

# Erlang in the Corelatus MTP2 Signalling Gateway

Matthias Läng: Corelatus AB  
matthias@corelatus.se

September 2001

## 1 Introduction

Corelatus is a startup founded by Thomas Lange, Matthias Läng and Ulf Svarte Bagge approximately one year ago. We build hardware for use in telecommunications networks.

This paper looks at our experiences using Erlang in a stand-alone system which terminates SS7 MTP2 and passes packets to a general-purpose server for further processing. Erlang was used throughout the project: from when it was just an idea to when it became a certified, tested, approved and mass-produced product suitable for placement in the core of an operator's SS7 network.

## 2 The Concept

Our concept was to build a stand-alone box to handle the parts of a telecommunications system which are difficult or expensive to implement using general-purpose computers. This would enable software developers to build complete systems by combining our hardware with ordinary server hardware. The most important features are:

- Carrier-grade hardware: a rack mountable chassis, dual 48V DC power inputs and no moving parts.
- Approved for use in telecommunication centers. This means passing lightning tests, surge tests, electrical and radio interference tests and so on.
- Support for aspects of media and signalling which are too timing-sensitive to be easily handled by a general-purpose server.
- Narrow interfaces. Electrically, this means *not* having a shared bus with other hardware. Physically, it means being in a separate box. Logically, it means the programming interface runs over TCP (on Ethernet). The narrow interfaces give us complete control of, and responsibility for, everything inside the box. If the box crashes, there is no doubt where the problem lies.

Our customers use our hardware, called the GTH, inside operators' core SS7 networks. Today's SS7 networks are mostly tied together with 2Mbit/s E1 or T1 links carrying SS7 signalling.

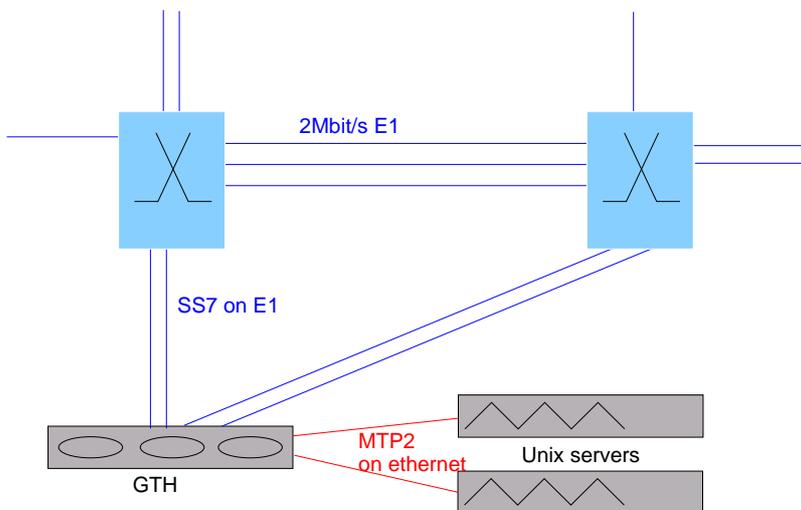


Figure 1: Unix servers connected to the telephony network via GTH

The parts of SS7 which have stringent timing constraints are MTP1 and MTP2. For this application, our hardware provides MTP1 and MTP2 on the E1 links and transfers packets over IP to Unix (or NT) servers for higher-layer SS7 processing. In IETF nomenclature, the Corelatius GTH acts as a *signalling gateway*.

