Design and Implementation of TCP and SCTP Connection Migration Function for an IP High Availability Framework

Joacim Häggmark

Luleå University of Technology
MSc Programmes in Engineering
Computer Science and Engineering
Department of Computer Science and Electrical Engineering
Division of Computer Communication
Design and Implementation of TCP and SCTP Connection Migration Function for an IP High Availability Framework.

Joacim Häggmark

October 25, 2007
Abstract

This thesis was carried out at Operax AB in Luleå. Today one of their products runs on servers which use 1+1 active/standby redundancy. This thesis is about replacing the current software for IP fail-over by constructing a new one which also supports migration of TCP and SCTP states.

Initially existing solutions for TCP-state transfer were studied to see if they could be built on but after the investigation was done we decided to design and develop an entirely new implementation. A basic design was created and TCP together with SCTP was studied to find which state-information that needed to be shared between the nodes and also which header-data that was needed to be changed after a connection migration.

The final design is a daemon which runs on two or more server nodes. The daemon listens to two separate IP-numbers, one service-ip and one for being able to hand-over connections to the other node when doing a switch-over. OpenAIS is used to provide heartbeat monitoring and sharing of state-information. An application API was also created so a service-application can get information if a new socket has been moved, if a switch-over is going to happen and to acknowledge a move and send state-specific data to the standby node. When a switch-over is made, all new connections will directly be passed on to the standby node and then all service-applications are notified. When the service-applications are ready to transfer the state, they send an acknowledgement back and then all packets for that connection will be redirected to the new server-node. When all connections have been moved the service-ip is moved to the standby node.

The main difference between this work and previous published solutions is that it supports SCTP-state transfers and that it’s written in user-space and therefore portable between different operating systems.
Preface

First of all I would like to thank Johan Larsson at Operax AB for offering me this thesis in the first place. I found it to be a good computer communications thesis which was both challenging and interesting. I also thank Anders Torger, also at Operax who together with Johan were my supervisors. I also wish to mention the importances of both Skellefteå Datorförening and Ludd for being places were most of my computer skills and interest has been developed. Thanks to my examiner Evgeny Osipov and to Viktor Leijon for valuable comments on the report.

Finally I would like to thank all my friends and family for their support and give special thanks to Maria.
Contents

Preface .................................................. 1

1 Introduction ........................................... 4
  1.1 Background ........................................ 4
  1.2 Purpose ............................................ 4
  1.3 Requirements ...................................... 5
  1.4 Terminology ....................................... 5
    1.4.1 Switch-over ................................... 5
    1.4.2 Fail-over ...................................... 5
    1.4.3 Checkpoints .................................... 5
    1.4.4 TCP ............................................ 5
    1.4.5 SCTP ........................................... 6

2 Analysis ............................................... 7
  2.1 Combining other solutions ......................... 7
  2.2 Basic idea ......................................... 7
  2.3 Moving TCP ........................................ 8
    2.3.1 Sequence numbers .............................. 8
    2.3.2 Flow control .................................. 9
    2.3.3 Maximum segment size ......................... 10
    2.3.4 Timestamps .................................... 10
    2.3.5 TCP Selective Acknowledgement ............... 10
  2.4 Moving SCTP ....................................... 10
    2.4.1 Multi-homing .................................. 10
    2.4.2 Initiate Tag ................................... 10
    2.4.3 Transmission Sequence Number ................. 11
    2.4.4 Number of Streams ............................. 11
    2.4.5 Stream Sequence number ....................... 11
    2.4.6 Advertised Receiver Window Credit ............ 11
    2.4.7 Selective Acknowledgement (SACK) ............. 11

3 Design ................................................ 12
  3.1 High level ........................................ 12
    3.1.1 Checkpoints ................................... 14
    3.1.2 States ......................................... 14
    3.1.3 TCP ............................................ 14
    3.1.4 SCTP ........................................... 14
  3.2 Application API .................................. 15
4 Implementation
4.1 Startup ............................................. 17
4.2 Processing packets thread ........................... 17
  4.2.1 TCP ............................................. 17
  4.2.2 SCTP ............................................. 18
4.3 Control socket thread ............................... 18
4.4 Cleaning thread .................................... 18
4.5 Checkpoint writing thread .......................... 18
4.6 External libraries ................................... 18
  4.6.1 OpenAIS ......................................... 18
  4.6.2 Libnet .......................................... 19
  4.6.3 Libpcap .......................................... 19
4.7 Security ............................................ 19
4.8 Limitations ......................................... 19
4.9 Performance ........................................ 19

