Scalable Distributed Erlang

Natalia Chechina
and RELEASE Team

March 6, 2014

Outline

RELEASE Aim

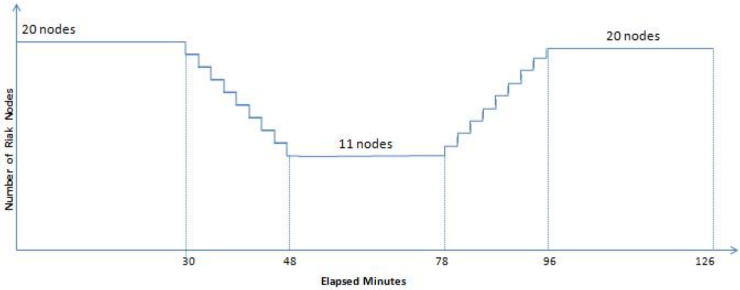
To scale the radical actor (concurrency-oriented) paradigm to build reliable general-purpose software, such as server-based systems, on massively parallel machines (10^5 cores).

Doesn't Erlang already provide scalable reliability? Erlang/OTP has an inherently scalable computation and reliability models, but in practice scalability is constrained (2011):

- **VM aspects**, e.g. synchronisation on internal data structures
- **Language aspects**, e.g. maintaining a fully connected network of nodes, explicit process placement
- **Tool support**

Why Distributed Erlang?

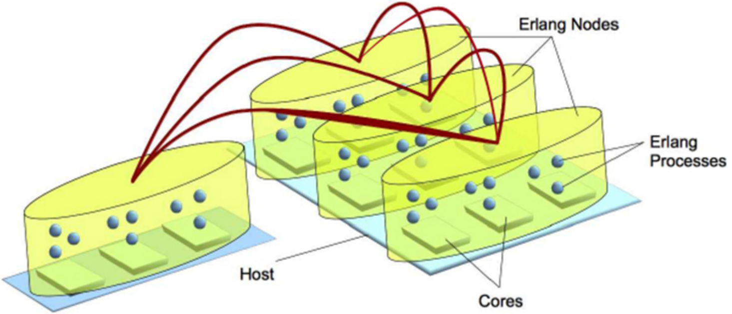
- **Reliability:** multiple hardware and software redundancy means that if one Host or Node fails, other Nodes can continue to deliver service
- **Scalability:** can only scale to around 100 cores on one Host (Node). Many systems use 1000s or 10000 cores



Distributed Erlang

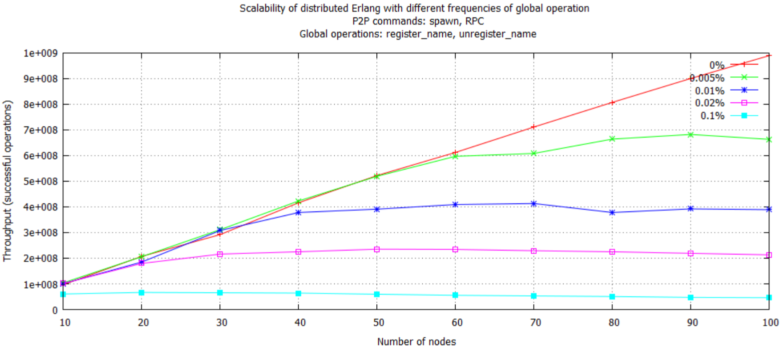
- Transitive connections
- Explicit Placement, i.e.

```
spawn(Node, Module, Function, Args) → pid()
```



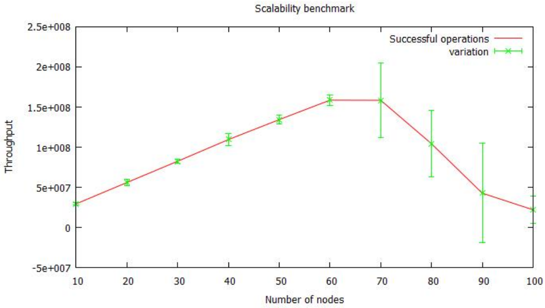
Distributed Erlang Scalability Limitations

- Global operations, i.e. registering names using `global` module
- Other global operations, e.g. using `rpc:call` to call multiple nodes



Distributed Erlang Scalability Limitations

- Single process bottlenecks, e.g. overloading `rpc`'s `rex` process
- All-to-all connections (no evidence yet)



Why Orbit[LN01]?