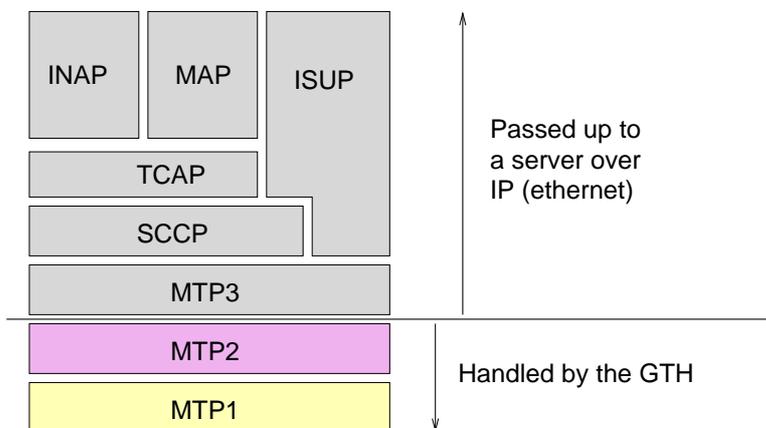


Figure 2: SS7 stack

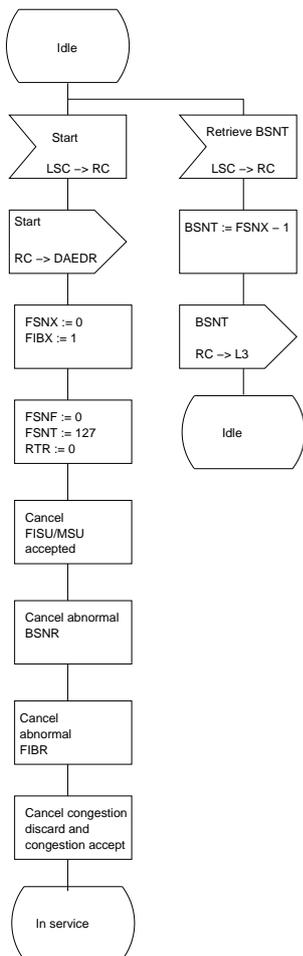
### 3 MTP2

MTP2 provides three services: it moves packets from point-to point, it removes incorrect packets and it handles retransmission. The official definition of MTP2 is the ITU standard Q.703. When we started, I had never looked at MTP2 before. I decided to implement MTP2 in Erlang as a pleasant way of gaining experience.

### 3.1 MTP2 in Erlang

The ITU specification is 89 pages long, of which 58 are SDL[4] diagrams. Our Erlang implementation is 2004 lines long, or about 33 pages.

We wrote the Erlang code by directly translating the state machines in the SDL part of the standard to Erlang `gen_fsm` state machines. SDL and Erlang are a good fit for each other: most of the SDL in MTP2 consists of ten concurrently executing state machines sending each other messages (*signals* in SDL-parlance). Using an automatic `gen_fsm` graphing tool [5] produces diagrams which are strikingly similar to the standard. The resulting code is also easy to compare to the standard:



Q.703, p.59

```
idle(start, StateData) ->
  ok = gen_fsm:send_event(
    StateData#state.daedr, start),
  {next_state, in_service, StateData#state{
    fsnx = 0,
    fibx = 1,
    fsnf = 0,
    fsnt = 127,
    rtr = false,
    fisu_msu_accepted = false,
    abnormal_bsnr = false,
    abnormal_fibr = false,
    congestion_discard = false,
    congestion_accept = false
  }};

idle(retrieve_bsnt, StateData) ->
  BSNT = sevenbit_dec(StateData#state.fsnx),
  ok = gen_fsm:send_event(StateData#state.l3,
    {bsnt, BSNT}),
  {next_state, idle, StateData};
```

While testing the Erlang implementation against itself, we found two errors in the ITU specification. <sup>1</sup>

<sup>1</sup>The errors are corrected in an ITU errata.

## 3.2 Bit-stuffing and Performance

Bit stuffing is central to MTP2 (and related protocols like HDLC). It solves the problem of how to delimit messages in a communications channel which is nothing more than a stream of bits. A special sequence of bits delimits messages, and messages are modified before transmission to ensure that they do not contain the delimiter. Sending the message 0000 1011 1111 1000 looks like this at the transmitter:

1. Send the message delimiter: 0111 1110
2. Modify the message in a way which guarantees that the delimiter never appears in the message. We do this by inserting an extra zero-bit whenever we just sent five one-bits in a row. The message is now 0000 1011 1101 1100 0
3. Send the message delimiter.

