What NOT to do with Erlang

Chandru Mullaparthi - Principal Software Architect - bet365
(Presented at CodeMesh 2015)

About me

- Worked in the telecoms domain from 1995-2014
- Worked with Erlang since 1999
- Currently with bet365

About bet365

- Founded in 2000
- Located in Stoke-on-Trent, UK
- The largest online sports betting company
- Over 19 million customers
- One of the largest private companies in the UK
- Employs more than 2,000 people
- 2014-2015: Over £34 billion was staked
- Very technology focused company

Message passing

Selective receive

Selective receive

```
receive
  Pattern_1 -> Expr_1;
  Pattern_2 -> Expr_2;
     …
  Pattern_n -> Expr_n
end.
```

Selective receive

```
receive
   Pattern_1 -> Expr_1;
   Pattern_2 -> Expr_2;
      …
   Pattern_n -> Expr_n
end.
```

If incoming msg rate > speed of execution of each msg, queues build up

Do not allow large message queues to build up for any process

Problems with large message queues

Problems with large message queues

- Scanning messages in a mailbox can become time consuming

Problems with large message queues

- Scanning messages in a mailbox can become time consuming
- Processes sending messages incur a reduction count penalty

Suppress unnecessary messages

Suppress unnecessary messages

```
msg_handler() ->
    receive
        {in, From, Msg} ->
            spawn_link(?MODULE,
                       worker,
                      [self(), Msg]),
            msg_handler();
        {result, Result} ->
            send_result(From, Result);
        {'EXIT', Pid, normal} ->
            ok;
        {'EXIT', Pid, Err} ->
            error_handler(Err);
    end.

worker(Parent, Msg) ->
    Res = do_something(Msg),
    Parent ! Res.
```

Suppress unnecessary messages

```
msg_handler() ->
    receive
        {in, From, Msg} ->
            spawn_link(?MODULE,
                       worker,
                      [self(), Msg]),
            msg_handler();
        {result, Result} ->
            send_result(From, Result);
        {'EXIT', Pid, normal} ->
            ok;
        {'EXIT', Pid, Err} ->
            error_handler(Err);
    end.

worker(Parent, Msg) ->
    Res = do_something(Msg),
    Parent ! Res.
```

```
msg_handler() ->
    receive
        {in, From, Msg} ->
            spawn(?MODULE, worker,
                  [From, Msg]),
            msg_handler()
    end.

worker(From, Msg) ->
    case catch do_something(Msg) of
        {'EXIT', Err} ->
            error_handler(Err);
        Result ->
            send_result(From, Result)
    end.
```

Beware of proc_lib:spawn

Beware of proc_lib:spawn

- Crashed processes produce crash reports

Beware of proc_lib:spawn

- Crashed processes produce crash reports
- Crash reports are sent to the error_logger

Beware of proc_lib:spawn

- Crashed processes produce crash reports
- Crash reports are sent to the error_logger
- error_logger is REALLY bad at handling high volume of error reports

