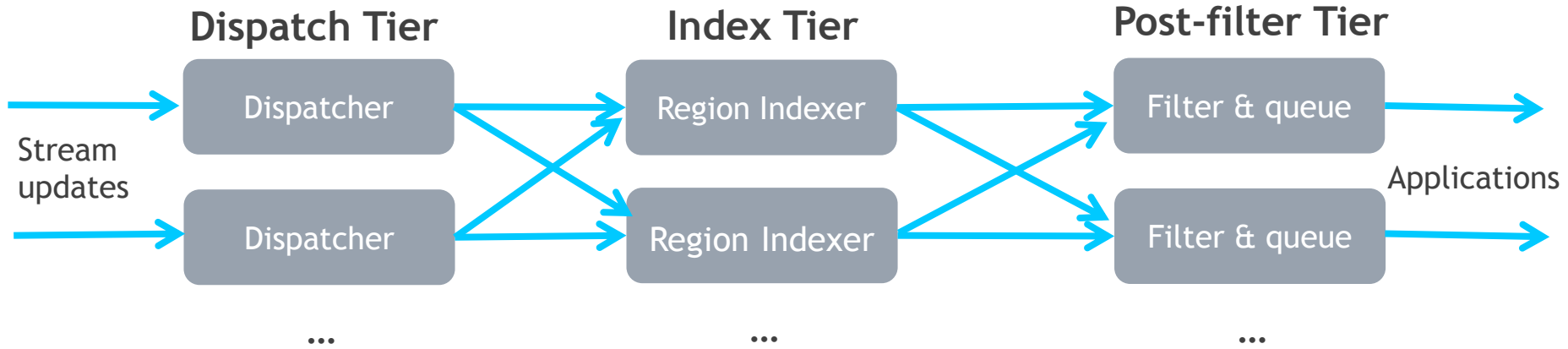




building a scalable geofence platform based on RabbitMQ custom exchanges

- Philippe Dobbelaere, Nokia Bell Labs Antwerp
(philippe.dobbelaere@nokia.com)
- 29-01-2016

Three-tier Architecture

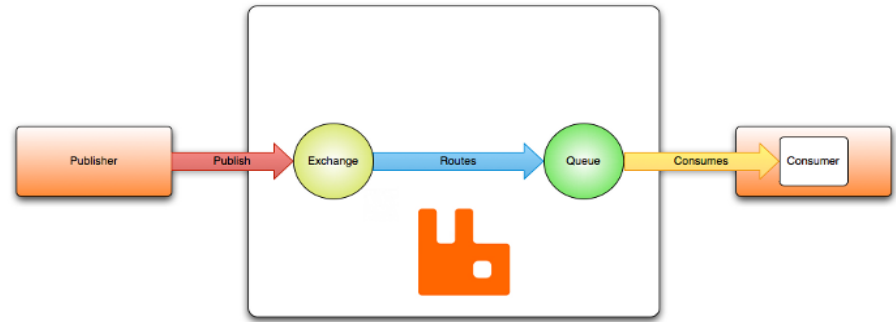


Goal: architect for horizontal scalability

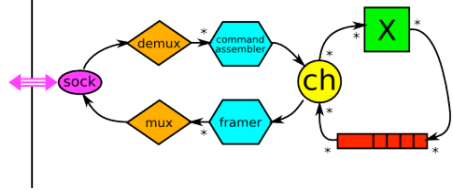
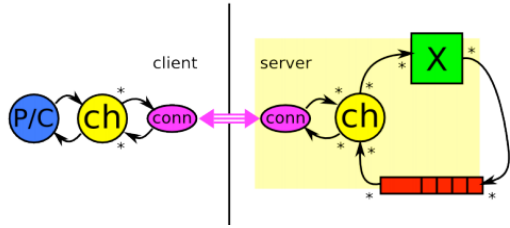
AMQP FORWARDING concepts

- messages are published to an exchange
- exchanges either forward or drop the messages based on a routing decision
= no buffering
- exchanges route to exchanges or queues
binding = $f(\text{key})$
- queues can buffer messages
durable queues even persist messages
- messages are consumed from queues
- a queued message can only be consumed once