The receiver does the reverse, including removing the extra zeros. If we could write the transmitter using the Erlang binary syntax, it would look something like:

```
encode_message(Bin) ->
    Flag = 2#01111110,
    <<Flag:8, stuff_message(Bin), Flag:8>>.

stuff_message(<<2#11111:5, Rest/binary>>) ->
    <<2#111110:6, stuff_message(Rest)>>;
stuff_message(<<A:1, Rest/binary>>) ->
    <<A:1, stuff_message(Rest)/binary>>.
```

The above is not legal Erlang; Erlang does not allow binaries sized anything other than a multiple of 8 bits. We solved this by exploding all binaries into lists of bits. Running on real SS7 signalling data, our reference implementation can handle one or two timeslots of MTP2 (one timeslot is 64kbit/s). The performance is not important, the objectives were to learn and to generate test bitstreams to verify our high-performance MTP2 implementation.

## 4 Disguising Distributed Erlang

Most of the current and future applications for our hardware use the GTH as part of a larger system, usually with one or more Unix servers controlling the GTH. Since our control system is written in Erlang, this would be a perfect application for distributed Erlang, apart from one catch: Erlang is not popular with non-Erlang users. We examined alternatives, including CORBA, ASN.1 and Megaco and also rejected these for one reason or another. Eventually we were inspired by a pair of comments we stumbled across:

*XML is little more than a notation for trees and for tree grammars, a verbose variant of Lisp S-expressions coupled with a poor man's BNF*  
(Phil Wadler) [7]

Any Erlang message can be encoded in XML[3]. Better still, the message stays human-readable while simultaneously looking a lot less obviously like Erlang.

*if you think TCP guarantees delivery, which most people probably do, then so does Erlang [when passing messages]* (Per Hedeland)[6]

If we assume the converse is also true, then XML and TCP can be combined to produce a protocol with the same functionality as message passing in Erlang. The protocol can be described by a machine-readable BNF-like language (an XML DTD). The user will be comfortably oblivious to the presence of Erlang in their system.

#### 4.1 A General XML Encoding

Using `jinterface` as a guide, we can produce a list of what needs to be encoded

Data type	Representation (example)
Atom	<code>&lt;atom name="abcd"/&gt;</code>
Binary	<code>&lt;binary&gt;base64-encoded data&lt;/binary&gt;</code>
Numbers	<code>&lt;number value="12"/&gt;</code>
Exit signal	see section 4.3
List	<code>&lt;list&gt;&lt;atom name="abcd"/&gt;&lt;binary&gt;B63MGUP&lt;/binary&gt;&lt;/list&gt;</code>
Pid	<code>&lt;pid a="0" b="12" c="19"/&gt;</code>
Port	<code>&lt;port id="1234"/&gt;</code>
Ref	<code>&lt;ref id="12ABC67"/&gt;</code>
Tuple	<code>&lt;tuple&gt;&lt;atom name="abcd"/&gt;&lt;binary&gt;128YTE63MG8&lt;/binary&gt;&lt;/tuple&gt;</code>

Binaries are encoded as base64, references are converted to binaries before being encoded. An example: our system can play lists of audio messages to a subscriber, perhaps "you have 56 dollars and 22 cents left in your account". In Erlang we represent this as

```
Pid = {start, {player, {pcm, 1, 3},
               [youhave, fiftysix, dollars, and,
                twentytwo, cents, leftinyouraccount]}}
```

Using the above encoding to carry the same information:

```
<tuple>
  <atom name="start"/>
```

```

<tuple>
  <atom name="player"/>
  <tuple><atom name="pcm"/><number value="1"/>
    <number value="3"/>
  </tuple>
  <list><atom name="youhave"/>
    <atom name="fiftysix"/>...</list>
</tuple>
</tuple>

```

This works, but it is only slightly prettier than an obfuscated PERL contest. It also fails to meet our goal of not explicitly exposing Erlang concepts in the API.

