Developing a Functional Tcp/Ip Stack Oriented Towards Tcp Connection Replication *

Javier Paris
MADS Group - LFCIA.
Faculty of Computer Science.
University of A Coruna.
Campus Elvina s/n. A Coruna,
Spain
Tel.+34-981-167000 ext. 1275
Fax.+34-981-167160
paris@dc. .udc.es

Alberto Valderruten
MADS Group - LFCIA
Faculty of Computer Science.
University of A Coruna.
Campus Elvina s/n. A Coruna,
Spain
Tel.+34-981-167000 ext. 1275
Fax.+34-981-167160
valderruten@dc. .udc.es

Victor M. Gulias
MADS Group - LFCIA
Faculty of Computer Science.
University of A Coruna.
Campus Elvina s/n. A Coruna,
Spain
Tel.+34-981-167000 ext. 1275
Fax.+34-981-167160
gulias@dc. .udc.es

ABSTRACT

Functional languages are not often associated with the development of network stacks, mainly due to the lower performance and lack of support for system programming than more conventional languages such as C. However, there are functional languages that offer features which make it easier to develop network protocols than using a more conventional approach based on an imperative language. Erlang, for instance, offers support for distribution, concurrency and soft real time built-in into the language. All those features are desirable to ease the development of network protocol implementations. It is also possible to implement a reasonably efficient network stack in this language, provided some precautions are taken in the design. By using Erlang distribution it is possible to support fault tolerant distributed TCP connections that take advantage of the distributed nature of the applications implemented in Erlang to provide low cost synchronization, with just some support from the application itself.

1. INTRODUCTION

*Work partially supported by MCyT, Spain, Project TIC 2002-02859, and Xunta de Galicia, Spain, Project PGIDIT03PXIC10502PN

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. LANC'05, October 10-12, 2005, Cali, Colombia (c) 2005 ACM 1-59593-008-6/05/0010..\$5.00

The development of experimental features in TCP/IP is often done either using simulators, or modifying the stack of a particular operating system. Simulators are a very useful tool in testing new features, but are not as useful as being able to observe the behaviour in a real environment. Modifying the stack of the operating system is more difficult, and it ties the development with a concrete version of the kernel.

Erlang [2] is a distributed and concurrent functional programming language developed by Ericsson for telecommunication applications. Using Erlang to implement a TCP/IP stack gives several advantages related to its functional design, concurrent and distributed support, fault tolerance and soft real time behaviour. A functional implementation makes the implementation small and easily understandable and modifiable.

This stack can be easily adapted to solve existing problems in TCP. Consider a distributed video on demand (VoD) server, where there are several machines that can be used as frontends to the system. The distributed nature of this VoD server allows it to support fault tolerance by redirecting requests to the machines that are working properly and avoiding those that are not working.

Recovering the connections that were being served by a crashed machine is trickier. With protocols like RTP (based on UDP) it is not too complex to support connection migration, as there is no state inherent to the interchange of packets. However, there are several streaming protocols that use TCP to control user session or even transport data (e.g. RTSP, HTTP). It is desirable to support TCP connection migration to be able to recover those connections in another server of the distributed system transparently from the clients.

The rest of this paper is structured as follows: Section 2 explains several similar approaches to the development of functional stacks, and to the problem of distributing TCP connections. Section 3 gives an introductory explanation to several features of the functional language Erlang. Section 4 explains the design of the TCP/IP stack. Section 5 presents our experience on the implementation of TCP in a functional language. Section 6 gives some performance analysis of the

stack. Section 7 explains the synchronization techniques to develop fault tolerant TCP connections for distributed applications. Finally, section 8 presents the conclusions of this work.

2. RELATED WORK

There are several previous work related to the development of TCP/IP stacks in functional languages. One of the most known ones is FoxNet[3], which is a Tcp/Ip stack together with a web server both developed in Standard ML. The main goal was to prove that functional languages were suited for system development.

There is a TCP/IP stack developed in Erlang called Erlang TCP[15]. This stack focus mainly on the development of a TCP/IP stack using the protocol framework X-Kernel[8]. It is focused neither towards performance nor towards features, but instead tries to be a very clean implementation of TCP/IP in Erlang.

Prolac is a language designed for network protocol implementation which shares many features with functional languages, such as being declarative. There is a TCP/IP stack developed in Prolac[11] which tries to be easily readable so it can be used to test experimental TCP features.