Use ETS as a message queue

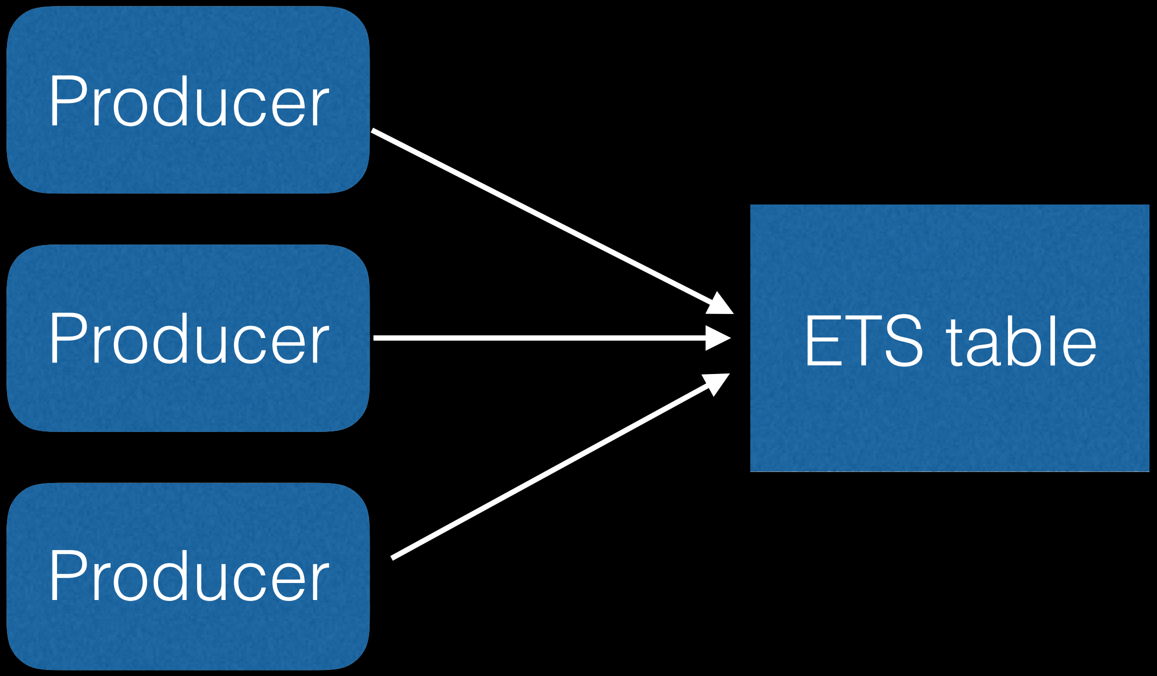
Use ETS as a message queue

Producer

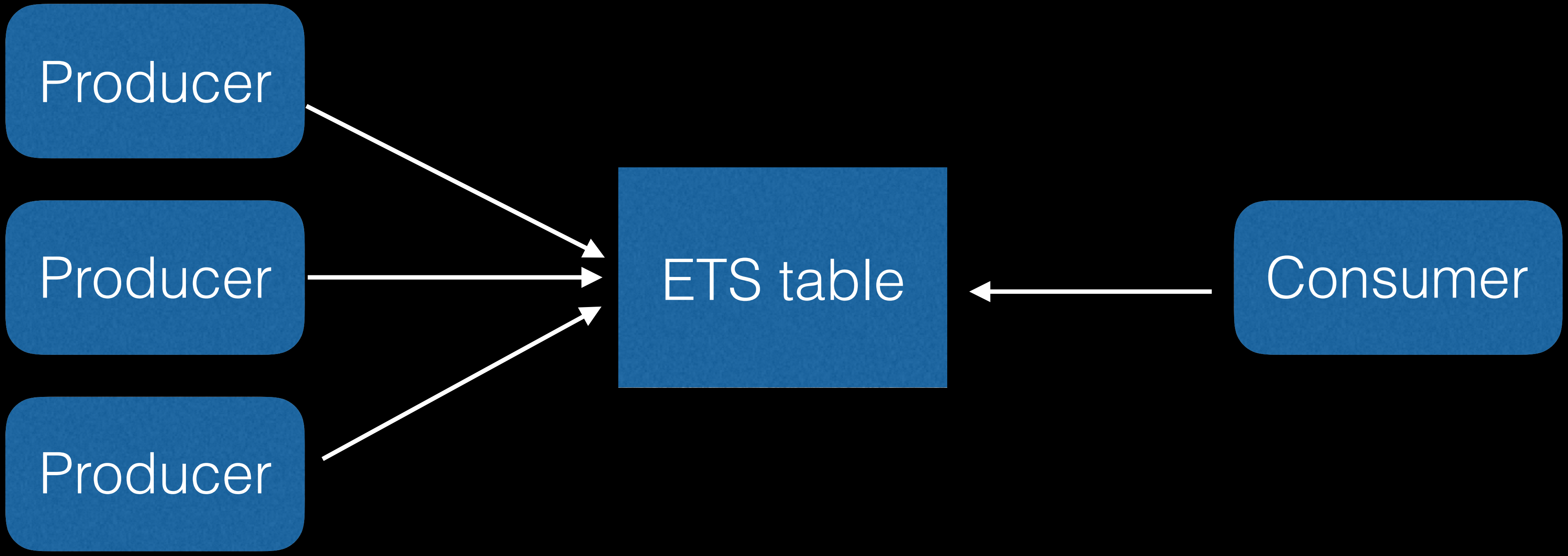
Producer

Producer

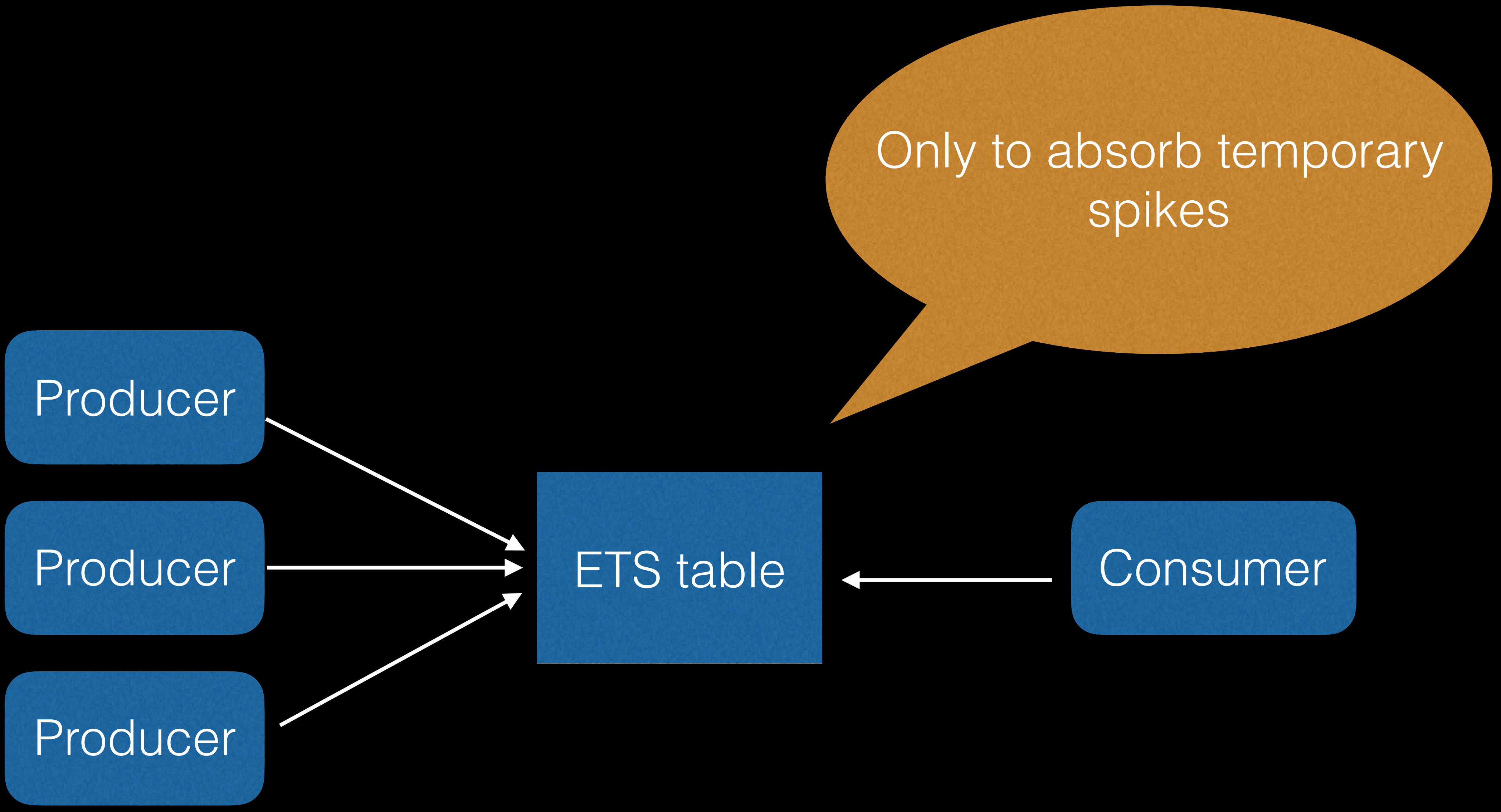
Use ETS as a message queue



Use ETS as a message queue



Use ETS as a message queue



Overload control

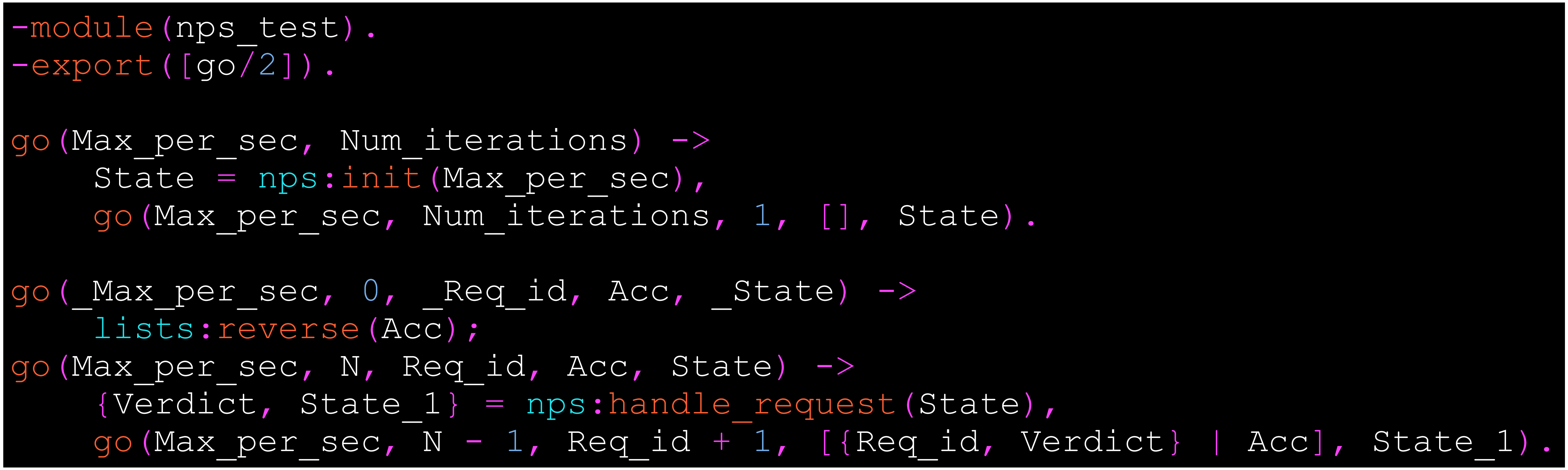
Built-in overload control

- See http://www.erlang.org/doc/man/overload.html
- Problematic because its a gen_server implementation
- Extra message passing
- Caters for ‘global load’ only - not interface specific load