5 Conclusions ........................................ 21
  5.1 Conclusions ........................................ 21
  5.2 Future work ....................................... 21

Bibliography .......................................... 22
Chapter 1

Introduction

1.1 Background

Operax designs highly resilient server software which uses 1+1 active/standby redundancy. The active instance serves client requests and continuously mirrors its state to the standby instance, so it can without delay take over from the active in the event of a failure or a planned switch.

The clients communicate with the server using different type of upper layer protocols but the underlying transport protocols are TCP, SCTP or UDP.

![Diagram of 1+1 Active/Standby System]

The clients need only to connect towards a single IP address, and does not know about the 1+1 resilient server arrangement. When a switch-over or fail-over occurs, the IP address is simply moved from the old active to the new active. This is realized through gratuitous ARP, such that the resilient switch in front of the nodes is re-programmed to send packets towards the new active.

1.2 Purpose

The purpose for the thesis is to replace the current third-party software for IP fail-over by constructing a new middleware which provides better transparency
towards the clients then the current solution.

1.3 Requirements

The solution should provide IP fail-over with TCP and SCTP state transfer. Switch-overs should be 100% transparent for clients and fail-overs as well if the client does not have any pending requests. There is only a need to transfer connections in fully connected state.

The solution should integrate towards SA Forum AIS [1], which provides logic for detecting node failures and steering active/standby states. The SA Forum AIS can also be used to share state between nodes.

Ideally the solution is implemented in user-space only and is portable between Linux and Solaris. But if kernel programming is required, a Linux-only solution is acceptable. Modifications to the kernel should however be kept to a minimum, and preferably employ the standard kernel module system.

A suitable API is designed for the applications to use, to detect the difference between new and taken-over connections, and means to associate taken-over connections with mirrored upper layer protocol session state. The middle-ware should not affect flow control of the transport protocols, which instead is controlled by the receiving applications.

1.4 Terminology

1.4.1 Switch-over

Switch-over is a planned switch from a primary server to a standby server. A switch-over is normally initiated by a human. Since both the primary and secondary server is running, a switch-over can be made in different steps.

1.4.2 Fail-over

Fail-over is the process of an unplanned switch from a primary server to a standby server. It normally occurs automatically when the primary server crashes.

1.4.3 Checkpoints

In this thesis when we talk about checkpoints we mean the Checkpoint service in the AIS-specification [1]. It's a facility for processes to record checkpoint data which are normally shared between different nodes.

1.4.4 TCP

Transmission Control Protocol (TCP) [2] is one of the core protocols of Internet. It provides reliable, in-order delivery of a stream and is used to send both e-mail and web-traffic. Before sending data a connection has to be established and that is done in three steps. For the TCP studies in this thesis I have used the book TCP Illustrated by W. Richard Stevens [3].
1.4.5 SCTP

Stream Control Transmission Protocol (SCTP) [4] is a transport layer protocol for IP-networks. It is reliable and uses congestion control as TCP but instead of transporting one stream as TCP it can transport multiple streams. SCTP also supports multi-homing and has a verification tag to prevent insertion of false messages. SCTP was originally designed for transport of PSTN signalling on IP-networks. If you don’t have any previous knowledge of SCTP, to understand this thesis you should probably read rfc3286 [5] which contains an introduction to SCTP.
Chapter 2

Analysis

2.1 Combining other solutions

I have found three solutions for TCP-state transfer. The first one is Migratory TCP [6] which is a TCP-compatible transport protocol. It provides a mechanism for the client protocol stack to migrate the connection to an alternate server. The migration is transparent to the client and may be triggered according to a migration policy. A kernel based FreeBSD implementation has been made but it doesn't seem to be publicly available.

The second one is tcpip [7] which allows cooperating applications to pass ownership of TCP connection endpoints from one host to another. It's implemented in the Linux kernel on the server and the client is unaware of the movement.