There are also several previous works on fault tolerant services. The most widely used techniques are the backup based systems, which provide fault tolerance by replication of the connection state in one or more slave servers. Clients communicate only with the primary server, while slave server monitor the connection. If the primary server fails, one of the backup servers overtake the primary one and resume the connection. Using this technique several fault-tolerant systems have been designed[13, 4, 5]

Most of these systems require that the client knows about the existence of backup servers and, in the event of server failure, choose another one and resume the connection by itself. Fault tolerance is implemented in the application layer, which forces the client to resume TCP connections.

However, there are many situations in which it is more desirable to do fault tolerance in the transport layer. With a fault tolerant transport layer it is possible to implement all the necessary behaviour on the server side. As there are many situations in which the client cannot be modified (e.g. with settop boxes used in video streaming), this provides the desired behaviour without any change to the client side.

In TCP/IP networks, the main transport level protocols are TCP and UDP. With UDP it is easy to implement a replication scheme where the operations are not interrupted in case of failure, because UDP is connectionless, and no state is kept in the transport layer.

TCP provides several desirable services that UDP does not, but it must keep metainformation about the connection, and this makes connection recovery in a different node much harder. There are several works that try to provide TCP with fault tolerance, like FT-TCP[16], ST-TCP[9], FT-TCP

provides fault tolerance making modifications to the behaviour of the TCP/IP stack. It uses two wrappers, one between Tcp and the Sockets, and another one between Tcp and Ip. Thus, it knows the state of the Tcp connection regarding the interactions with applications, and regarding the communications with the other endpoint of the connection. These two wrappers send this information to a remote machine that logs it. If the server fails, a backup server may recover the state of the connections by reading this log and replaying the communications with userspace. While the recovery is taking place, acknowledgements with zero window are sent to the client to prevent it from closing the connection due to timeouts. The main drawback of this approach is the high cost of recovering connections due to the time needed by the backup to replay the log.

ST-TCP keeps a backup server that listens to the traffic to keep the connection state replicated. The backup monitors the network to see when the primary server fails, and resumes the connection where it was interrupted. As with FT-TCP, application services are replicated without their knowledge. The main drawback of this is that it doubles the application server load by having to keep two running servers at the same point in the primary and the backup. In distributed applications one can take advantage of the fact that the knowledge about ongoing connections is often distributed around the server.

3. ERLANG

Erlang [2] is a distributed and concurrent functional programming language developed by Ericsson for telecommunication applications. The language has no constructs inducing side effects to an implicit store with the exception of communications among threads (processes, in Erlang terminology). Erlang evaluates expressions eagerly as many other strict functional languages such as ML.

Values in Erlang(i.e., non-reducible expressions) range from numbers and *atoms*(symbolic constants, lower-case in Erlang syntax) to complex data structures (lists, tuples) and functional values, which are treated as first-class citizens. A function is defined by a set of equations, each stating a different set of constraints based primarily on the structure of the arguments(*pattern-matching*). Iterative control flow is carried out by using function recursion. Lists, the most important data structure in functional languages, are written $[a, b, c]$ with $[]$ being the empty list, and a list whose first element is X and whose rest is Xs is denoted $[X|Xs]$. Thus, to define a function that tests for membership in a list, we can write:

```
member(X, [])    -> false;
member(X, [X|Xs]) -> true;
member(X, [Y|Ys]) -> member(X, Ys).
```

Hence, $member(3, [1, 2, 3])$ evaluates to the atom *true*, while $member(5, [1, 2, 3])$ evaluates to *false*. In the absence of a static type system such as many modern functional languages provide, lists can contain heterogeneous values such as $[a, [], 1]$. Besides lists, Erlang programmers can also use *tuples*, similar to C structures, which are constructed in arbitrary but finite length by writing $A, B, \dots, N$. Lists and tuples can hold any valid Erlang value, from numbers and atoms to lists, tuples and even functional values.

What makes Erlang different from other functional languages is its support for concurrency and distribution. With Erlang's primitives for concurrency, it resembles formal calculi such as Milner's CCS[10] or Hoare's CSP[7]. The idea behind the scenes is clear: concurrent programming is far simpler in a functional language[1] than in an imperative one because of the absence of side-effects (only process creation and communications).