"Hello, world" example routing



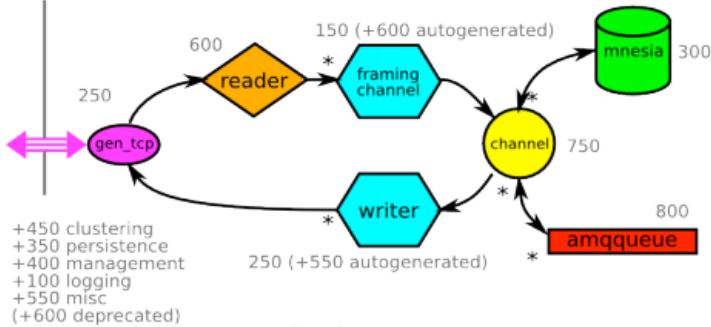
RabbitMQ CONCEPTS



three different areas of parallelism

- 8-stage pipeline end-to-end between **producer and consumer**
- parallelism across **connections** and **channels** - with **queues** acting as synchronisation points (queues are about the only stateful part of the core AMQP model)
- **queues** as active erlang processes

(excluding comments and blank lines)



+450 clustering
 +350 persistence
 +400 management
 +100 logging
 +550 misc
 (+600 deprecated)

... 4950 + 1150 autogenerated + 600 deprecated

RABBITMQ exchange IMPLEMENTATION

create(?TX, #exchange { name = XName, durable = Durable, auto_delete = AutoDelete, arguments = Args }) -> **ok**

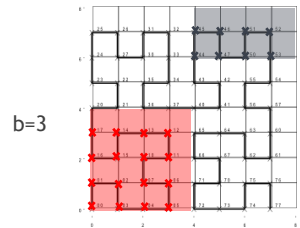
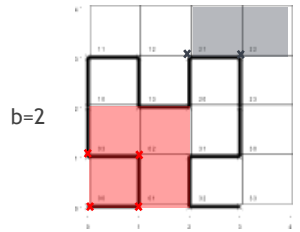
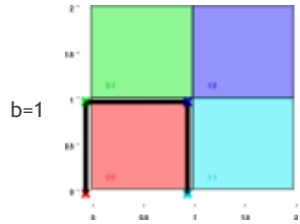
delete(?TX, #exchange { name = XName } , Bindings) -> **ok**

add_binding(?TX, #exchange { name = XName } , Binding) -> **ok**

remove_bindings(?TX, #exchange { name = XName } , Bindings) -> **ok**

route(#exchange{ name = XName } , #delivery{ message = #basic_message { routing_keys = Routes, content = #content { payload_fragments_rev = FragmentsRev } } }) -> **[#exchange or #queue]**

Hilbert space filling curve



not limited to 2D

points that are close on index are also close in n-D

maps both ways between (X, Y) and H_IDX

e.g. with b=1 we get

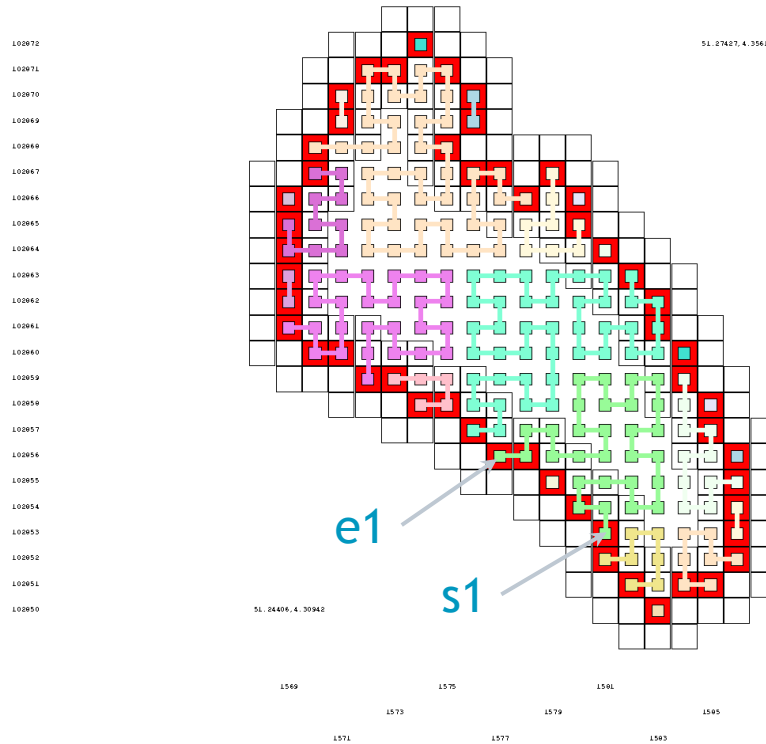
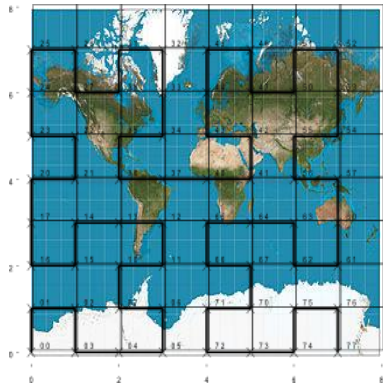
$$(0,0) \Leftrightarrow 0$$