Overload control - example

```erlang
-module(nps).
-export([init/1, handle_request/1]).

-record(nps_state, {max_per_sec, timestamp, cur_vol = 0}).

init(Max_per_sec) ->
    #nps_state{max_per_sec = Max_per_sec,
               timestamp   = timestamp()}.

timestamp() ->
    erlang:monotonic_time(seconds).

handle_request(
  #nps_state{cur_vol       = C,
             timestamp     = Prev_time,
             max_per_sec   = Max_per_sec} = State) ->
    Cur_time = timestamp(),
    case Prev_time < Cur_time of
        true ->
            {allow, State#nps_state{cur_vol = 1, timestamp = Cur_time}};
        false when C >= Max_per_sec ->
            {deny, State};
        false ->
            {allow, State#nps_state{cur_vol = C + 1}}
    end.
```

Overload control - test

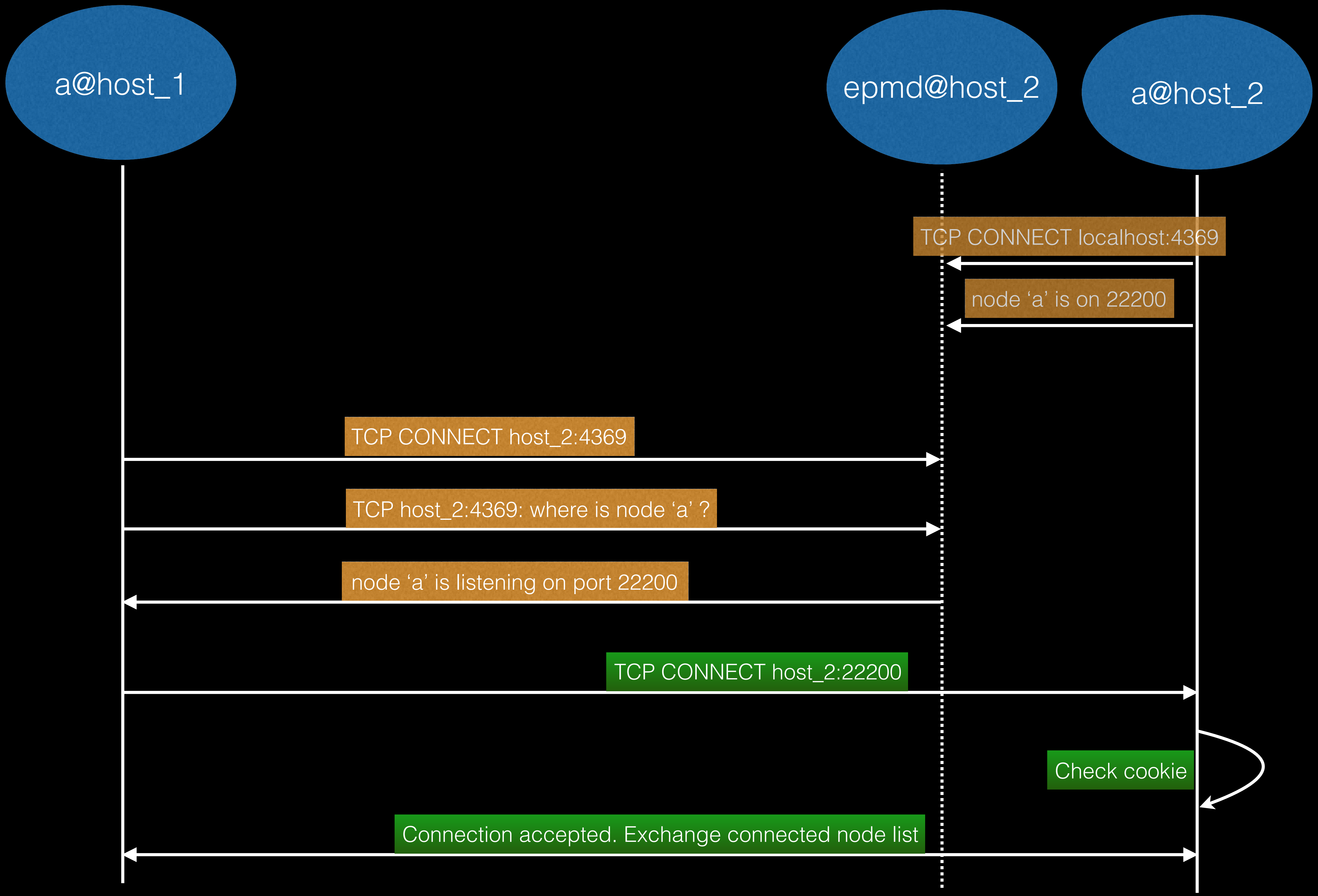


```erlang
-module(nps_test).
-export([go/2]).

go(Max_per_sec, Num_iterations) ->
    State = nps:init(Max_per_sec),
    go(Max_per_sec, Num_iterations, 1, [], State).

go(_Max_per_sec, 0, _Req_id, Acc, _State) ->
    lists:reverse(Acc);
go(Max_per_sec, N, Req_id, Acc, State) ->
    {Verdict, State_1} = nps:handle_request(State),
    go(Max_per_sec, N - 1, Req_id + 1, [{Req_id, Verdict} | Acc], State_1).
```

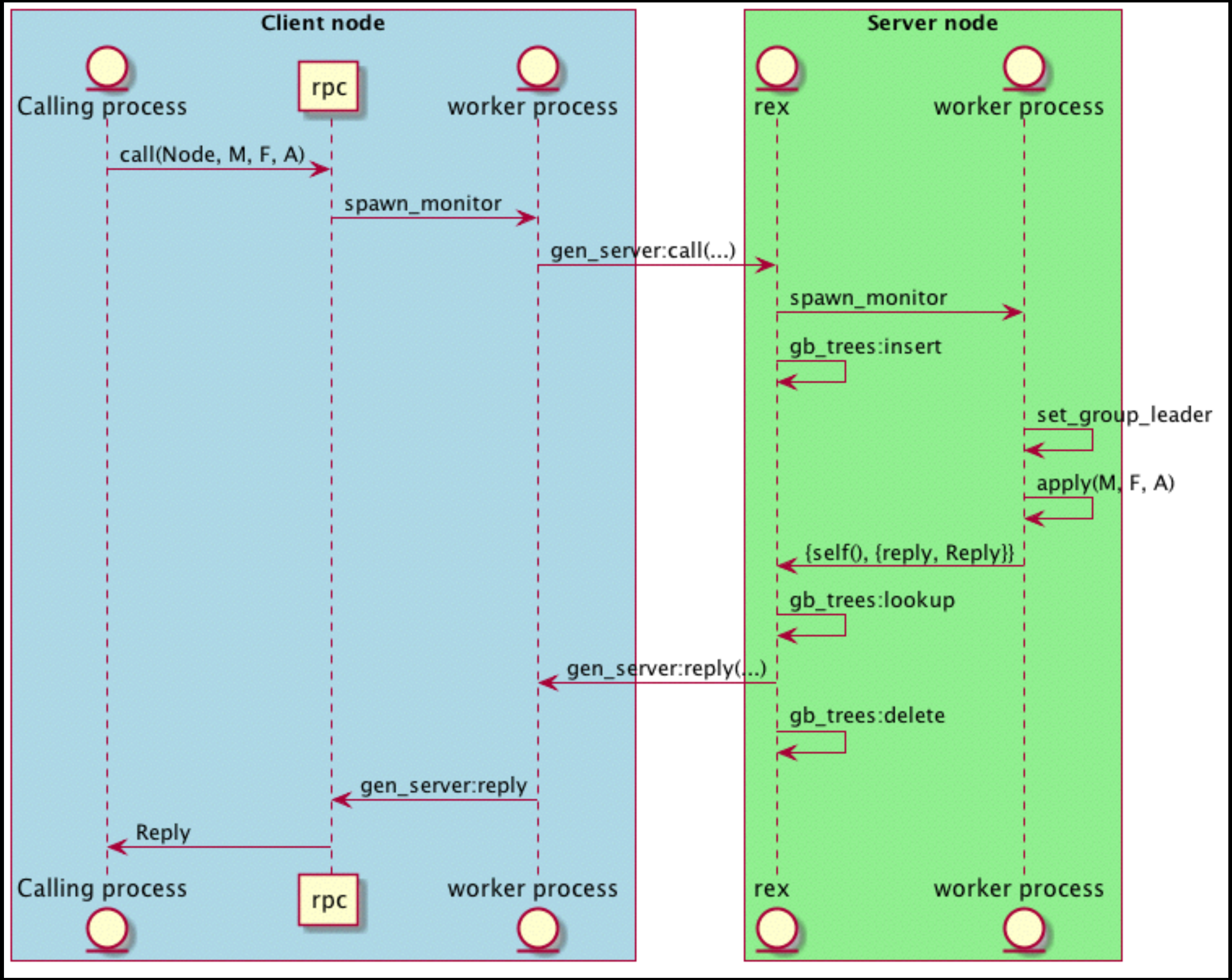
```
6> nps_test:go(10, 15).
[{1,allow}, {2,allow},
 {3,allow}, {4,allow},
 {5,allow}, {6,allow},
 {7,allow}, {8,allow},
 {9,allow}, {10,allow},
 {11,deny}, {12,deny},
 {13,deny}, {14,deny},
 {15,deny}]
```