New threads can be created with the primitive `spawn`. Given a module M , a function F exported from module M , and a list of arguments $[A_1, A_2, \dots, A_N]$, the expression `spawn(M, F, [A1, A2, ..., AN])` starts a new process to compute $M : F(A_1, A_2, \dots, A_N)$. Once evaluated, `spawn` returns the *process identifier*, `Pid`, of the newly created lightweight process.

In other to allow interaction among processes, a couple of asynchronous message passing primitives are available:

- *Asynchronous send:*

`Pid ! Msg`

`Msg` is sent to process `Pid` without blocking the sending process. If `Pid` exists, the message is stored in `Pid`'s *mailbox*. Any valid Erlang value can be sent to other processes, including complex data structures containing lists, tuples, functions or process identifiers.

- *Mailbox pattern matching:*

```
receive
    Pat1 -> Expr1;
    ...
    PatM -> ExprM
end
```

It searches the process mailbox looking for a message that matches one of the patterns $Pat_1, \dots, Pat_M$ sequentially. If no such message exists, the process blocks until it arrives. The result is the evaluation of $Expr_i$ with the bindings carried out in Pat_i .

The natural extension of Erlang to a distributed framework is to have processes running in more than one Erlang virtual machine (*nodes*, in Erlang terminology) possibly running at different physical nodes. In order to create a process on a remote node, `spawn(Node, M, F, Args)` should be used, adding *Node* as an extra parameter with the location of the newly created process. The nice feature is that all the concurrency constructs are semantically equivalent in this distributed framework (even though communications among remote processes is less efficient, of course).

4. DESIGN

The design of the system is based on using multiple processes that cooperate exchanging asynchronous messages to one another. This is the usual approach taken in the design of systems using Erlang.

The system uses at least two processes for each stateless protocol in the stack (Fig. 1), that is, for every protocol but TCP. One of these processes manages input packets, parsing

them, and forwarding them to the upper level protocols. The other process builds binary packets for a given data payload and destination.

These two processes are registered processes¹ in all the protocols of the stack, with a name like *protocol.reader* for input packet processing, and *protocol.writer* for output packet processing. The protocol communicates sending messages to the reader of the upper level protocol, or to the writer of the lower level one.

Figure 1: Tcp/Ip Stack Process Architecture

There is a common pattern in all the modules for communication with the upper and lower level protocols, defining functions to send packets to upper level protocols and to receive packets from lower levels.

This communication between layers is done using by message passing in an asynchronous way.

%%% Protocol Interface

```
start(Configuration) ->
    init(Configuration).

send(Packet, Dst) ->
    catch protocol_writer ! {send, Packet, Dst}.

recv(Packet) ->
    catch protocol_reader ! {recv, Packet}.

init(Conf) ->
    spawn(protocol, init_reader, [Conf]),
    spawn(protocol, init_writer, [Conf]).
```

¹A registered process in Erlang is a process with an associated name which can be used instead of its *PID*

The writer process reads data from its mailbox, builds a packet with it, and sends it to the suitable lower level protocol. A server process is implemented with a recursive function that calls itself with the new state in each iteration. For protocols that do not need to keep state, it just calls itself with the old state:

```
%% Writer Loop

init_writer(Conf) ->
    register(protocol_writer, self()),
    writer_loop(Conf).

writer_loop(Conf) ->
    receive
        {send, Packet, Dst} ->
            {ok, Bin_Packet}=build_packet(Packet, Dst),
            send_packet(Bin_Packet, Dst, Conf);
    end,
    writer_loop(Conf).
```

The reader process works in a similar way, reading packets from lower level protocols, decoding them, and sending the obtained data to the corresponding upper level protocol.

```
%% Reader Loop

init_reader(Conf) ->
    register(protocol_reader, self()),
    reader_loop(Conf).

reader_loop(Conf) ->
    receive
        {recv, Packet} ->
            case catch decode(Packet, Conf) of
                {ok, Upper_protocol, Data} ->
                    Upper_protocol:recv(Data);
                {error, Error} ->
                    {error, Error};
                _ ->
                    {error, failed}
            end
    end,
    reader_loop(Conf).
```

4.1 Raw Socket

A TCP/IP stack in userspace needs a way to read and write raw packets from the network interfaces of the system. In Linux this is provided by *raw sockets*. A raw socket receives network traffic “as-is”, without stripping the network protocol headers, and anything written to it is also transmitted without changes to the network.

The stack must also use a different MAC address than the interface so that it does not interfere with the operating system stack. If both were to use the same MAC address, each one would reset the TCP connections of the other, among other problems. Using a different MAC address implies that the interface must be in *promiscuous mode* for this to work.