$$(0,1) \Leftrightarrow 1$$

$$(1,1) \Leftrightarrow 2$$

$$(1,0) \Leftrightarrow 3$$

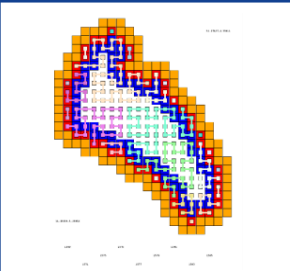
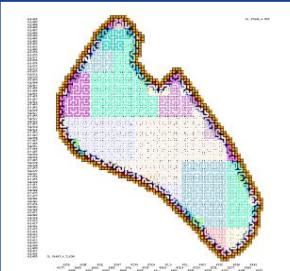
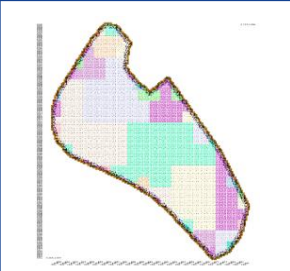
Hilbert 2d Rasterization



[[s1, e1] , [s2, e2], ...]

Resolution



	<p>Hilbert resolution 17 bits polygon points: 77 ranges: 23</p>
	<p>Hilbert resolution 19 bits polygon points: 230 ranges: 82</p>
	<p>Hilbert resolution 20 bits polygon points: 437 ranges: 166</p>

index tier exchange = x-geo

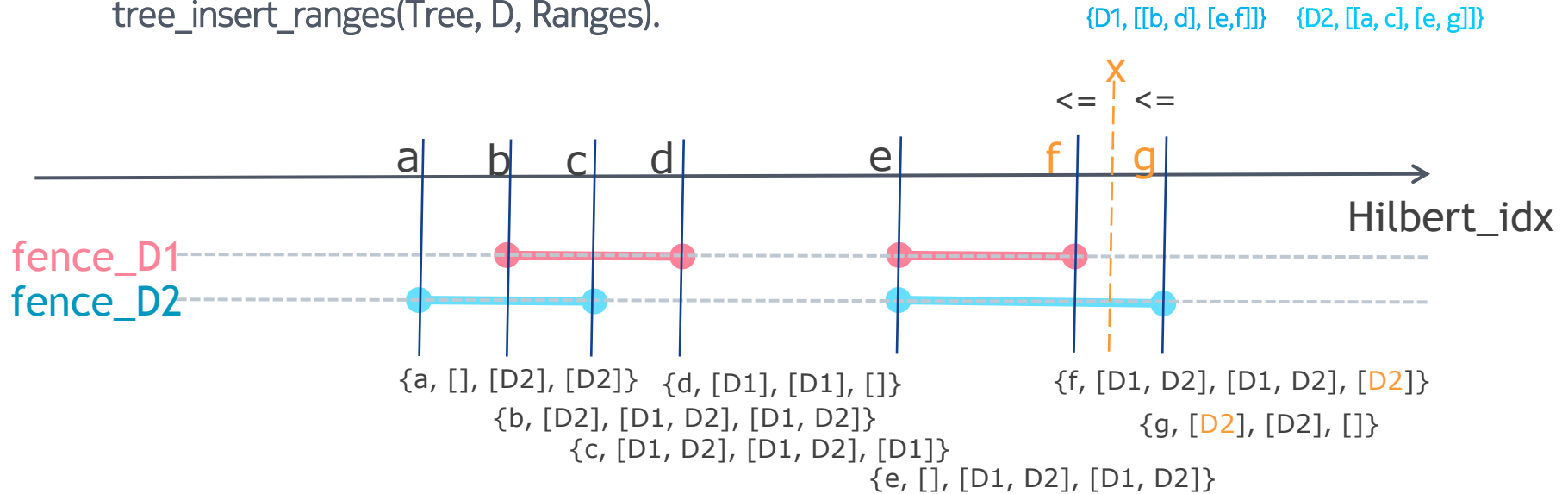
route(...) ->

```
{Status, Hilbert, ObjectId} = extract_Hilbert_Id(Headers),  
FenceDests = geo_dests(Tree, Hilbert).
```