Native RPC

Naming and Discovery


a@host_1
epmd@host_2
a@host_2
TCP CONNECT localhost:4369
node ‘a’ is on 22200
TCP CONNECT host_2:4369
TCP host_2:4369: where is node ‘a’ ?
node ‘a’ is listening on port 22200
TCP CONNECT host_2:22200
Check cookie
Connection accepted. Exchange connected node list

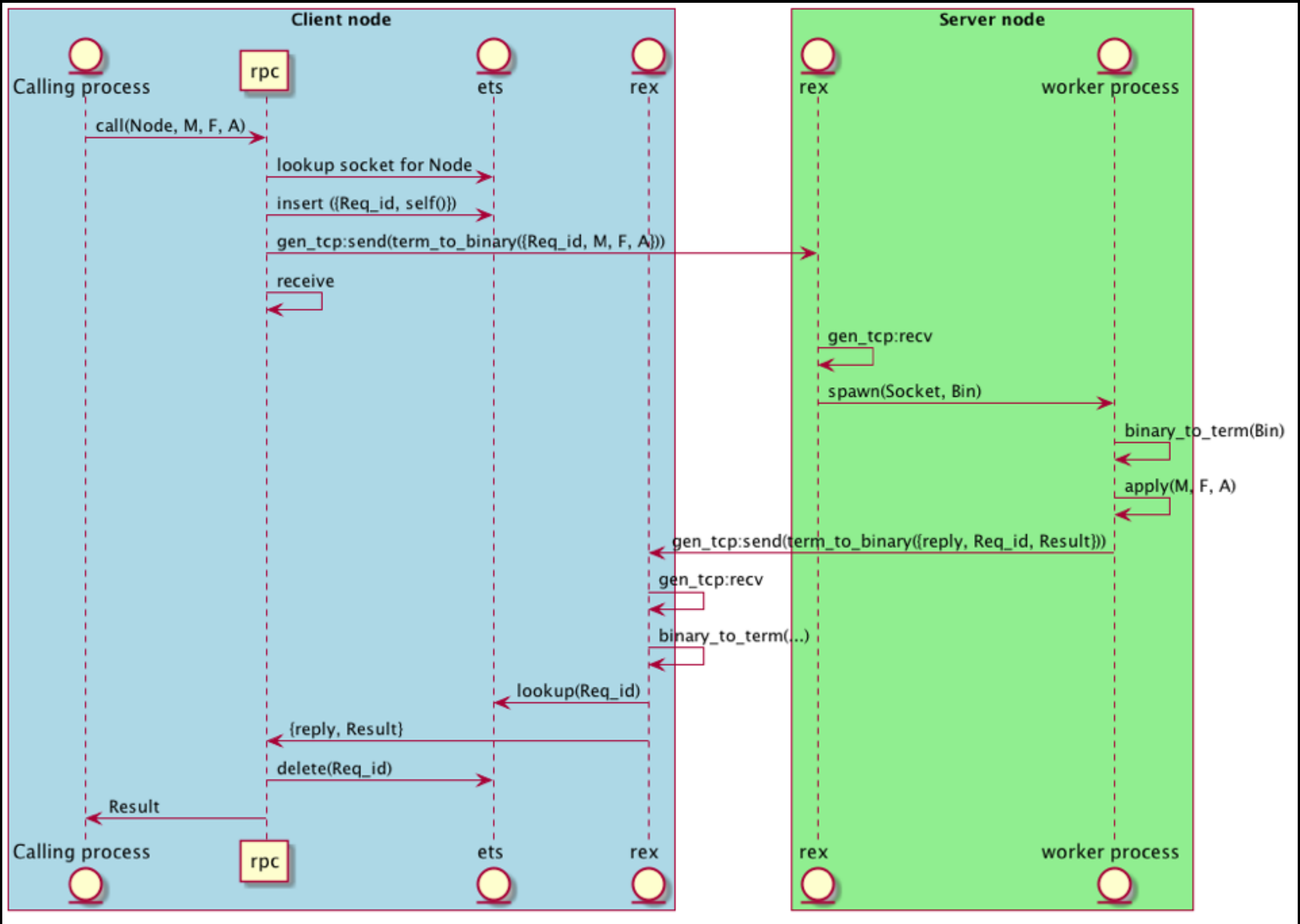
Native RPC - internals



Limitations of native RPC

- No overload control on the server side
- ‘rex’ is a message queue hotspot
- Inefficient implementation
- Head-of-line blocking problem, potentially delaying net_kernel heartbeats

A more efficient RPC



Advantages of proposed mechanism

- Possible to introduce overload control
- Can use a different transport protocol (e.g. SCTP)
- Clean load balancing and failure handling
- Use multiple connections
- Workaround head-of-line blocking problem

Long-lived stateful processes

- Harder to implement correctly
- Garbage collection issues
- More effort required to get correct supervision strategy

Mnesia

Mnesia

- Built-in KV store
- Supports ACID transactions
- Supports real-time replication of tables