Opening a raw socket, and putting network interfaces in raw mode cannot be done in Erlang, so this part of the system is implemented as an Erlang port driver, which communicates with a driver implemented in C. This port driver communicates with an Erlang module that is the virtual network interface for the rest of the stack.

4.2 Arp Protocol

The Arp module has the two usual processes to manage input and output of Arp packets. There is a third process that provides an Ip packet queue. This queue is used to keep packets that must wait for their destination MAC address to be resolved. Each time an Arp reply arrives, the queue is searched for packets that were waiting for that MAC address, and they are sent to the Ethernet layer.

The Arp modules keeps a hash table that associates Ip addresses with Mac addresses. This table is used by the reader and the ip_queue processes.

4.3 Ip Protocol

The currently implemented IP support is the minimal to have a working end point in the communication. That is, the stack does not support forwarding (although it would not be a difficult feature to be added). Its primary function is to create and decode packets for the upper level protocols.

The Ip module keeps a hash table for Ip fragments that may arrive. Each time a new fragment arrives, the fragment id is checked for completion, in which case the packet is sent to the upper level protocol.

4.4 Tcp Protocol

TCP is the most complex protocol of the stack due to its connection oriented operation. Each TCP connection must have some information about its current state, and the protocol behaviour changes according to the current state.

Incoming TCP packets are handled by a TCP process that keeps a table with all open connections and its associated processes. This process parses incoming packets and redirects them to the corresponding connection based on the source and destination Ip addresses and ports.

Each TCP connection is defined by three processes that perform different tasks:

- Tcp Writer: Packet building and other sending related behaviour.
- Tcp Reader: Receiving related behaviour.
- Tcb (Tcp Control Block): State information and process synchronization.

These processes keep synchronization by sending messages to one another.

4.4.1 State Behaviour Modules

Several TCP connection states share much of their behaviour. The stack takes advantage of it by using small modules for each of the TCP states with callbacks for behaviour that changes (Fig. 2). These callbacks are:

- Send function wrapping. This allows each of the states to take actions based on the type of the packet to be sent. For example, the established state may delay a *fin* packet until it clears all the buffered data to be sent.

Figure 2: State Behaviour Control

- Several callbacks that let the state take actions when processing input packets that meet any of these conditions:
 - An *ack* of not yet sent data.
 - An *ack* which does not acknowledge any new data.
 - An *ack* that acknowledges new data.
 - Packet containing new data.
 - Packet having out of order data.
 - Packet having the *fin* flag set.
- Several callbacks that let the state allow or disallow certain user calls:
 - Data sending.
 - Data reading.
 - Closing the connection.

4.4.2 Packet Receiving

Packet receiving is also managed by an independent process. Input packet processing is very similar in all states but CLOSED, LISTEN and SYN_SENT, so these states implement their own behaviour. All the remaining states use a common infrastructure based on callbacks which are called in the points where each state may take a different action.

4.4.3 Packet Sending

Packet sending is managed by an independent process. Sending behaviour is much less straightforward than receiving, which is essentially triggered by the reception of packets from the network. Sending is triggered by several different events (new data to be sent, window opened by peer, data acknowledged by peer, received data that must be acknowledged, user wants to close connection ...). It must keep sending data for a while and stop when it cannot send any more data.

The sending process listens to messages from other processes. If there is any message that requires that a specific packet type should be sent at once, it will process that request. The packet types are:

- *syn*, which is used for sending a packet to open a new connection.
- *synack*, which sends an answer to a syn packet.
- *data*, which tries to send a packet with data.
- *ack*, which sends a packet immediately. It will also send data if there is data that may be sent. The difference with the data packet type is that an ack will be sent even if there is no data available. It is used to acknowledge data received.
- *fin*, which sends a fin packet to close the connection.
- *rto*, which is used to retransmit data which has already been sent.

If there is no message in queue, the sender process will either send a packet with data if there is data available or sleep waiting for a message if there is no data to be sent. When there is data available, it will be awoken by the Tcb process.

When the process knows the packet type to be sent, it will send the request to the current state module. This module decides whether this request should be served or not (this depends on the current state; e.g. the stack stores data to be sent when the connection is not yet established, but the connection opening states will not allow this data to be sent). Once the packet is accepted, a TCP packet will be built with the information recovered from the Tcb, and sent to the IP module.