x -> [D2]

add_binding(...) ->

```
tree_insert_ranges(Tree, D, Ranges).
```



erlang NIF for Hilbert index calculation

```
static ERL_NIF_TERM axes_to_line_2D_26b(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {  
    // C code to access erlang arguments and interface to C implementation of algo  
}  
  
static ErlNifFunc nif_funcs[] = {  
    {"axes_to_line_2D_26b", 2, axes_to_line_2D_26b /* , ERL_NIF_DIRTY_JOB_CPU_BOUND */},  
    {"line_to_axes_2D_26b", 1, line_to_axes_2D_26b /* , ERL_NIF_DIRTY_JOB_CPU_BOUND */}  
};
```

- allows to have identical implementations of algo in multiple environments (erlang, nodejs, perl, (c)python, ...) = polyglot approach for algo (rabbitmq was already polyglot approach for messages)
- easier to code algorithms in environments that support mutable structures
- take care
 - this runs on an erlang VM scheduler thread, so should not preempt for more than say 1 ms
 - any SIGSEGV will kill erlang VM

POST-FILTER TIER EXCHANGE = X-FENCE (*stateful* fence)

route(...data...)

```
Dests = rabbit_router:match_routing_key(Rname, ['_']), %% fanout
ets:insert(Times, {{Epoch, ObjectId}}),
ets:insert(Objects, {ObjectId, Epoch}),
ets:insert(Inserts, {Wr_ctr, ObjectId}).
```

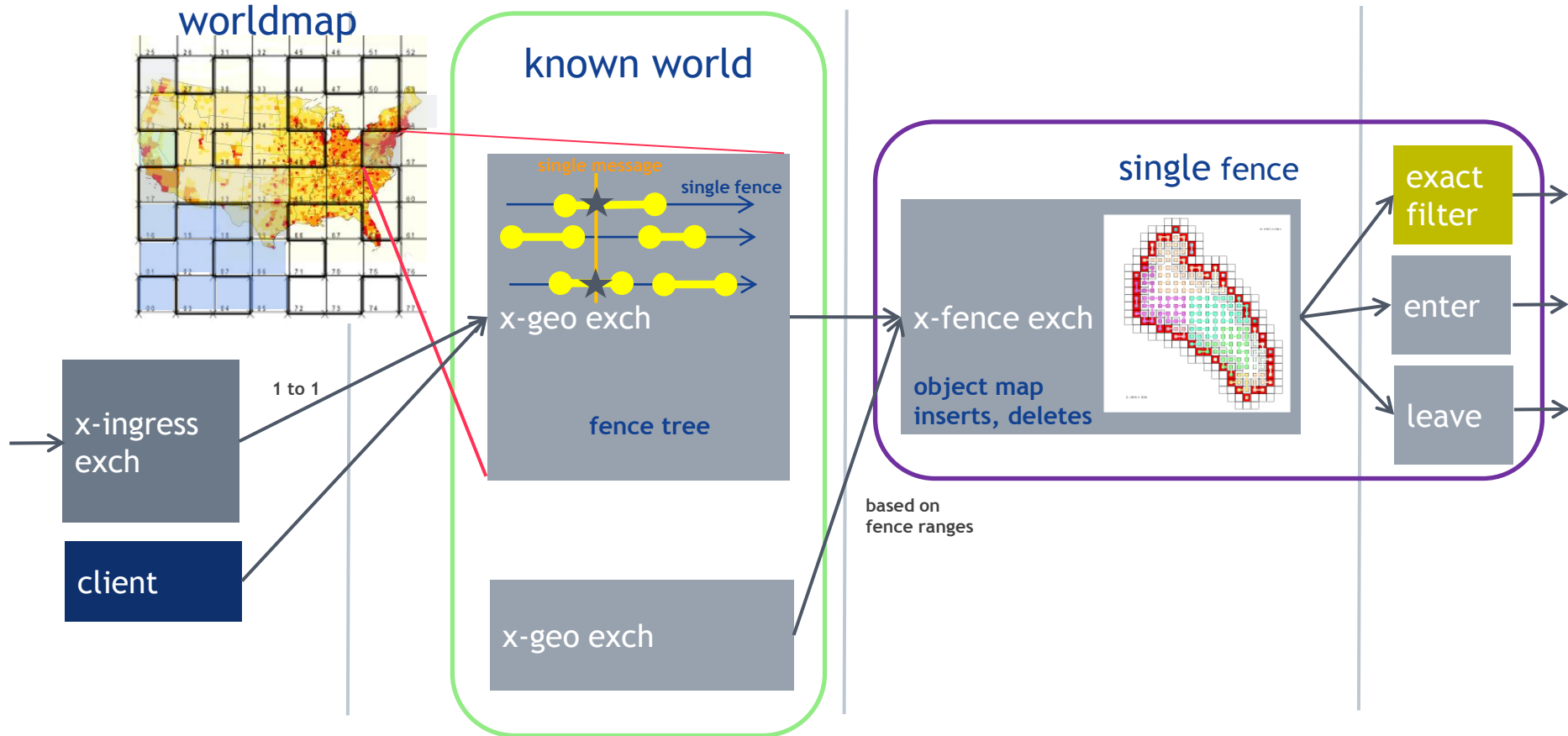
route(...control...)