The last one is MIGSOCK (Migratable TCP Socket in Linux) [8] which migrates TCP-sockets. It's also implemented in the Linux kernel on both the server and the client. It has TCP interoperability as long as it doesn't try to migrate a socket where the other endpoint does not support MIGSOCK.

The first assignment was to search for existing solutions and see if they were possible to use entirely, partially or if it was better to create a new one from scratch. Migratory TCP and MIGSOCK were not possible to use at all since they require modifications on the client. Tcpcp looked a lot better but it only supports switch-over and not fail-over. It would be possible to extend tcpcp but since it's kernel based it would be hard to extend it to support Solaris and also a bit trickier to support SCTP. It was also a drawback that it had a GPL-license which prevents Operax to include it in closed-source products. Because of these reasons we decided to do an entirely new implementation.

2.2 Basic idea

A basic design was created which consists of a middleware in userspace which uses OpenAIS [9] to provide heartbeat monitoring and sharing certain state-specific information between the servers. The middleware has its own service-ip and when a packet arrives from the client, the middleware collects specific information and then change the source ip to the service ip and the destination ip to the server's own ip. When we get a reply from the server we collect
some state information and then changes the source ip to the service ip and the
destination ip to the client’s ip.

When the server crashes and a hot-standby node goes active and takes the
service-ip it goes through all open connections and for all connections in estab-
lished state it starts a new connection to the same service on this server. When
a packet from an old connection arrives we then have an open connection to
send it to. This requires that not only the ip but also other protocol-specific
information like sequence-numbers needs to be changed in the packets.

To identify the connection when the packet returns from the server the mid-
dleware can only rely on the destination port number which is the source port
number that was used when the packet was sent to it. This is because all pack-
ets will have the service-ip as destination-ip. Therefore we need the middleware
to use a unique port number for all connections which will be changed in all
packets, not just the moved one.

2.3 Moving TCP

To move a TCP-state and to be able to do it in a fail-over case there is a lot of
TCP state-specific information we need to share between the server nodes. Since
the ip and port number is changed we need to recalculate the TCP-checksum
and the ip-checksum in all packets regardless if the state has been transferred
to another server or not.

2.3.1 Sequence numbers

The sequence number is set to a number when a connection is initiated and
then it’s increased with one for each byte of data that is sent. This is done
separately on both the client and the server. The acknowledgement number in
the TCP-header is the next sequence number that the node expects to receive.

![Figure 2.1](image)

Figure 2.1 shows when a client sends a data-packet which consists of 7 bytes
and the server sends an ACK back.

When a state is being taken over we need to send a sequence-number to start
a new connection to the service on the new node. It would be possible to use
the latest sequence-number plus the number of bytes sent from the client but
instead I chose to use the server’s last ack-number minus one. This is because then we can restore the data that hasn’t been acknowledged since the client has it in its buffer. Then we have the sequence-numbers from the client and the ack-numbers from the server in order. On packets to the client we can’t choose sequence-numbers since they are generated by the new server’s kernel. Therefore when we start up the new connection and get the SYN-ACK packet back we use the client’s latest ack-number and the new server’s sequence-number to calculate a difference between them and then we store it. After that all sequence-numbers from the server and ack-numbers from the client are changed using the earlier calculated difference.

Figure 2.2 shows what happens if server 1 crashed and the TCP-connection in figure 2.1 is moved to server 2. For sequence-numbers all we need to share between the nodes are the latest ack-numbers that are seen in both directions and these need to be stored on all TCP-packets that are passing through.

2.3.2 Flow control

Both nodes advertise a window size which is the number of bytes that the node is willing to accept. We need to share the latest windows size we have received from the client so our middleware can insert it into the SYN-packet when a connection is taken over.

There is also a window scale option which allows the value to be scaled and therefore provides windows larger than the 16-bit field normally would allow. The window scale option can only appear in a SYN-packet and therefore the value is fixed after the connection is established. The client’s window-scale is needed to insert it into the SYN-packet when a connection is taken over. We both need to share the first server’s window-scale (that is sent to the client) and the one from the current server and if they differ we need to recalculate the window size on all packets sent from the server.
2.3.3 Maximum segment size

The Maximum Segment size (MSS) defines the largest segment that the node can receive. The option can only appear in SYN-packets so they are fixed after the connection is established. We only share the received MSS since for the concept to work the servers need to be on the same LAN and should advertise the same MSS, if not it would be really hard to solve.