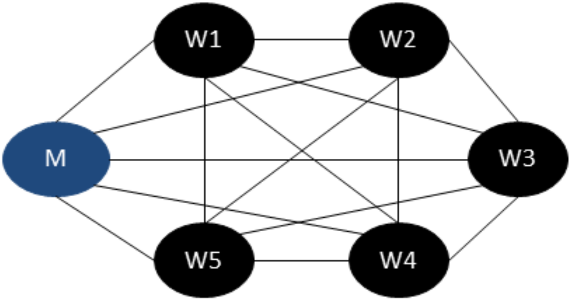
- Uses a Distributed Hash Table (DHT) similar to NoSQL DBMSs like Riak [Bas13], i.e. the hash of a value defined where the value should be stored
- Uses standard P2P techniques and credit/recovery distributed termination detection algorithm [MC98]
- Is only a few hundred lines and has a good performance and extensibility

Orbit in Distributed Erlang

Main components: `master.erl`, `worker.erl`, `table.erl`, `credit.erl`

× Pid = spawn_link(worker, init, [TabSize, TmOut, SpawnImgComp])

✓ Pid = spawn_link(Node, worker, init, [TabSize, TmOut, SpawnImgComp])



Design Principles

General:

- Working at Erlang level as far as possible
- Preserving the Erlang philosophy and programming idioms
- Minimal language changes

Reliable Scalability Design Principles:

- Avoiding global sharing
- Introducing an abstract notion of communication architecture
- Keeping Erlang reliability model unchanged as far as possible

SD Erlang Design Approach

- Typical hardware architecture
- Anticipated failures
- Need to scale
  - Persistent data structures (Riak, Casandra)
  - In-memory data structures (Uppsala University, Ericsson)
  - **Computation**

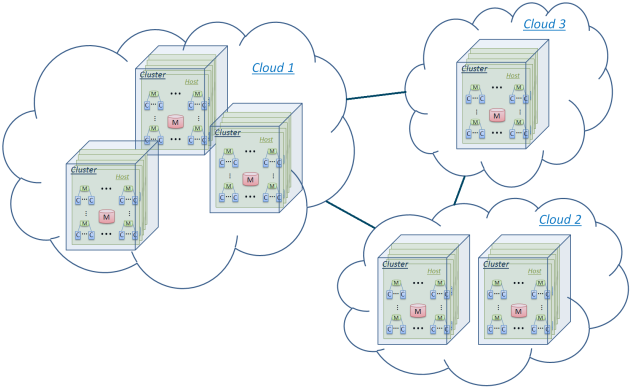
Target Platforms

We target reliable scalable general purpose computing on stock heterogeneous platforms, i.e.

- general server-side computation, like a messaging server
- standard hardware, operating systems and middleware
- not specialised high-performance computing hardware/software stacks

Typical Target Architecture - 10^5 cores

Non-uniform communication
(Level0 – same host, Level1 – same cluster, etc)



Scaling Computation

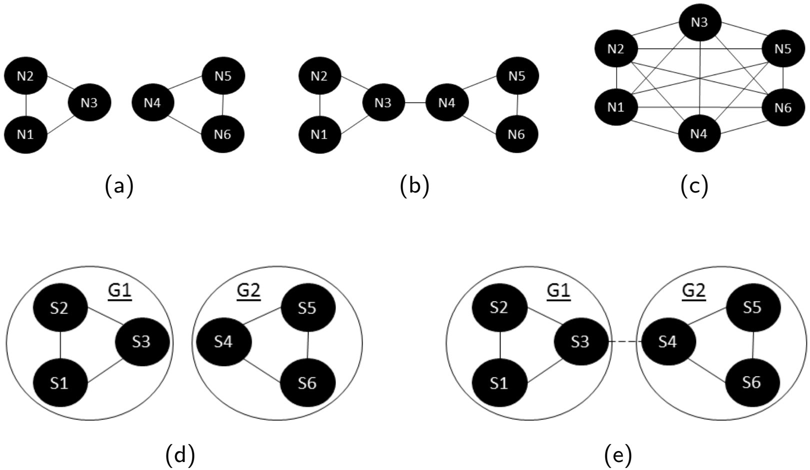
SD Erlang is a small conservative extension of Distributed Erlang

- **Network Scalability**
  - All-to-all connections are not scalable onto 1000s of nodes
  - *Aim:* Reduce connectivity
- **Semi-explicit Placement**
  - Becomes not feasible for a programmer to be aware of all nodes
  - *Aim:* Automatic process placement in groups of nodes