Mnesia - table types

- 3 types of tables
  - ram_copies
  - disc_copies
  - disc_only_copies

Mnesia - table management

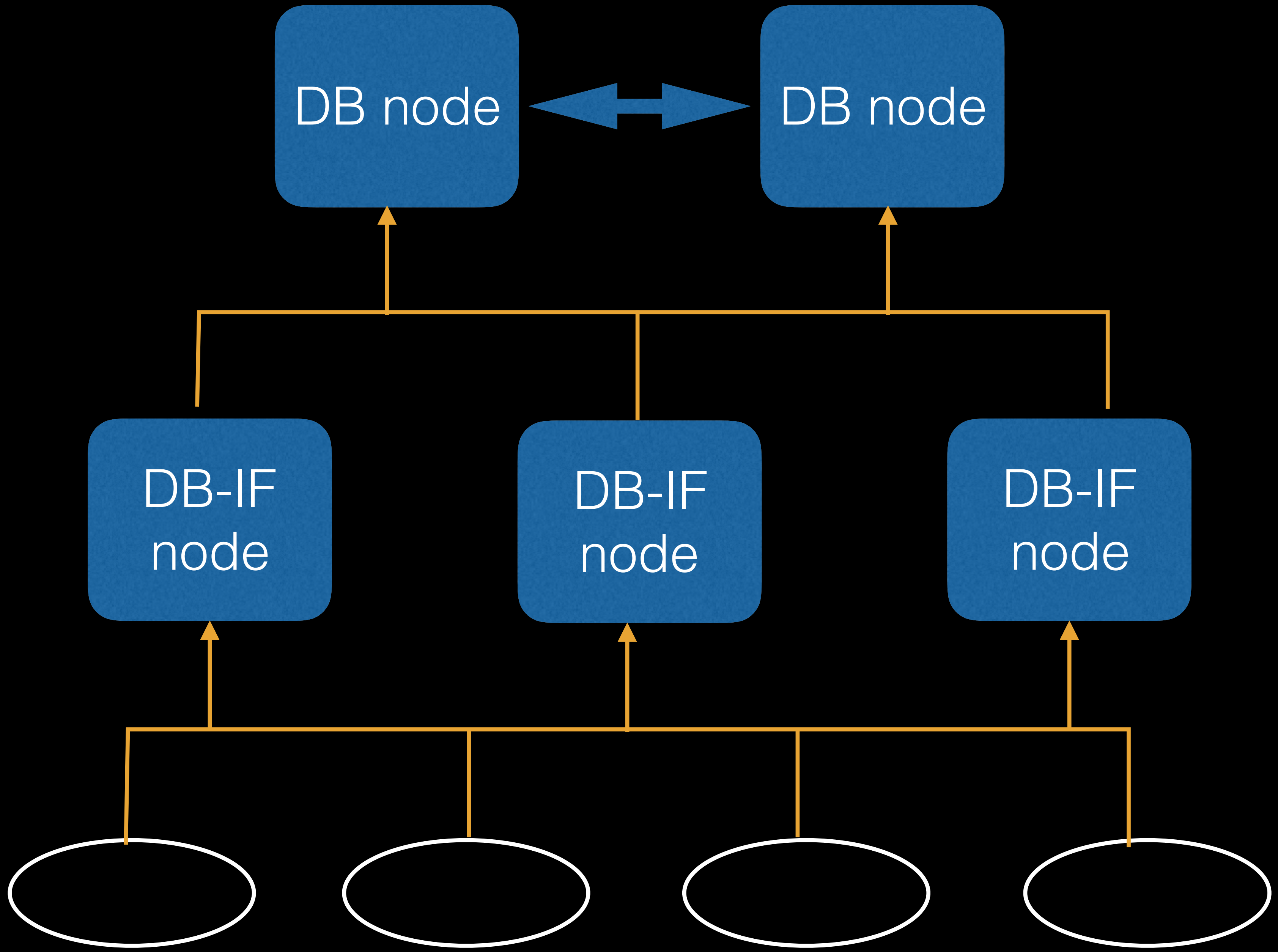
- Data in a table is stored in a <Table>.DCD file
- All modifications to persistent tables are written to LATEST.log
- ‘Occasionally’, contents of LATEST.log are written to <Table>.DCL files
- ‘Occasionally’, contents of <Table>.DCL are dumped to <Table>.DCD

Mnesia - problems

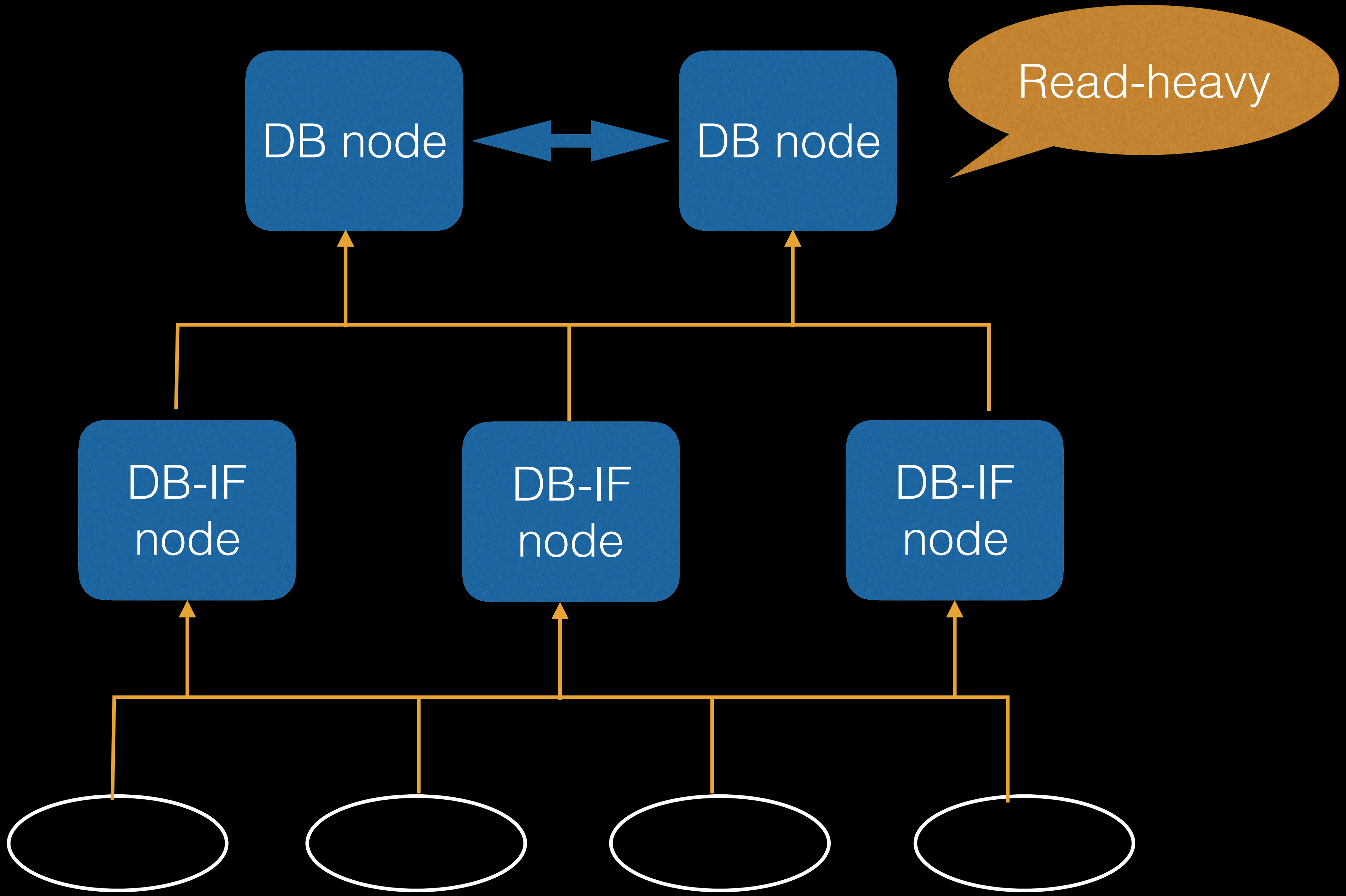
- Table management on disk leads to Mnesia overload for write-heavy applications
- Net-splits are resolved by restarting nodes (data loss)