2.3.4 Timestamps

The timestamp option lets the sender place a timestamp value. These are used both to calculate a round-trip time and also for PAWS (Protection Against Wrapped Sequence numbers). We need to share the latest received timestamp in both connections. In the SYN-packet when a connection is moved we send the client’s latest timestamp value and if we get a timestamp from the server we use it to calculate a timestamp offset which we then apply to all timestamps from the server and timestamp-echoes from client.

2.3.5 TCP Selective Acknowledgement

We look for the SACK-permitted TCP-option which we then share. If SACK is enabled after a move we need to change all ACK-numbers in SACK TCP-options from the client using the same difference that was calculated for the sequence-numbers.

2.4 Moving SCTP

SCTP [4] also has a lot of state-information that needs to be shared to support fail-overs. SCTP supports multi-homing and its use is if the client/server has multiple IP-numbers and then multiple paths can be used. This does not help us since the communication is still between the same nodes.

2.4.1 Multi-homing

If the client sends a list of multiple IP-numbers in its INIT-packet we need to store those so we know that packets from ANY of those IP-numbers using the same source port belong to the same state. If our server sends a list of several IP-numbers we need to remove all of them that are not our service-ip. It would be possible to use multiple service IP-numbers to be able to use multiple-paths between the nodes.

2.4.2 Initiate Tag

Initiate Tag is later used as Verification Tag in all sent packets. The client’s initiate tag is used in the first SCTP INIT-packet after a move and the server’s init-tag is used to change the Verification tag on all packets from the client. For not losing security we also separately save initiate tag from the first server so we can verify that the client is sending the correct verification tag before replacing it.
2.4.3 Transmission Sequence Number
Each user data fragment gets a TSN which is independent of any stream sequence-number. Unlike TCP they are added by one/packet instead of 1/byte but we handle them the same way as TCP sequence numbers. Therefore we need to share the latest TSN-acks seen in both directions.

2.4.4 Number of Streams
There are two values, one for the number of outbound stream and one for the number of inbound streams. They are set during the connection establishment and are then fixed values. We need to share both of them to be able to setup the same amount of streams when a connection is being taken over.

2.4.5 Stream Sequence number
SCTP has a sequence number for each specific stream. We need to share these separately for ALL streams in both directions. When a connection is taken over we calculate a stream sequence number difference for each stream which is also shared if the connection is moved again.

2.4.6 Advertised Receiver Window Credit
The dedicated buffer space in bytes the sender has reserved. The client’s value is shared to be used in the INIT-packet when a connection is being taken over.

2.4.7 Selective Acknowledgement (SACK)
After a move we need to change all the Cumulative TSN Acks from the client using the same difference as for the TSN.
Chapter 3

Design

3.1 High level

The chosen design is to develop a daemon which runs on two or more server nodes. The daemon listens to two separate IP-numbers, one service-ip and one for being able to hand-over connections to another node when we do a switch-over.

When packet that initiates a new connection arrives to the service-ip the daemon initializes a struct with state specific information for that connection. At the beginning it adds the client’s ip, client’s port number and also a unique local port number for the transport-protocol. If it’s available it uses the same port as the client’s source-port. The daemon then changes the source ip to the service-ip, the source-port to the newly allocated local port number and the destination-ip to the ip of the node the daemon is running on. Finally the daemon recalculates both ip and transport-protocol checksums and then sends the packet. If its not a packet that initiates a new connection we instead save the values we describe in the analysis chapter.
When a packet arrives from the destination ip the daemon uses the destination port to check which state it belongs to. Then it saves the values described in the analysis chapter to the state struct. After that it changes the source ip to the service-ip, the destination ip to the originating client’s ip and the destination port to the source port used by the originating client. Here we also recalculate checksums and sends the packet out on the network.

If a state has been moved we do the same thing except that we change sequence-numbers etc in the packets. All state-information is shared with the other nodes by using OpenAIS checkpoints.