4.4.4 Tcp Control Block

The connection state is kept in a structure called TCP control block (Tcb). This structure is managed by a process, which updates its values as the reader and writer processes send messages to it.

However, it does more than a simple get/set on values. There are values that must be recomputed each time another value changes, there are timers to be set periodically, messages to the reader and writer processes when certain events happened... The Tcb process manages all those, as it has knowledge over the entire state of the TCP connection.

The Tcb (Fig.3) is structured around a small module, the *Tcb data updater*, whose only function is to read and write the Tcb data structure. This module is decorated with other modules which add features to it. Using decorators[6] it is easy to add optional functionality to TCP, and configure whether it is going to be used or not when the connection is created by selecting which decorators are chained to the Tcb data storer. The decorator structure is not fixed; it may be changed during the life of the connection to adapt the behaviour of the Tcb to changing network conditions.

As an example, the dependent field update behaviour previously described is implemented through a decorator, which monitors the updates of the different fields and updates dependent fields when needed.

Figure 3: Tcb Decorator Architecture

5. FUNCTIONAL STRUCTURES

Erlang code is compiled to binary code for a virtual machine. This virtual machine interprets the code, allocates and deallocates memory on demand through a garbage collector and schedules the different Erlang processes running in the system. All those Erlang processes are running in a single thread of the operating system, and the virtual machine schedules between them. Note that Erlang processes are very lightweight[1]. They are very cheap to create and destroy, and scheduling is also fast.

The implementation of several parts of the stack had to take into account the way of working of the virtual machine in order to have a reasonably fast stack.

5.1 Packet Building

We push hard to avoid multiple copies of data in memory. The usual way to manipulate data in a functional language is through the use of lists. However, Erlang copies these lists when sent in messages to other processes. On the other hand, there is a binary type which is stored in the heap, and it is not copied between processes.

The problem of this binary type is that creating a new binary from two smaller ones involves copying the data. As the packet creation is done by sticking new data at the beginning of a packet, this means that the packet would have to be copied several times in its way to the network interface. This should clearly be avoided.

The approach taken is a mixed one: binaries are used for the data and each of the headers. All of these binaries are stored in a list. The cost of copying the list is not big if the elements inside do not have to be copied: As the elements are binaries, they are not copied. This list has at most one element per protocol in the stack, that is, a maximum of

Figure 4: Packet Building

four elements (Fig 4).

When the data is sent to the Ethernet port driver. Erlang translates this list of binaries into an array of IOVECS, which is the following C structure:

```
struct ErlIOVec {
    struct iovec *iov;
    int vsize;
};

struct iovec {
    void *iov_base; /* Starting address */
    size_t iov_len; /* Number of bytes */
};
```

No copies are done in this step either. All those operations make something similar to scatter/gather I/O.

5.2 Timers

Process creation and communication are cheap in Erlang. That can be used to implement timers as follows:

```
set_timer(Msg, Time) ->
    spawn(?MODULE, timer, [Msg,Time, self()]).

timer(Msg, Time, From) ->
    receive
    after Time ->
        From ! Msg
    end.
```

A timer process is created. This process uses the message receive function with a timeout, but without actually expecting any message. After the timeout, a message is sent to the process that set the timer. That message is the signal that the timer has expired.

6. PERFORMANCE MEASURES

Performance measures were made in a 1 Gbit Ethernet network. The Erlang TCP/IP stack was installed in a Pentium

4 at 2.8Ghz with 1GB of memory and a Broadcom 5700 Ethernet card running Linux 2.6.6. The other side was a 1.4Ghz Pentium M running Linux 2.6.10, with a Broadcom 5705 Ethernet card. The utility IPERF was used to receive and transfer data to carry out the benchmarks.

To measure throughput a 1GB file was transmitted between the two machines in both directions, and measured ten times (Fig.5), taking the mean of the measures. As expected, the larger the Maximum Transfer Unit (MTU), the higher the throughput. The total throughput in the best case is around 150Mbps. As a comparison, Linux TCP/IP stack in the same situation gave a total throughput of 615 Mbps, which is around 4 times higher. The difference in performance can be explained by the cost of copying from userspace and because of the use of a virtual machine. Linux also takes advantage of advanced features present in the network cards, such as segmentation offload, checksum computing in the card, or scatter-gather copies from memory. A test with Jumbo frames would be desirable, and it should be expectable that both stacks would increase their performance. However, the Broadcom 5705 ethernet card does not support Jumbo frames.

