Kamailio World Conference, 2015 Berlin

Leveraging Erlang Node for Scalability

Seudin Kasumovic
Bicom Systems

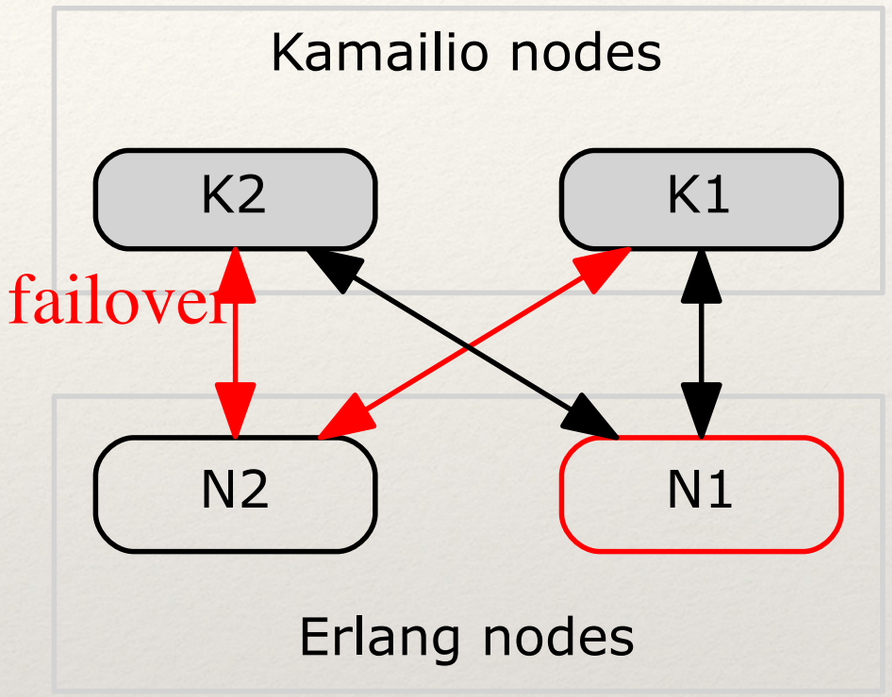
www.bicomsystems.com

Erlang/OTP

- Erlang is functional programming language
- Separate process in the virtual machine
- Asynchronous message passing
- Concurrency and high reliability
- Scale with distribution
- Run-time update/upgrade
- Open Telecom Platform (OTP) Framework

Erlang benefits

- Concurrency: can scales to million of processes per VM
- Let it crash: handle unforeseen errors gracefully
- Run forever: run-time upgrade
- Scale with distribution: run same application on multiple nodes
- Failover and takeover: start application on available node, and back when up



Erlang module

- The goals:
  - Provides as much as possible Erlang data types
  - Asynchronous communication
  - Bidirectional RPC
  - No wrapper in Erlang
  - Failover and takeover connection
  - Allow the creation of custom modules for specific applications

Erlang data types

- Numbers: integer and float
- Atom
- list and tuple
- String
- Process id and reference
- Record
- Bit strings and binaries
- Functional object, map, etc…

```
{‘$gen_cast’,
  {create,
    {cdr,
      {call_desc,
        ”b04f6dca-1233-4a8d”,
        0, outbound
      },
      {service_desc,
        subscriber,1,2,buy},
        ”4420…806”,”4420…807”,
        "UK National”,{{2015,5,12},
        {14,19,6}},
      answered,
      <0.13393.0>,
      0.013,69}
}}
```

Kamailio config variables

- $avp(id), $var(id), $shm(id), $xavp(id), …
- Data value can be string or integer
- Different context and life times: transaction, dialog, process
- Prepends value on the list (e.g. AVP)
- XAVP: extended AVP
  - Can contain multiple named values
  - Internally implemented as list of lists

XAVP in background

- Problem: nested XAVP in script
- Advantages:
  - Allows nested lists
  - Bind to transaction
- API

```
$xavp(a=>foo) = "foo";
$xavp(b=>a) = $xavp(a);

+++++ start XAVP list: 0xb1eaa054 (level=0)
     *** XAVP name: b
     XAVP id: 110
     XAVP value type: 6
     XAVP value: <xavp:0xb1eaa004>
+++++ start XAVP list: 0xb1eaa004 (level=1)
     *** XAVP name: a
     XAVP id: 109
     XAVP value type: 2 (SR_XTYPE_STR)
     XAVP value: <<xavp:0xb1eafca0>>
----- end XAVP list: 0xb1eaa004 (level=1)
     *** XAVP name: a
     XAVP id: 109
     XAVP value type: 6
     XAVP value: <xavp:0xb1eafca0>
+++++ start XAVP list: 0xb1eafca0 (level=1)
     *** XAVP name: foo
     XAVP id: 6992683
     XAVP value type: 2
     XAVP value: foo
----- end XAVP list: 0xb1eafca0 (level=1)
----- end XAVP list: 0xb1eaa054 (level=0)
```