If the active server crashes, another server takes the service-ip by using gratuitous ARP. It also reads in all the connections from the checkpoint to local structs, goes through them and for all connections that are in the established state it sends connection initialization packets to the same service on this server. When a connection has been established packets from the client are sent there.

When the active node and a standby node do a switch-over all new connections will directly be transferred to the standby node. Thereafter all service-applications will get a message on the control-socket telling them to prepare for a switch-over. When they are finished they send an ack back on the control-socket. Together with that ack they can send data that will be offered to the service-application on the node that is going to be active. When the ack is received all packets that arrive from the client to that connection will be redirected to the new server-node. After a timeout or when all connections have acked, we will redirect all packets from the client to the new server-node. The new server-node will read in the connections as they are moved from the currently active node, start-up the connection and offer possible data from the service-application on the other node. When the currently active node reports that it has finished moving data the standby node will become active and send gratuitous arp to announce that it now has the service-ip.
3.1.1 Checkpoints

We use three different sections, one for TCP states, one for SCTP and a final one for DATA that is used for switch-overs. TCP and SCTP sections first contain a number, which is the number of the maximum number of states that has been used so far. It’s also the number of states that exist in the checkpoint and the number that the nodes will have to read in at a fail-over or switch-over.

The DATA section is used to be able to move certain states to another node and also if wanted send application-data. The section first contains a number that says how many states it contains. Then there is a small struct for each state that says which state it is, the length of the application-data and an application-data offset. If application-data offset and data length is not zero the offset is a pointer to where in the DATA section the data can be found.

The local TCP and SCTP structs are divided into two parts, one that needs to be changed at nearly every packet and one that is seldom changed. Because of that, normally we just write the part that changes often to the checkpoint.

3.1.2 States

We need to know when the connection is established so we know if we should move it or not. We also need to know when it’s finished to free the local port and the state struct. Therefore we must follow how the connection changes state. To be able to know when a connection is closed without too much guessing in which state both nodes think they are in we set a timer every time we change state and if it has been in any other state then established too long we end the state.

3.1.3 TCP

When we do a fail-over or a connection is switched-over the new node first sets the state into a MOVED-state and then sends a SYN-packet to the nodes real-ip to the same service. The SYN-packet contains the sequence-number and TCP-options as described earlier. We then wait for a SYN-ACK reply and when we get it we calculate a difference for sequence-numbers and timestamps as described in the analysis chapter. Next we change state to established and send an ACK-reply for this SYN-ACK to our real-ip. We change state directly since packets to our real-ip go to localhost so it shouldn’t be able to get lost. Then we just drop the SYN-ACK packet.

There are five TCP state values that we save in the TCP struct part that are changed often, they are client ack-number, server ack-number, client window-size, client timestamp and server timestamp.

3.1.4 SCTP

When we do a fail-over or a connection is switched-over the new node first sets the state into a MOVED-state and then sends an INIT-packet to the nodes real-ip to the same service. The INIT-packet contains our initiate tag, number of streams and initial TSN as stated in the analysis chapter. We then wait for an INIT-ACK packet and first we check so the numbers of outbound and inbound connections are the same. Then we calculate a TSN-diff and we also do the
same for the stream sequence number for all streams. The INIT-ACK packet also contains a parameter with a state cookie, we copy the cookie and send it back in a SCTP COOKIE-ECHO packet. Then we drop the INIT-ACK packet and wait for a COOKIE-ECHO packet and when we get it we change the state to established.

There are only four values that we save in the SCTP struct part that are changed often. They are client TSN-ack, server TSN-ack and client stream sequence-number and server stream sequence-number. But since we must save stream sequence-number for each stream we store them in two arrays with the length of a fixed max-stream value.

3.2 Application API

To be able to get information about the real client, when a connection is being moved etc an API was created which can be used by connecting to a unix-domain socket. Some data for example real client ip, real client port, possible data would be able to send first in a TCP connection but then the server applications would need to be completely rewritten. It would also be really hard to in the middle of a connection insert data that informs the application that the connection is going to be moved.

The protocol is very simple and has six different types.

- Connection question
  Is sent by the application to ask about the newly received connection.

- Connection info
  An answer to the Connection question sent by the daemon.

- Is moving
  Sent by the daemon when a connection is about to be moved.

- Ack moving
  Sent by the application to tell that the connection is ready to be moved.