Mnesia - where it worked

Mnesia - where it worked

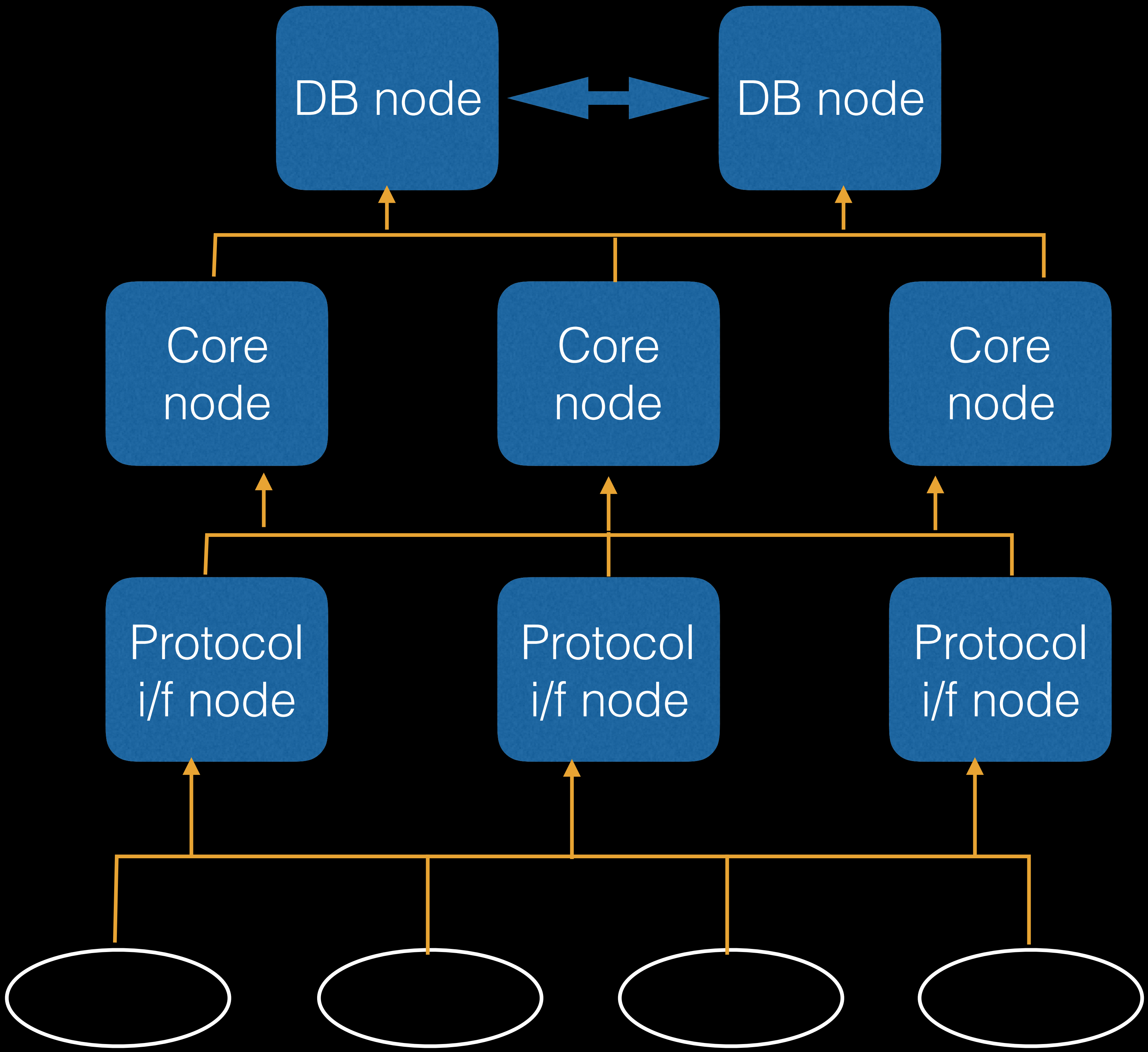


Mnesia - where it worked

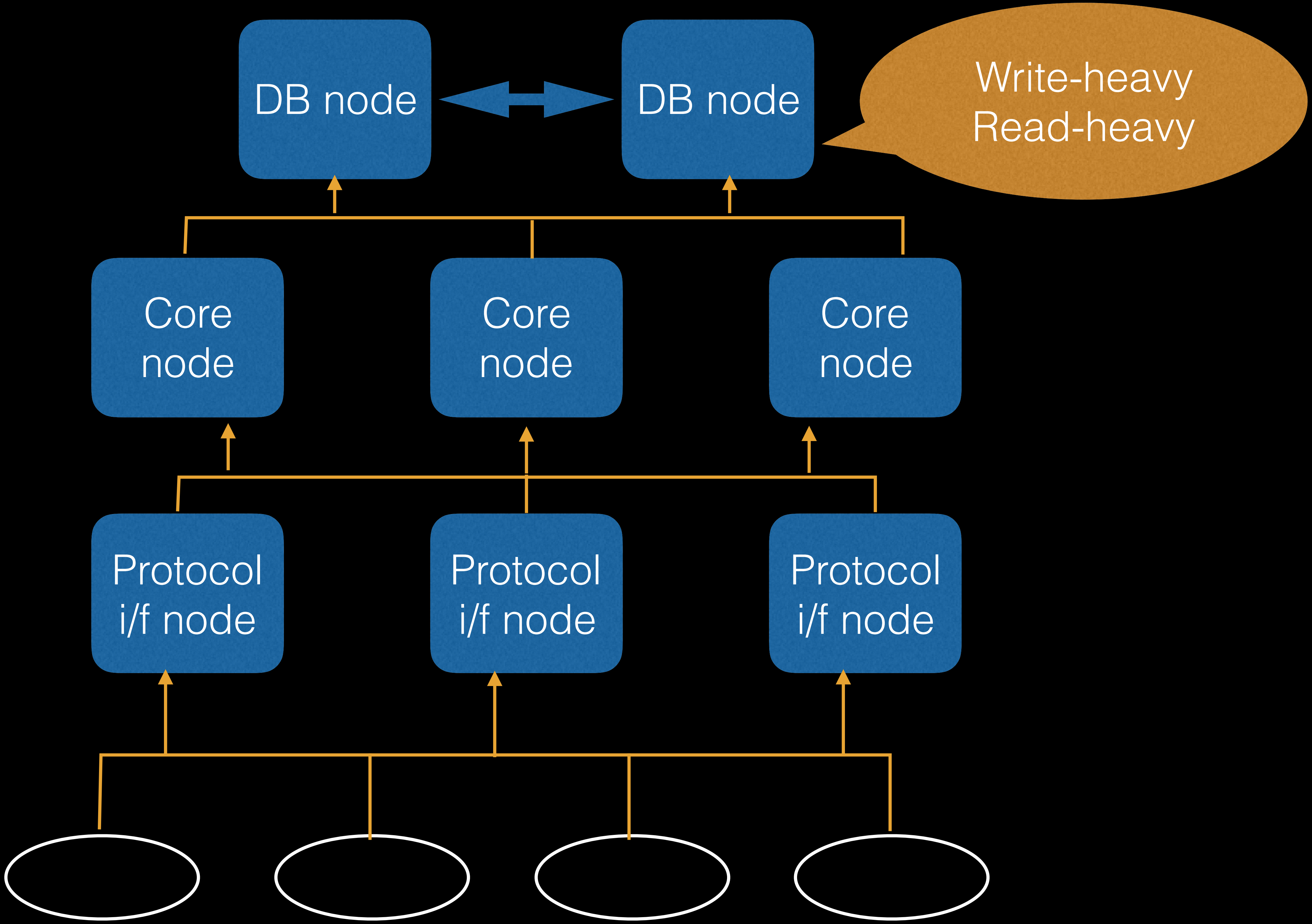


Mnesia - where it didn’t work

Mnesia - where it didn’t work



Mnesia - where it didn't work



Don’t use Mnesia in a replicated write-heavy use case

Don’t use Mnesia in a replicated write-heavy use case

Replicated read-heavy is OK

Don’t use Mnesia in a replicated write-heavy use case

Replicated read-heavy is OK

Standalone write-heavy is OK

Hot code loading

Hot code loading - considerations

Hot code loading - considerations

- Process state management

Hot code loading - considerations

- Process state management
- Installation & Rollback

Hot code loading - considerations

- Process state management
- Installation & Rollback
- Traceability

Don’t use hot code loading to patch your systems unless you have automated installation and rollback scripts