Exported config variables

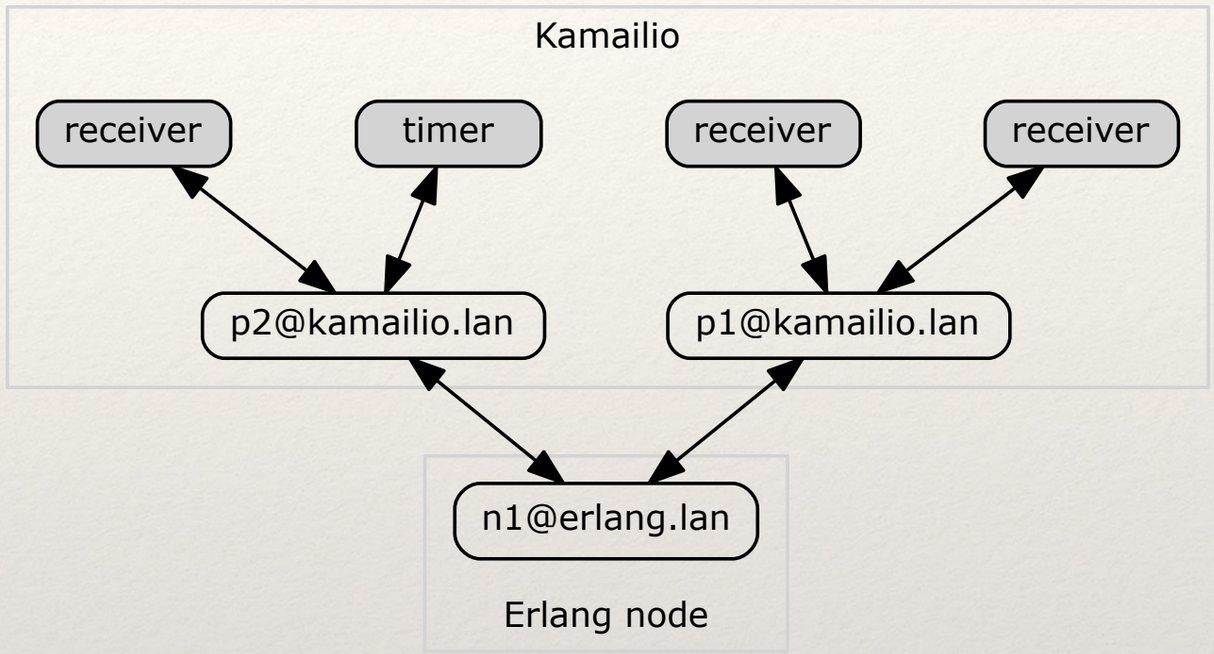
- Based on XAVP structures
- Container PVs:
  - $erl_list(name) - Erlang list
  - $erl_tuple(name) - Erlang tuple
  - $erl_xbuff(name) - generic
- $erl_atom(name) - Erlang atom
- $erl_pid(name) - Erlang PID
- $erl_ref(name) - Erlang reference

```
# {example,message}
$erl_atom(example) = "example";
$erl_atom(message) = "message";
$erl_tuple(M) = $erl_atom(message);
$erl_tuple(M) = $erl_atom(example);

+++++ start XAVP list: 0xb1f27b7c (level=0)
     *** XAVP name: [tuples]
     XAVP id: 2988485433
     XAVP value type: 6
     XAVP value: <xavp:0xb1f0c234>
+++++ start XAVP list: 0xb1f0c234 (level=1)
     *** XAVP name: M
     XAVP id: 68
     XAVP value type: 6
     XAVP value: <xavp:0xb1f0c270>
+++++ start XAVP list: 0xb1f0c270 (level=2)
     *** XAVP name: t0
     XAVP id: 31432
     XAVP value type: 6
     XAVP value: <xavp:0xb1f2bddc>
+++++ start XAVP list: 0xb1f2bddc (level=3)
     *** XAVP name: a2
     XAVP id: 27940
     XAVP value type: 2
     XAVP value: example
     *** XAVP name: a1
     XAVP id: 27943
     XAVP value type: 2
     XAVP value: message
----- end XAVP list: 0xb1f2bddc (level=3)
----- end XAVP list: 0xb1f0c270 (level=2)
----- end XAVP list: 0xb1f0c234 (level=1)
     *** XAVP name: [atoms]
     XAVP id: 1350224368
     XAVP value type: 6
     XAVP value: <xavp:0xb1f2c2b8>
+++++ start XAVP list: 0xb1f2c2b8 (level=1)
     *** XAVP name: message
     XAVP id: 1632000518
     XAVP value type: 6
     XAVP value: <xavp:0xb1f0c434>
```

Connecting to Erlang node

- Kamailio C-node helper processes
- epmd
- Module params:
  - no_cnodes
  - cnode_alivename
  - cnode_host
  - erlang_nodename
  - cookie
- Scale: add more C-nodes




```
(n1@erlang.lan)14> net_kernel:i('p1@kamailio.lan').
Node = 'p1@kamailio.lan'
State = up
Type  = hidden
In  = 1098
Out  = 1099
Address  = 10.1.40.111:46609
```