- Get data
  Sent by the application to get application data that it was told about from the connection info reply.

- Data reply
  An answer from the daemon to a "Get data" request containing data sent from the old application.

Protocol headers, all types first have the Generic header and connection info, ack moving and get data only uses the generic header. Since an application could be using one control-socket for many connections all "Is moving" packets are sent with IP proto and port set to zero.

Generic

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
<table>
<thead>
<tr>
<th>Type</th>
<th>IP proto</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Connection info**

<table>
<thead>
<tr>
<th>Real src ip</th>
<th>Moved</th>
<th>MData exists</th>
<th>src port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Ack moving & Data reply**

<table>
<thead>
<tr>
<th>Application Specific Information</th>
<th>Data length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4

Implementation

There are four different threads, one control socket for controlling the application API, one for packet processing, one for cleaning and one for checkpoint writing.

4.1 Startup

Before starting the control socket thread the daemon first initializes the checkpoints, mutexes and libnet. If we run on Linux we then use libpcap to compile a filter and then open a PF_PACKET socket to read from network which we then apply the filter to. On Solaris we instead use libpcap to read from the network. Then we start the packet processing thread, cleaning thread and checkpoint writing thread.

If the daemon start as active it first reads in possible old connections for example if we restarted the daemon without having any hot-standby node. Then it sends out gratuitous arp and then waits until a switch-over is initialized when it starts moving all connections. If the daemon starts inactive it waits for a fail-over or switch-over.

4.2 Processing packets thread

This is the largest function, it handles all packet processing. First it checks if it’s an arp packet and if it’s to our own daemon ip we always answer. If it’s for the service-ip we only answer if we are active. Otherwise, we check if it is an IPv4 packet and if it’s TCP or SCTP.

4.2.1 TCP

If it is TCP and the packet arrives from the client we check that the TCP-checksum is correct. We do not do this if the packet is from our own node since Linux does not add checksums for packets that are sent to the localhost interface. If it’s a SYN-packet we process it, otherwise we try to find a state for the packet. We then save certain pieces of the header and for all packets we also process TCP options since they for example contain timestamps and SACKS. If it’s a moved state we also change the values we have written about earlier. If a packet is of a type that changes the state we also do that. If it’s
an ordinary packet in the established state we schedule for only the small part of the state struct to be written, otherwise we schedule the entire struct. If the packet arrives from the client we just set the checksums to zero since they won’t be checked, otherwise we calculate them and finally write the packet back onto the network.

4.2.2 SCTP

If a packet arrives from the client we check that the SCTP-checksum is correct. If it’s an INIT-packet we process it, otherwise we try to find a state for the packet. The packet can contain several chunks which we all must process so here we enter a while-loop until we don’t have any more chunks. We then save certain pieces of the header. SCTP has a lot of different header types so there is a lot more work here than it is for TCP. If it’s a moved state we also change the values we have written about earlier. If a packet is of a type that changes the state we also do that. If it’s an ordinary packet in the established state we schedule for only the small part of the state struct to be written, otherwise we schedule the entire struct. For SCTP we need checksums when sending to localhost so we calculate them when sending in both directions.

4.3 Control socket thread

The control socket thread listens to new connections on the unix-domain socket and it uses select for not blocking any connection. It consists of a loop which first checks if we are doing a switch-over and in that case it sends out the is moving packet to all sockets. Otherwise it just waits until it gets a message that it can process.

4.4 Cleaning thread

The cleaning thread goes through all TCP and SCTP ports and if they are not in established state and the timer has expired it cleans the local port number and the state struct.

4.5 Checkpoint writing thread

We are using a FIFO queue for checkpoint writing which contains the port number, if it’s SCTP/TCP and if the entire struct should be saved or not. Functions that the process packet thread calls merge these two values into a 32-bit integer and then add it into the queue if it isn’t already there. This thread then reads from the queue and writes to the checkpoint.

4.6 External libraries

4.6.1 OpenAIS

OpenAIS [9] is a BSD licensed implementation of the SA Forums Application Interface Specification [1]. The Application Interface Specification is a software
API which is used to develop applications that maintain service during faults. It consists of Availability Management Framework (AMF), Cluster Membership (CLM), Checkpointing (CKPT), Event (EVT), Messaging (MSG), and Distributed Locks (DLOCK). At this date only Cluster membership, Checkpointing and eventing is in production form.