%% immediately trigger removal from Objects , add to Deletes if notified by x-geo

Objects: current view on ObjectId's that are inside the fence Inserts: list of entering ObjectId's with read/write versioning Deletes: list of leaving ObjectId's with read/write versioning Times: remove from Objects, add to Deletes if no new message within x ms
--

optionally post-filter based on exact polygon so that approximation due to hilbert resolution does not leak through

geofiltering at scale



RabbitMQ GEO FENCES



generic production-grade message broker
custom exchanges

- x-ingress: special "shovel" rabbitmq app that uses worldmap to find destination x-geo (shovel uses amqp based networking, robust in case of flaky networks)
- x-geo: special "fanout" exchange that routes based on hilbert index
- x-fence: special "fanout" exchange that has an objectmap to generate enter and leave messages

custom exchanges are deployed inside a standard rabbitmq broker

- dedicated machine or cloud VM
- researching erlang clustering/networking extensions that work across cloudified environments

erlang OTP robustness and performance features

RabbitMQ amqp based federation concept to wire multiple x-geo to a single x-fence

RabbitMQ clustering difficult across VM hosts

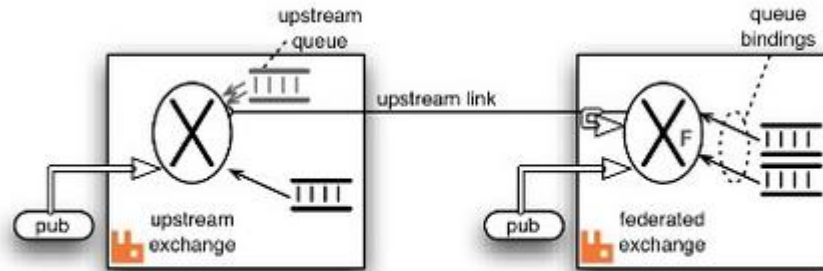
- **epmd** listens on a fixed port (default 4369) and serves a naming service that maps erlang node names to erlang rpc (rex) listener ports.
- within an erlang cluster, the distributed connection protocol of net_kernel will contact epmd and succeeds in setting up a connection to the target erlang node on the port returned by epmd
- when erlang clusters are deployed inside VM environments (or docker containers), ports inside the container are remapped to other ports at the level of the VM host (must be unique at the VM host)
- net_kernel is not aware of this port remapping. To make matters worse, discovering the port mapping often involves an API call that needs the port inside the VM as argument.
- conclusion: without modifications to net_kernel:inet_tcp_dist, clustering will not work across VM hosts
- **solutions:**
 1. **do not use** erlang networking across VM hosts (rabbitmq: federation, shovel uses AMQP/TCP)
 2. **replace inet_tcp_dist** (-proto_dist argument) with a VM aware version

RabbitMQ federation

on the downstream cluster:

```
rabbitmqctl set_parameter federation-upstream my-upstream \  
'{"uri": "amqp://server-name", "expires": 3600000, "exchange": "specific_upstream_exchange"}'  
rabbitmqctl set_policy --apply-to exchanges federate-me "^amq\." \ {"federation-upstream": "my-upstream"}
```