## 4.2 A Problem-specific Encoding

Our API doesn't have that many messages, and the messages are not arbitrary terms. They are all tuples which start with an atom. There are only eleven top-level message types. If we defined a separate XML encoding for each message we could cut the size of the messages, remove the references to tuples and lists and write a more precise DTD to specify the interface.

Example: to start a DTMF detector on the GTH, we use the internal message

```
{start, {dtmf_detector, {pcm, 3, 9}, Message_dest_pid}}
```

A single-purpose encoding for this message is

```

<start>
  <dtmf_detector dest="apic13">
    <pcm_source span="3" timeslot="9">
  </dtmf_detector>
</start>

```

Similarly, the example for message playback becomes

```

<start><player>
  <clip name="youhave"/><clip name="fiftysix"/>...
  <pcm_sink span="3" timeslot="9"/>
</player></start>

```

As a final touch, we replaced the words *start* and *stop* with *new* and *delete* to add to the illusion of object orientation. The DTDs[2] on our website define our API's grammar.

### 4.3 Processes and Linking

The XML encoding takes care of formatting the data we want to send back and forth. But there is more to distributed Erlang than just sending messages. There is also RPC, a global namespace, node monitoring and process linking. Supporting all of these seemed unnecessary, so instead of modeling the external control system as an Erlang node, we modeled it like an Erlang port.

An Erlang port is linked to the process which created it. An Erlang port can send and receive messages. If the port dies, the controlling process is notified.

In our API, the external system is connected via a TCP socket. If the controlling process dies, we close the TCP socket. If the socket closes, we kill the controlling process. If any of the Erlang API processes die, all remote processes are notified. This poor man's emulation of Erlang's linking and message passing allows the system to recover from a client process death:

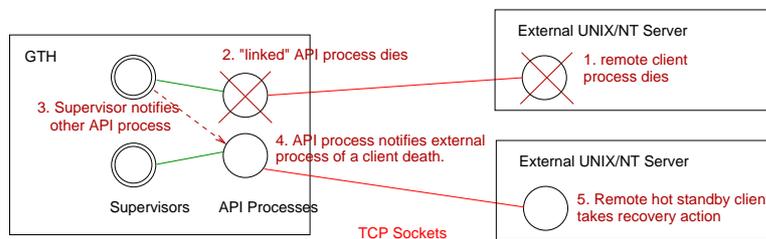


Figure 3: Recovery on a second server due to a first server's death

## 5 Conclusions

Erlang is a nice protocol specification language. Our reference implementation of MTP2 in Erlang is more compact than the SDL + plain English MTP2 specification. By being able to execute our "specification" we had a ready-made test system for our high-performance implementation and we found two errors in the original ITU specification.

Distributed Erlang is a proven way of building robust distributed applications. By encoding Erlang terms in XML and connecting to the non-Erlang node using TCP, it is possible to create a human-readable, language-neutral protocol. By duplicating a proven approach we know it is possible to build robust systems using this approach.

## References

- [1] *The GTH API Documentation*  
<http://www.corelatus.com/gth/api>
- [2] *The API DTDs*  
[http://www.corelatus.com/gth/api/gth\\_in.dtd](http://www.corelatus.com/gth/api/gth_in.dtd)  
[http://www.corelatus.com/gth/api/gth\\_out.dtd](http://www.corelatus.com/gth/api/gth_out.dtd)
- [3] *Extensible Markup Language (XML) 1.0*  
W3C Recommendation  
<http://www.w3.org/TR/2000/REC-xml-20001006>
- [4] *Specification and Description Language (SDL) Forum*  
<http://www.sdl-forum.org>
- [5] *gen\_fsm Graphing Tool*  
Vance Shipley from Motivity Telecom sent me this tool in private correspondence.
- [6] *Post to Erlang-Questions Mailing List* 9. October 1999, archived at [www.erlang.org](http://www.erlang.org)
- [7] *The Next 700 Markup Languages*  
Philip Wadler. Invited Talk, Second Conference on Domain Specific Languages (DSL'99), Austin, Texas, October 1999.