Figure 5: Throughput Transferring a 1GB File

The effect of several concurrent connections can be seen in Fig.6. Up to 3000 concurrent connections were opened with a remote machine, but this reduced the throughput. A similar test was done with Linux, but opening only 500 connections, as the behaviour quickly dropped over that. Linux showed no reduction in throughput with those connections, but a few connections used all the available bandwidth while the others did nothing. The Erlang stack loses throughput but each of the connections gets the same share of the bandwidth. The different schedulers used probably explain this behaviour: with Linux there are several threads which are scheduled by the operating system, while in Erlang all of the threads run in a single process, which does its own scheduling. The Erlang scheduler is able to run a larger number of threads per second than the Linux scheduler due to the lightweight nature of the Erlang processes.

Figure 6: Throughput under Concurrency

7. TCP CONNECTION DISTRIBUTION

Distributed applications spread the load of their tasks over several machines connected by a network. The distribution provides two big advantages: fault tolerance, because the application may distribute work of failed machines to others; and higher throughput, as the load can be shared across the system.

For a distributed server that provides its services through the network, fault tolerance can be useful to improve its availability. However, if the service uses TCP as its transport protocol, failures in single machines affect all the operations that had connections to it. To prevent this from affecting the service as perceived by the user, the client must reconnect to the service in order to mask the failure. This is a good solution for an ad-hoc service in which the server and the client are designed this way, but there are many situations in which that is not directly applicable, e.g. if the service is a standard one and there is no control over the specified behaviour in case of failure, or there are several clients over which no control is possible.

One solution to this problem could be to replicate the state connection in the server. The state connection includes all the state the distributed application has in the machine that is serving the request, and all the state of the TCP connection. If more than one machine of the distributed system has the state of the connection, it may take over failures of other machines without the client ever noticing.

There are several possible approaches to the replication of the state of a TCP connection. As distributed applications have or may have knowledge about current operations in more than one place of the system, it would be desirable to take advantage of this to make the cost of the synchronization lower.

A common approach in making TCP fault tolerant has been to have backup servers, to which the packets directed to the primary server are somehow copied. The output from these servers is discarded, but this way their state is similar. There are problems with this approach depending on the contents of the stream, but there are also techniques to work around them.

One of the problems of this approach is that it duplicates the load of the primary server in the backups because each backup has to repeat the same steps as the primary. The main advantage is that it makes adapting existing servers easier.

However, one of the main reasons for using a distributed server is the performance gain that is obtained by having several machines collaborating. If the load is duplicated at different computers, this is lost. Keeping a backup connection should not imply having the same load for it as the primary server. The way to achieve this is to make it possible for other machines in the distributed system to recover the state of the distributed system in the machine affected without having to follow it step by step. Being able to do that depends, of course, on the distributed application.

7.1 Synchronizing Tcp State

The state of TCP connections is kept in a structure named Tcb (TCP Control Block). The synchronization requirements of the different variables stored in this structure is different:

- *Initial Sync:* Some of the state can be copied when the connection is established, and after that it will not be modified again, e.g. the IPs and ports of the connection.
- *Immediate Sync:* Several variables must be kept synchronized at all times between the primary and backup servers. When one of these variables is modified, the server must notify the backup servers before continuing, e.g. sequence numbers.
- *Deferred Sync:* Some variables can be synchronized periodically. This is because even if their values are not completely updated, they can be recomputed when the connection continues in the backups, e.g. all variables related to the measure of the round trip time.

The goal is to try to reduce the synchronization overhead as much as possible. If primary has to sync with the backups for every packet sent or received, then the load will be somewhat similar to that of the primary, which is exactly what we are trying to avoid.

Let us look at the problem from the perspective of what happens when something is received or transmitted.

7.1.1 Sending Data

Let us start with the case that the server is sending data to the client, and this is just answering with acks. The easiest way to do an active synchronization is to make it for every packet send or received (Fig.7). The primary synchronizes with the backups before every packet is going to be sent, and after every ack is received. If the primary fails at any moment, the worst effect would be that of a lost packet, which is a situation that TCP recovers easily from.

However, synchronizing with the backups for every packet is time-consuming, and it is affected by latency. There are

Figure 7: Sending with a Backup Server

several possible optimizations, that benefit the normal operation putting an extra cost to recovery.