Failover & takeover connection

- Pretty simple approach
- Not follows Erlang application failover/takeover
- Manage by Erlang have more flexibility
  - erlang:disconnect_node/1, net_adm:ping/1
- After lost connection Kamailio C-node:
  - Periodically trying to connect
  - Accept connection from other node
  - Other node must use valid cookie

Message passing

- Message passing is asynchronous
- Erlang process ID may equally refer to local process or process on remote node
- Process can be identified by name (registered process)
- Exported functions:
  - erl_send, erl_reg_send, erl_reply
- Event route “erlang:<reg_process_name>”
  - “Registered pseudo process”

MP example/1

In Erlang shell:

```
(node1@erlang.lan)38>R=#call_desc{callid=“123123123",branch=1,direction=inbound}.
#call_desc{callid = "123123123",branch = 1,
           direction = inbound}
(node@erlang.lan)39> P=rpc:call('kamailio@pbx.lan',erlang,whereis,[self]).
<13499.9.0>
(node1@erlang.lan)40> node(P).
'proxy@kamailio.lan'
(tbe1@tbe.lan)41> P ! R.
#call_desc{callid = "123123123",branch = 1,
           direction = inbound}
```

Kamailio:

```
event_route[erlang:self]
{
  xlog(“L_DEBUG”,”$$erl_xbuff(msg)=$erl_xbuff(msg=>format)\n);
}

DEBUG: <script>: $erl_xbuff(msg)={call_desc, "123123123", 1, inbound}
```

MP example/2

```
# event route acts as registered process
event_route[erlang:greetings] {

        xlogl("L_INFO","Received message: $erl_xbuff(msg=>format)\n");

        $erl_atom(hello) = "hello";
        $erl_tuple(reply) = "Erlang";
        $erl_tuple(reply) = $erl_atom(hello);

        # reply greeting
        erl_reply("$erl_tuple(reply)");
}

%% in erlang shell

(node1@erlang.lan)24> {greetings,'proxy@kamailio.lan'} ! {hello,"Kamailio"}.
{hello,"Kamailio"}
(node1@erlang.lan)25> flush().
Shell got {hello,"Erlang"}
ok

> logged info message:
INFO: <script>: 951:Received message: {"hello", "Kamailio"}
>
```

Bidirectional RPC

- Classic construct in distributed computing
- Erlang RPC is replaced by a message to send and receive
- Erlang module provides:
  - RPC calls to Erlang node from script
  - Implements the Erlang transport and encoding interface for Kamailio RPCs
- Exported function: erl_rpc

RPC examples/1

- From Erlang node:

```
(node1@erlang.lan)28> rpc:call('proxy@kamailio.lan',dispatcher,list,[]).
[{[{<<"NRSETS">>,10},
   {<<"RECORDS">>,
    [{<<"SET">>,
      {[{<<"ID">>,339},
        {<<"TARGETS">>,
         [{<<"DEST">>,
           {[{<<"URI">>,<<"sip:172.16.24.2:5060">>},
             {<<"FLAGS">>,<<"DX">>},
             {<<"PRIORITY">>,2},
             {<<"ATTRS">>,
              {[{<<"BODY">>,<<"s=39"...>>},
                {<<"DUID">>,<<>>},
                {<<...>>,...},
                {...}|...]}}]}},
          {<<"DEST">>,
           {[{<<"URI">>,<<"sip:203.0.113.2:5060">>},
             {<<"FLAGS">>,<<"IP">>},
             {<<"PRIORITY">>,1},
             {<<"ATTRS">>,
              {[{<<“BODY”>>,<<...>>},{<<...>>,...},{...}|...]}}]}}]}]}},
– more –
```

RPC example/2

- From Kamailio:

```
# example of call erlang:list_to_tuple(["one","two"])
# on remote node

$erl_list(L) = "two";
$erl_list(L) = "one";

# put list into list
$erl_list(args) = $erl_list(L);

erl_rpc("erlang","list_to_tuple","$erl_list(args)","$erl_xbuff(repl)");

xlogl("L_DEBUG","type(repl): $erl_xbuff(repl=>type),
                 format(repl): $erl_xbuff(repl=>format)\n");

> log output:
...
DEBUG: <script>: 386:type(repl): tuple, format(repl): {"one", "two"}
...
```

Erlang module API

- Exported functions are cerated on this API
- Create custom module (application specific)
- Uses dynamic **ei** buffer with encoded Erlang term
- Available functions:
  - rpc, send, reg_send, reply
  - XAVP serialization APIs: xavp2xbuff, xbuff2xavp

Future works

- Add more exported functions:
  - Get node name (from PID, or self)
  - Get self process id
  - Generate Erlang reference
  - Serialize and unserialize exported config variables
  - Extend exported config variable contexts:
    - dialog, hash tables
- Add event routes/callbacks for connected and lost node events
- Add transformations, e.g. easy create atom from string
- Add more Erlang control message handlers: link, exit, trace…
  - Emitting and receiving EXIT signals

Questions?