent Invocation." *IEEE Trans. Comp. Syst*. Vol. 2, no. 2, pp. 39-59, Feb. 1984.

[31] Kermarrec, A., Kuz, I., Van Steen, M., Tanenbaum, A.: "A Framework for Consitent, Replicated Web Objects." *Proc. 18th Int'l Conf. on Distributed Computing Systems*. IEEE, 1998. pp. 276-284.

[32] Chawathe, Y., Brewer, E.: "System Support for Scalable and Fault Tolerant Internet Services." *Proc. Middleware '98*. IFIP, 1998. pp. 71-88.

[33] Ozsu, T., Valduriez, P.: *Principles of Distributed Database Systems*. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 1999.

[34] Sheth, A. P., Larson, J. A.: "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Comput. Surv*., vol. 22, no. 3, pp. 183-236, Sept. 1990.

[35] Gwertzman, J., Seltzer, M.: "The Case for Geographical Push-Caching." *Proc. Fifth Workshop Hot Topics in Operating Systems*. IEEE, 1996. pp. 51-55.

[36] Forouzan, B.: *TCP/IP Protocol Suite*. International edition. McGraw-Hill, 2nd ed., 2003. ISBN 0-07-119962-4.

[37] Tanenbaum, A. S., Van Steen, M.: *DISTRIBUTED SYSTEMS – PRINCIPLES AND PARADIGMS*. Vrije Universiteit, Amsterdam, the Netherlands. Prentice Hall, 2002. ISBN 0-13-088893-1.

[38] WinPcap, http://winpcap.polito.it, last modified Monday, May 3, 2004, 17:41. The reference was added May 6, 2004.

[39] A. Burt, S. Narayanappa and R. Thurimella. Techniques for Client-Transparent TCP Migration. Submitted 2003.

[40] H. Wu, A. Burt, R. Thurimella. Making Secure TCP Connections Resistant to Server Failures. Department of Computer Science, University of Denver, Denver, CO 80208, USA.

[41] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In Proc., ACM SIGCOMM '98, Sep. 1998.

[42] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Cluster-based Scalable Network Services. In Proc. ACM SOSP '97, Oct. 1997.

[43] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware Request Distribution in Cluster-based Network Servers. In Proc. ASPLOS '98, Oct. 1998.

[44] High-Availability Linux Project, http://linux-ha.org/, last updated: 2 March, 2004. The reference was added May 13, 2004.

[45] BIG-IP® LoadBalancer Limited 520, http://www.f5.com/f5products/bigip/LB520/ The referance was added May 13, 2004.

[46] Microsoft Windows 2000 and 2003 Server – Network Load Balancing Technical Overview, http://msdn.microsoft.com. The referance was added May 13, 2004.

[47] R. Zhang, T. Abdelzaher, J. A. Stankovic. Efficient TCP Connection Failover in Server Clusters. Department of Computer Science, University of Virginia, Charlottesville, VA 22904, USA.

[48] A. C. Snoeren, D. G. Andersen, H. Balakrishnan. Fine-Grained Failover Using Connection Migration. MIT Laboratory for Computer Science, Cambridge, MA 02139, USA.

[49] M. Marwah, S. Mishra, C. Fetzer. TCP Server Fault Tolerance Using Connection Migration to a Backup Server. Department of Computer Science, University of Colorado, Campus Box 0430 Boulder, CO 80309-0430. AT&T Labs-Research, 180 Park Avenue Florham Park, NJ 07932, USA.

[50] R. Ekwall, P. Urbán, A. Schiper. Robust TCP Connections for Fault Tolerant Computing. École Polytechnique Fédérale de Lausanne (EPFL), Distributed Systems Laboratory CH-1015 Lausanne, Switzerland.

[51] H. Bakke, T. Meland. *Appendix A.* Department of Information and Communication Technology, Agder University College, Grooseveien 36, 4876 Grimstad, Norway.