First, it is not really necessary to synchronize after every packet sent. The only critical metadata that must be synchronized at all times is the sequence numbers and the window sizes, that is:

- **snd_una**, the last ack received.
- **snd_nxt**, the sequence number of the next byte to be sent.
- **rcv_nxt**, the sequence number of the last data packet received from the other side.
- **snd_wnd**, the window size advertised by the other side of the connection.

As we are only looking at the sending behaviour, **rcv_nxt** is constant, and does not need to be synchronized. We will look at its synchronization in section 7.1.2.

The way to recover the rest of the data from the backup, even if it is not fully synchronized, is to send a packet that is out of the send window of the other side. As the standard specifies[12], when a packet is received which does not overlap the receive window an ack is generated and sent in response. This ack has the next sequence number expected in the ack field, which will be our **snd_una** and **snd_nxt**; and the new send window in its window field (**snd_wnd**). With this information, the backup server can compute how many bytes were sent since the last synchronization with the primary server, and may purge the already sent bytes from the send buffer. Now, it has all the required information to continue the connection at that point.

Using this optimization synchronization is only needed in interactions with the user, mainly when new data is buffered for sending. When this happens, the primary can update the metadata as well (Fig.8).

The main cost of this technique is that all the sent data must be copied to the backups, which for bulk data transfer may be costly. However, in a distributed application there are many times in which the data can be safely regenerated

Figure 8: Optimized Sending with a Backup Server

in the backup node if it is supplied some information (e.g. a web server which is sending a big file can be resumed at the backup just by knowing the position of the last byte sent). So, by adding a new function to the sockets API, which lets the application send some private metadata to the backup TCP stacks, this can be later retrieved by the application in the backup nodes when the connection is resumed. Using this metadata, and the number of bytes sent since that metadata was stored, the distributed application may resume the connection at the point it was left, with the advantage that the synchronization of all the data has been exchanged for the synchronization of some metadata which is less bulky and probably does not need to be transferred so often.

There are only two required points of synchronization in this case: when the application requests some metadata synchronization, and whenever `snd_nxt` wraps around 4GB, as we need to accurately know the number of bytes transmitted.

In the web server example, it can just send the position requested in the file. With this and the number of bytes sent since the position was sent, the backup server has the information needed to resume the connection with only one synchronization message for all the established state of the connection.

7.1.2 Receiving Data

Now let us look at the opposite case: the reception of data with no data being sent. Of all the data that was being synchronized in the previous case:

- `snd_una` is constant, as no data is sent.
- `snd_nxt` is also constant.
- `rcv_nxt` now changes every time a packet with data is received.
- `snd_wnd` may change depending on the buffer space in the other side of the connection.

As in the previous case, all this data may be recovered by sending a packet out of window (the only field we did not look at in the previous case, `rcv_nxt`, can be recovered from

the sequence number of the ack received). This way all the required metadata may be recovered from the information the other side has.

Figure 9: Receiving with a Backup Server

As in the previous case, the only synchronization needed is when new data is received. So the easiest way to synchronize in this case is doing that for every packet with new data that is received (Fig.9).

Figure 10: Optimized Receiving with a Backup Server

However, the other side will not erase the data from their buffers until we send an ack (Fig.10). So the only synchronization point needed is just before sending one. If the connection fails before the ack, the packets not yet acknowledged can be considered as lost packets and they will be retransmitted by the other side. The backup node may help this when resuming by sending several acks to trigger fast retransmit[14].

As the synchronization must be performed for every ack sent, it may be desirable to delay acks for some time to reduce the number of primary-backup communications.

7.1.3 Sending and Receiving Simultaneously

The behaviour is just the combination of the two previous behaviours, that is, a synchronization is needed when an ack of new data is going to be sent, or when the user queues more data in the send buffer. If the sending is optimized so that no synchronization is needed for sending data, then the only point of synchronization is the sending of an ack.

Figure 11: Send Synchronization

7.2 Synchronizing the Distributed Application

The TCP/IP stack keeps the TCP state between the primary server and the backups synchronized, but the distributed application also has to keep its internal state synchronized between them. In some cases the synchronization may be cheap (as we saw in the distributed web server sending bulk data), but there are cases in which the cost is going to be higher.

Synchronizing the state of the distributed application is a task of the application level, but the stack should try to prevent duplicated traffic between the primary and the backups, due to two reasons:

- Copying the transmitted and received data twice (one for the application and one for the TCP/IP stack) is inefficient.
- It introduces a problem with the synchronization of TCP and the application in the backup nodes, which may have a different view on the state of the connection. If only one of the application or the stack synchronizes and the primary fails the state in the backup node is going to be inconsistent.