- all exchanges matching "`^amq\.`" will get an associated upstream exchange.
- messages published to the upstream exchanges are copied to the federated exchange



fence and exchange management examples

creating a fence

- adding x-fence exchange with bindings to x-geo expressing the Hilbert index ranges of this fence (if multiple x-geo: select bindings for that x-geo, use rabbitmq federation to collect all messages)

adding an x-geo instance by splitting off a region from an existing x-geo (scaling out)

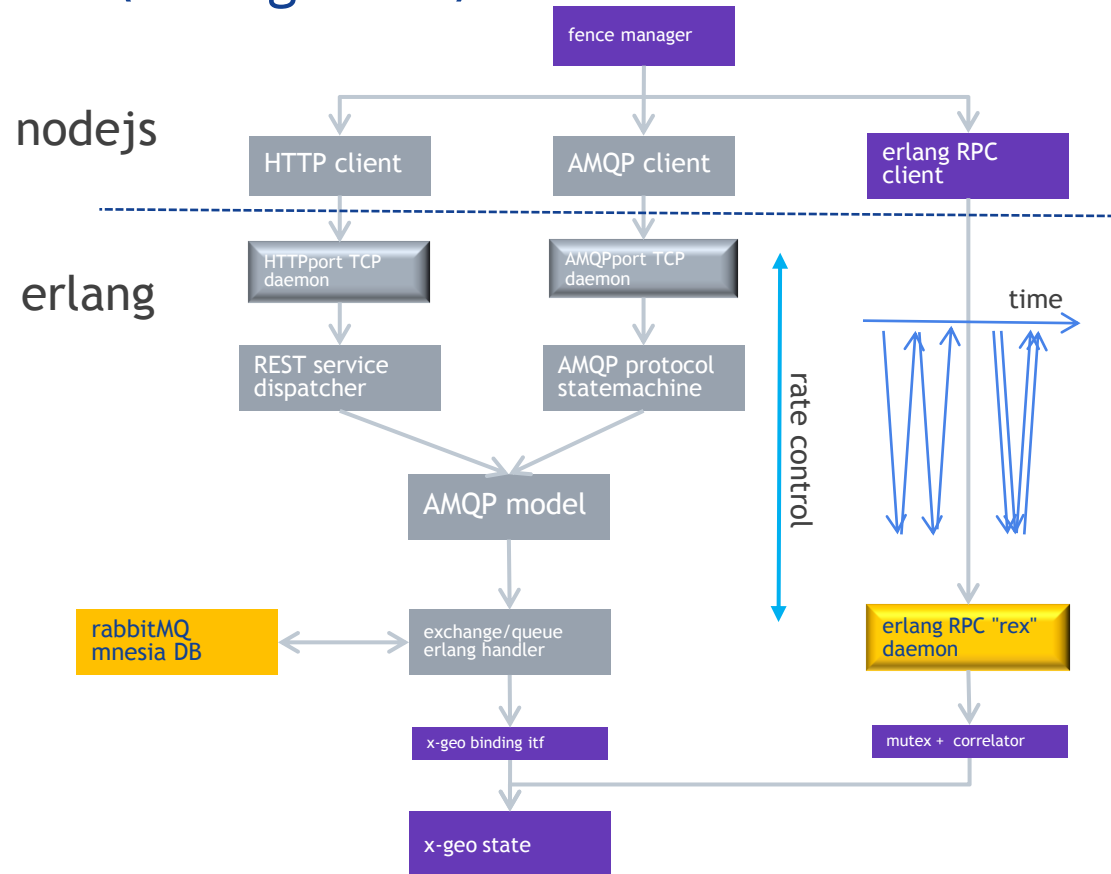
- create bindings on the new x-geo for all existing x-fence exchanges that are bound to the split region
- adapt the worldmap so that split region is routed towards new x-geo
- remove bindings to existing x-geo

removing an x-geo instance by joining a region to another x-geo (scaling in)

- create bindings on the target x-geo for all existing x-fence exchanges on the x-geo that will shut down
- adapt the worldmap so that region is routed towards target x-geo
- remove bindings on x-geo that shuts down, remove x-geo

fence management API alternatives (moving fences)

- multithreaded AMQP daemon
- multithreaded HTTP daemon
- multithreaded Erlang RPC daemon
 - blocking client (accumulates latency)
 - callback driven client (single latency)



NOKIA