An Erlang High Performance TCP/IP Stack

Javier París  Víctor M. Gulías  Alberto Valderruten

<{paris,gulias,valderruten}@dc.fi.udc.es>
Índice

1 Introduction
2 Design
3 Optimization Issues
4 Tests
5 Distribution
Motivation

Problem:
- Streaming video server in Erlang.
- Uses protocols like HTTP and RTSP => TCP.
- Connections usually last for hours.
- Cut in the connection => playing stops.
- Clients are usually closed settop boxes.

Tcp must be fault tolerant in the server side.
Índice

1. Introduction
2. Design
3. Optimization Issues
4. Tests
5. Distribution
Overall Design

Process Structure:

- Ethernet Input
- Ip Input
- Arp Input
- Raw Socket Input
- C Raw Socket Input

- Ip Output
- Arp Queue
- Ethernet Output
- Raw Socket Output
- C Raw Socket Output

- Tcp Connection
- Tcp Control Block
- Tcp Reader
- Tcp Writer

Introduction
Design
Optimization Issues
Tests
Distribution
Network Interface

- Erlang does not support file operation on devices => Linked-in driver.
- Linked-in to avoid unnecessary data copies.
- Interface with the card is a Linux Raw Socket.
- Network Card must be in promiscous mode.
- Data is read and written directly to the interface.
Server Model

- Two processes for each stateless protocol:

```erlang
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).
```
Server Model

The reader is similar:

```erlang
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};
            end
        end,
    reader_loop(Conf).
```
Tcp Design

Tcp is stateful. Must keep state for each connection:
- Reader process.
- Writer process.
- TCB (Tcp Control Block): state and synchronization.
State behaviour is implemented using callbacks and wrappers:
Índice

1. Introduction
2. Design
3. Optimization Issues
4. Tests
5. Distribution
Functional Scatter Gather:

- **Data**
- **Tcp Header**
- **Ip Header**
- **Eth Header**

- **Headers**
  - **Tcp Header**
  - **Ip Header**
  - **Eth Header**

- **Data**
  - **<<Data>>**
  - **<<Tcp Header>>, <<Data>>**
  - **<<Ip Header>>, <<Tcp Header>>, <<Data>>**
  - **<<Eth Header>>, <<Ip Header>>, <<Tcp Header>>, <<Data>>**
Checksum

Checksum was done in several ways:

- Deconstructing a binary one byte at a time: 13MB/s

  ```erlang
checksum(<<Num:16/integer, Remainder/binary>>, Csum) ->
  checksum(Remainder, Csum + Num);
```

- Deconstructing a binary eight bytes at a time: 28MB/s

  ```erlang
checksum(Bin = <<N1:16/integer,N2:16/integer,
           N3:16/integer,N4:16/integer,
           N5:16/integer,N6:16/integer,
           N7:16/integer,N8:16/integer,
           Rem/binary>>, Csum)
  when size(Bin) >= 16 ->
  checksum(Rem, Csum+N1+N2+N3+N4+N5+N6+N7+N8);
```
Checksum

- Using the same binary eight bytes at a time: 32MB/s
  ```erlang
checksum(Bin, Position) when size(Bin) >= 8 ->
  <<_:Position/integer, N1:16/integer,
  N2:16/integer, ..., _/binary>> = Bin,
  checksum(Rem, Csum+N1+N2..+N8);
```

- Using a C linked-in driver: 136MB/s
Índice

1. Introduction
2. Design
3. Optimization Issues
4. Tests
5. Distribution
Throughput

Throughput Transfering a 1GB File

- Sending
- Receiving

MTU vs. Throughput (Mbps)
Concurrency

Throughput under Concurrent Connections

Total Throughput

Connections

Mbps
Distribution

- Aim: Make it possible to recover ongoing TCP connections of failed machines in backup nodes.
- Erlang applications are often distributed => Try to take advantage of distributed information about the state to ease the cost.
First Approach: Make a synchronization before every packet sent and after every packet received.
Distribution: Optimizing Send

Optimization:

- Data that must be synchronized at all times: Sequence Numbers and Window Information.
- Client also has this information at all times.
- An out of window packet must be answered with an ack (RFC 793) => The backup can recover this information by sending an out of window packet to the client.
- The client can use initial sequence number to generate an invalid sequence number. When the initial sequence number is about to enter window again the primary node must warn the backups => Forced Synchronization only every 4GB.
The application must continue where it left in the failed node. The stack may help by saying how many bytes have been transmitted.
Distribution Reception

Harder to Optimize:
- Buffered received data cannot be recovered.
- Synchronization is needed every time an ack is received.
Future Challenges:

- Failure detection and collision avoidance between master and slaves.
- How to physically distribute the servers.
- Support for load balancing in distributed by using connection migration.
An Erlang High Performance TCP/IP Stack

Javier París, Víctor M. Gulías, Alberto Valderruten

<{paris,gulias,validerruten}@dc.fi.udc.es>