TCP sockets

TCP sockets

- Do NOT use {active, true} mode on sockets in production

TCP sockets

- Do NOT use {active, true} mode on sockets in production

- {active, once} is safest

TCP sockets

- Do NOT use {active, true} mode on sockets in production

A spinlock is acquired in the Linux kernel for every read

- {active, once} is safest

TCP sockets

- Do NOT use {active, true} mode on sockets in production

A spinlock is acquired in the Linux kernel for every read

- {active, once} is safest

- {active, N} seems to yield the highest performance

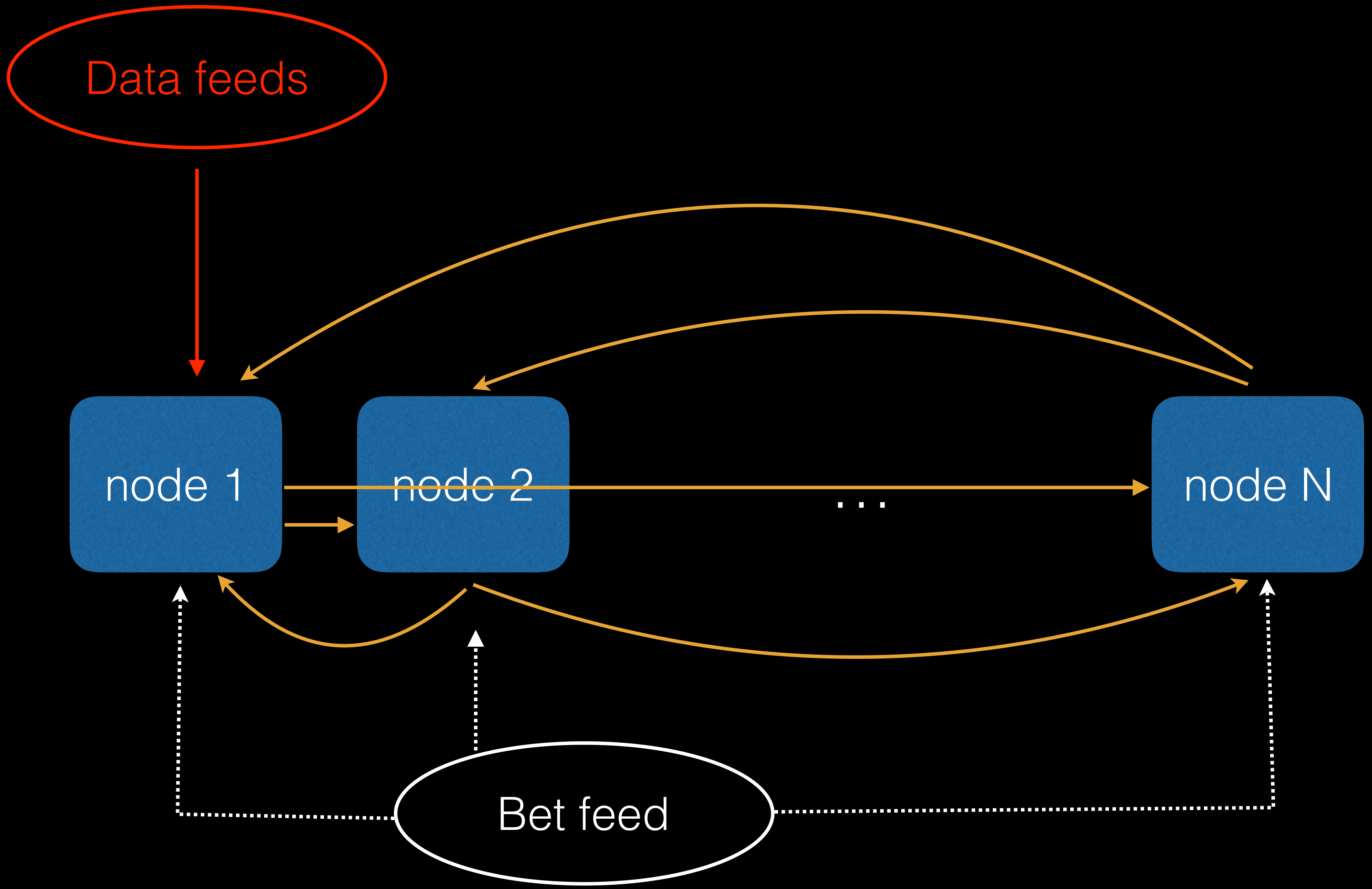
Overall system design

Design guidelines

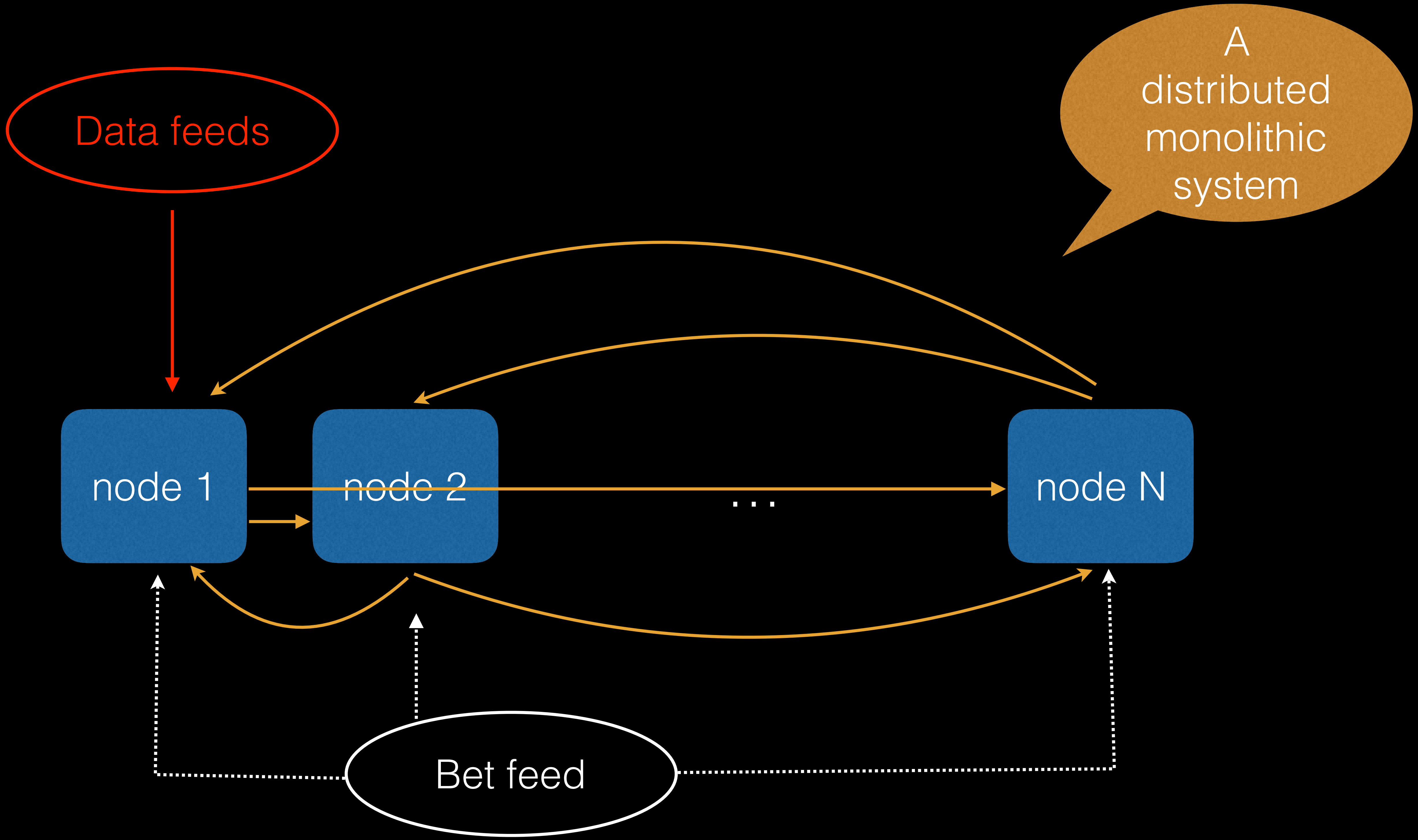
- Make each Erlang node as independent as possible
- Each node should be a independent unit of computation