The synchronization of the state of the TCP connection and the application state related to the connection must be atomic, so they cannot be sent at different times. The TCP/IP stack must provide a way to synchronize the connection state for the application. With this, the synchronization with the backups may be done in a single message, which guarantees the consistency between the view of the connection of the TCP/IP stack and the distributed application (Fig.11).

The sockets API has to be modified with the addition of new functions that allows the application to pass opaque synchronization data that should be delivered to the backup. If the application does not require these functions, the stack may assume that the application can synchronize its state using the last synchronization made and the number of bytes transmitted since then; hence, it can use the optimization technique described in section 7.1.1 to reduce the traffic between the primary and the backup.

To synchronize on reception, the key aspect is when to send an ack. Acks can be delayed for a time to make use of the buffering of the other side of the connection to recover state. Reception should use a policy that:

- Gives time to the application to read data and produce a synchronization message to the backup node. This avoids having to copy the data to the other node, and the data of several messages may be compiled in one update.
- If the application does not read the data in a given amount of time, the data should be transmitted to the backups. This allows the primary to send an ack to the other side and prevent retransmission from happening. This data can be discarded if a later synchronization advances the reception (`rcv_nxt`) past it.

8. CONCLUSIONS

The development of a TCP/IP stack in a functional language offers several desirable features. First, it gives an easy way to learn from an implementation of the protocol. It is also easier to add features to the protocol than in an imperative implementation. The userspace implementation also eases the ability to add advanced features in the future, such as TCP connection replication.

Erlang may not seem to be well suited for an application like a network stack, which performs mainly repetitive tasks over data, and that involves copying large amounts of data and dividing it into small chunks, and performing operations over them. However, it provided several unexpected benefits, like the fairness in the scheduling in presence of thousands of threads.

Several techniques for distributing the state of Tcp connections were presented. As Erlang applications are usually distributed, it was desirable to take advantage of the distribution of state the application already does. To this end, several optimizations to reduce the synchronization costs were presented. These optimizations are mainly based on cooperation between the Tcp/Ip stack and the application. Obviously, this implies changes to the Sockets API, but we believe that the benefits offered to distributed applications offset the problems.

Distributed applications, such as distributed VoD servers, may take advantage of the fault tolerance offered by these connections to provide uninterrupted service even in presence of total failure in some machines of the server. This connection replication can also be used by the VoD server to migrate some connections when there are overloaded servers in the system, thus improving the quality of service to its clients.

9. REFERENCES

- [1] J. Armstrong. Concurrency oriented programming in erlang. In *Proceedings of the Lightweight Languages Workshop 2002*.
- [2] J. Armstrong, R. Virding, C. Wikström, and J. Vliissides. Concurrent programming in erlang, 2nd edition, 1996.
- [3] E. Biagioni. A structured TCP in standard ML. In *Proceedings, 1994 SIGCOMM Conference*, pages 36–45, London, UK, Aug. 31st - Sept. 2nd 1994.

- [4] N. Budhiraja and K. Marzullo. Highly-available services using the primary-backup approach. *Proceedings of the 2nd Workshop on Management of Replicated Data*, Monterey, CA, 1992.
- [5] F. Cristian. Understanding fault-tolerant distributed systems, Oct. 16 1991.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Element of reusable object oriented design, 1995.
- [7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1984.
- [8] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.
- [9] M. Marwah, S. Mishra, and C. Fetzer. Tcp server fault tolerance using connection migration to a backup server. *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN 2003)*, San Francisco, CA, june 1993.
- [10] R. Milner. *A Calculus for Communicating Processes*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.
- [11] D. R. Montgomery, E. Kohler, and M. F. Kaashoek. A readable TCP in the prolac protocol language, Feb. 20 1999.
- [12] U. of Southern California. Rfc 793: Transmission control protocol., Sept. 1981.
- [13] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability issues in computer system design. *ACM Computing Surveys*, 10(2):123, June 1978.
- [14] W. Stevens. RFC 2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, Jan. 1997.
- [15] T. Törnkvist. Etcp: Erlang tcp/ip, 1996.
- [16] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud. Engineering fault-tolerant TCP/IP servers using FT-TCP, Apr. 22 2003.