4.6.2 Libnet

Libnet [10] is a high-level API that allows you to construct and inject network packets. It can be used to easily create packets both at the IP-layer and link-layer. It is distributed under the BSD-license.

4.6.3 Libpcap

Libpcap [11] is a library which provides a packet filtering mechanism based on the BSD packet filter (BPF). The pcap library provides a high level interface to all packets on the network. It is distributed under the BSD-license.

4.7 Security

The design has taken too little security considerations. For example for TCP we do not check that sequence-numbers are in the right window and therefore someone could fool the daemon that a connection is about to be closed and after a timeout the daemon will clean the state. For SCTP the design is rather secure since we use the Initiate tags to verify the SCTP-packets.

4.8 Limitations

When the implementation was done OpenAIS Application Management Framework was only in experimental stage and was not possible to use and therefore the implementation became more of a proof of concept. SCTP-multihoming support was designed for but not implemented due to time constraints.

On Solaris, we get too high delays when capturing packets with libpcap so we can't get higher speeds then about 300kbit/s.

4.9 Performance

In the beginning of the implementation libnet was used for all packet creation and sending but libnet took too much CPU-capacity so instead we use raw-sockets when sending normal packets and libnet only when creating our own packets for example arp response, SYN and INTT packets. On Linux after discovering that libpcap took a lot of cpu-capacity I changed to use PF_PACKET directly.

The most interesting performance test would be how long time a fail-over would take but since the parts of OpenAIS that handled heartbeat and fail-over was not ready this test was not possible to perform.

In reality this implementation would not be used for high-speed data transfers but I have tested it to verify that it doesn't take too much cpu-capacity and that it can handle several connections in high-speeds. The benchmarks are
done with a Fedora 5 Linux Pentium M 1.4Ghz as a server and a FreeBSD 6.0 Pentium D 3.0Ghz as client. The tests have been done on 10 occasions at different times. The machines are on the same gigabit-LAN, no TCP-parameters have been tuned and a normal ttcp between them gives 350Mbit/s. At all times the performance through the daemon was about 90Mbit/s both before and after the connection has been moved. Since the fail-over test is done manually and takes about a second TCP will drop its rate directly after the move but after a while it will reach 90Mbit/s again. The total throughput is the same for 1 connection as for 10 connections. The limitation on 90Mbit is due to maximum usage of cpu-capacity on the server.

The two things that take a lot of cpu-capacity and therefore slow the speed down are checksum calculations and checkpoint writing.
Chapter 5

Conclusions

5.1 Conclusions

In section 2.3 I have found the TCP header and options that need to be shared and modified to be able to handle fail-over and switch over and in section 2.4 I have done the same for SCTP. For what I know this work hasn't been done on SCTP before.

In chapter 3 I describe a design for both the daemon and application API and in chapter 4 I describe the implementation.

The main source of problems has been with OpenAIS, checkpoints have worked on and offf and sometimes even hanged the Linux kernel but in the end it worked fine. I should have given up earlier using OpenAIS AMF and instead used some existing IP fail-over/switch-over mechanism.

5.2 Future work

The implementation works but before it can be of any use it needs to be modified to use some existing IP-failover/switch-over mechanism or wait for OpenAIS AMF to be in production state. If it is going to be used on Solaris the packet capturing mechanism needs to be changed, possible to use DLPI directly instead of libpcap. It would also be good to remove all libnet usage since it's rather easy to create the packets without it and it would increase the speed of a fail-over.

OpenAIS checkpoints is a big speed limitation, one improvement would be to split the state-structs into three instead of two pieces where one does not get shared at all. After that it would also be good to see if the checkpoint code in OpenAIS could be improved, especially when doing really small writes.

For SCTP there is SCTP-multihoming which should be implemented and possibility to use an unlimited number of streams, today that is a fixed value at compile-time.

In TCP on Linux the SYN-packet that should get back if there is no one listening on a port goes to the node-ip instead of the service-ip and therefore the implementation doesn't get it and can't send it back to the client. Therefore the client gets a timeout instead of a connection refused.
Bibliography