Bad design - example

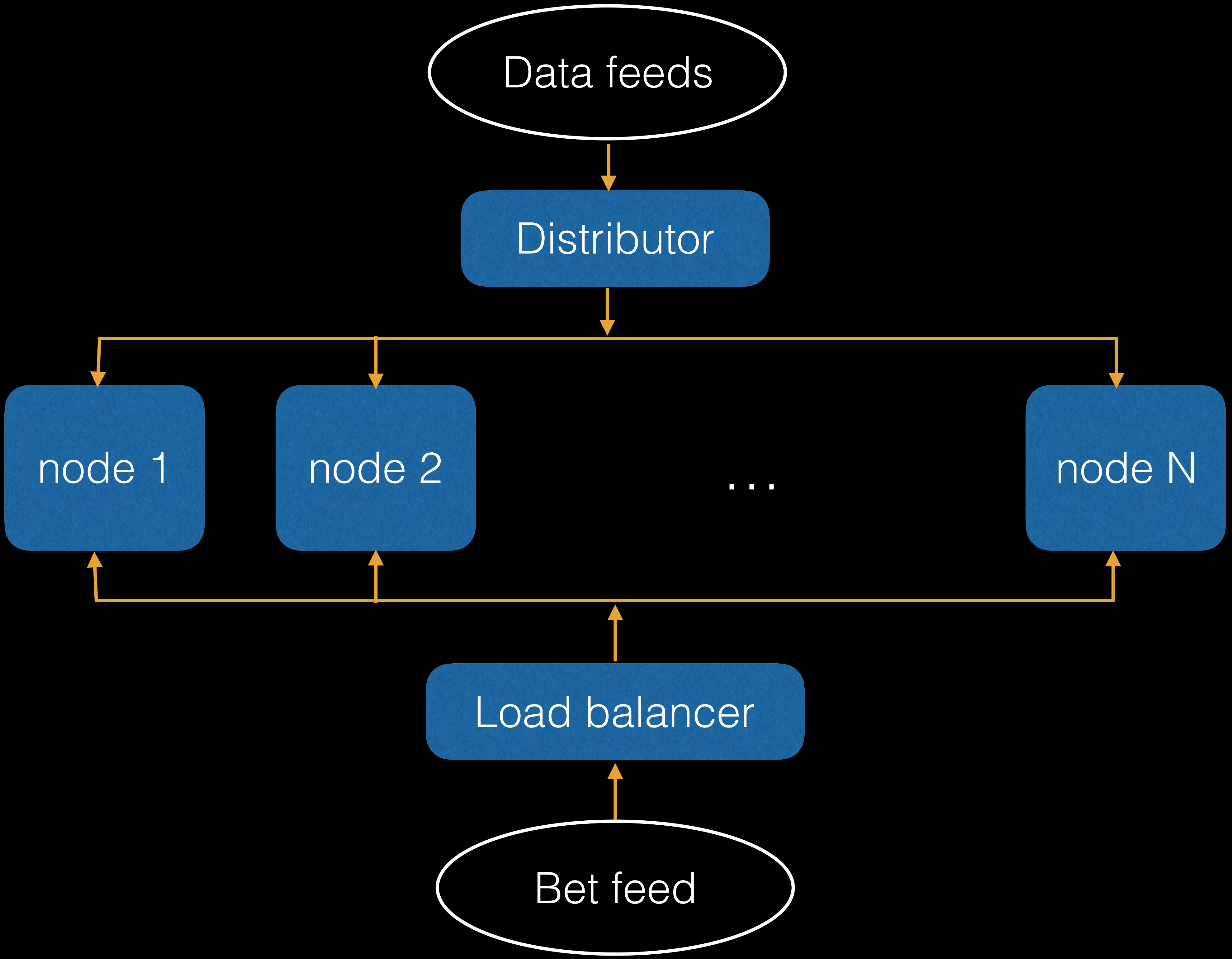
Bad design - example



Bad design - example



A better way


Data feeds
Distributor
node 1
node 2
. . .
node N
Load balancer
Bet feed

Recap

Recap

- Prefer lots of short lived stateless processes over a few long lived ones

Recap

- Prefer lots of short lived stateless processes over a few long lived ones
- Beware of message queue build up

Recap

- Prefer lots of short lived stateless processes over a few long lived ones
- Beware of message queue build up
- Beware of native RPC limitations

Recap

- Prefer lots of short lived stateless processes over a few long lived ones
- Beware of message queue build up
- Beware of native RPC limitations
- Mnesia is awesome (for certain use cases)

Recap

- Prefer lots of short lived stateless processes over a few long lived ones
- Beware of message queue build up
- Beware of native RPC limitations
- Mnesia is awesome (for certain use cases)
- {active, N} works best for TCP sockets

Recap

- Prefer lots of short lived stateless processes over a few long lived ones
- Beware of message queue build up
- Beware of native RPC limitations
- Mnesia is awesome (for certain use cases)
- {active, N} works best for TCP sockets
- Overload control is not optional

Recap

- Prefer lots of short lived stateless processes over a few long lived ones
- Beware of message queue build up
- Beware of native RPC limitations
- Mnesia is awesome (for certain use cases)
- {active, N} works best for TCP sockets
- Overload control is not optional
- Pay attention to overall system design

Open source at bet365

Open source at bet365

- Better ODBC support

Open source at bet365

- Better ODBC support
- A proper SOAP implementation in Erlang

Open source at bet365

- Better ODBC support
- A proper SOAP implementation in Erlang
- Assisting Ericsson to develop a package manager for Erlang

Acknowledgements

Source code highlighting: ‘Highlight’ courtesy of Andre Simon
http://www.andre-simon.de/dokuwiki/doku.php