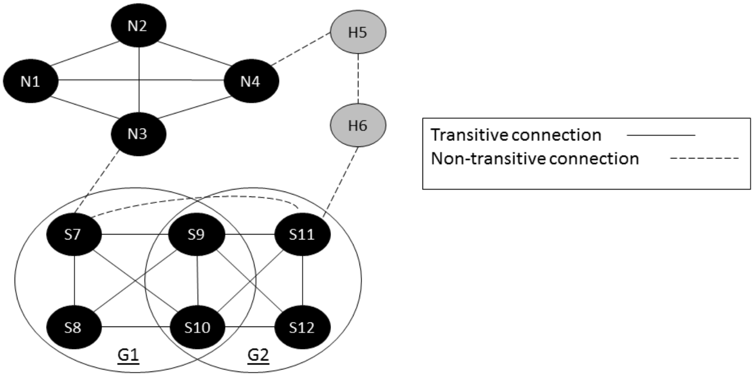
Network Scalability

- Types of nodes
  - Free nodes (normal or hidden) belong to *no s_group*
  - S_group nodes belong to *at least one s_group*
- Nodes in an s_group have transitive connections only with nodes from the same s_groups, but non-transitive connections with other nodes

Free Node Connections vs. S_group Node Connections



Types of Connections between Different Types of Nodes



Why s_groups?

Requirements to the node grouping approach

- Preserve the distributed Erlang philosophy, i.e. any node can be directly connected to any other node
- Adding and removing nodes from groups should be dynamic
- Nodes should be able to belong to multiple groups
- The mechanism should be simple

A list of considered approaches

- Grouping nodes according to their hash values
- A hierarchical approach
- *Overlapping s_groups*

S_group Functions

S_groups can be started

- At launch using `-config` flag and a `.config` file
- Dynamically using `s_group:new_s_group/0,1` functions

Main Functions

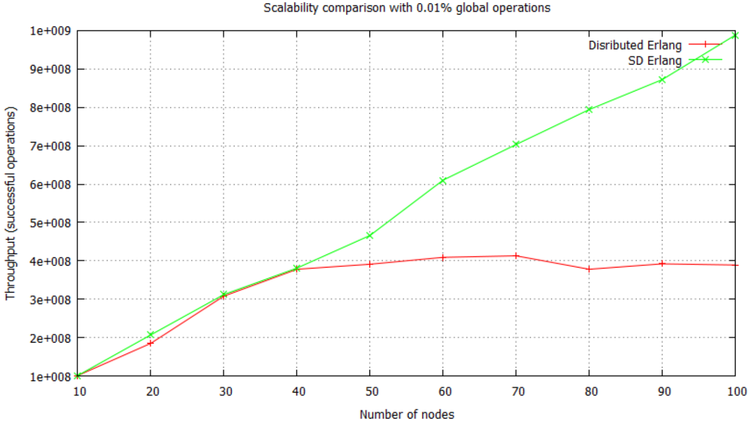
```
new_s_group([Node]) → {SGName, Nodes} | {error, Reason}
new_s_group(SGName, [Node]) → {SGName, Nodes} | {error, Reason}
delete_s_group(SGName) → 'ok' | {error, Reason}
add_nodes(SGName, Nodes) → {ok, SGName, Nodes} | {error, Reason}
remove_nodes(SGName, Nodes) → 'ok' | {error, Reason}
```

Additional Functions

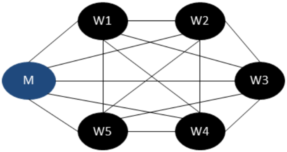
S_group information: `s_groups/0, own_nodes/0,1, own_s_groups/0, info/0`
Name registration: `register_name/3, unregister_name/2, re_register_name/3`
Searching and listing names: `registered_names/1, whereis_name/2,3`

Sending a message to a process: `send/3,4`

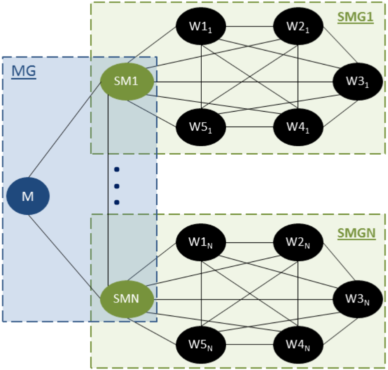
SD Erlang Improves Scalability



Distributed Erlang Orbit vs. SD Erlang Orbit



(f)



(g)

Distributed Erlang Orbit → SD Erlang Orbit

Distributed Erlang Orbit:

- `master.erl`, `worker.erl`, `table.erl`, `credit.erl`

SD Erlang Orbit:

- `master.erl`, `worker.erl`, `table.erl`, `credit.erl`
- + `submaster.erl`, `grouping.erl`

Details of the differences between the files can be checked by using, for example, `diff module1 module2` unix function

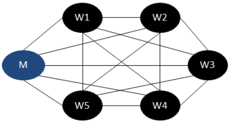
master.erl

Distributed Erlang Orbit

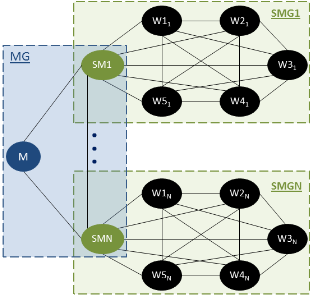
- Spawns worker processes

SD Erlang Orbit

- Spawns submaster and gateway processes



(h)



(i)

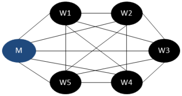
worker.erl

Distributed Erlang Orbit

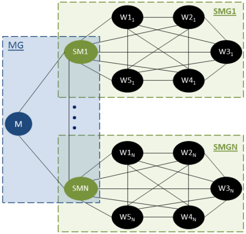
- Sends a message with vertex X directly to the target process

SD Erlang Orbit

- Sends a message with vertex X directly to the target process only if the process is in the own s_group, otherwise sends it to a gateway process



(j)



(k)

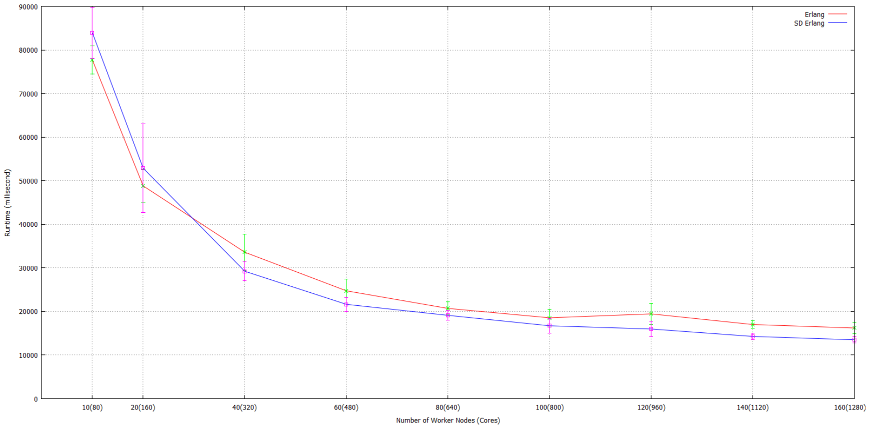
submaster.erl

- Initiates submaster and gateway processes
- Submaster processes start worker processes
- Submaster processes transfer credit from Worker processes to the Master Process
- Gateway processes receive {`Vertex, Credit`} pair and identify its corresponding s_group

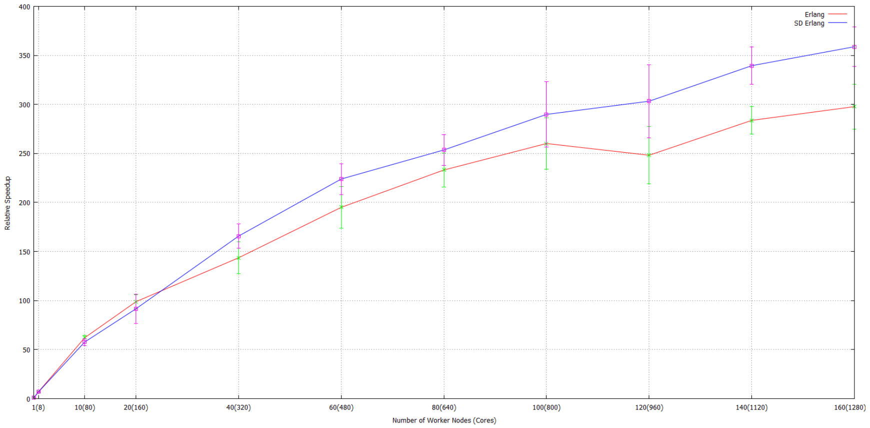
SD Erlang grouping.erl

- Creation of s_groups on Submaster nodes
- Creation of the master s_group, i.e.

Scalability of Distributed Erlang Orbit & SD Erlang Orbit



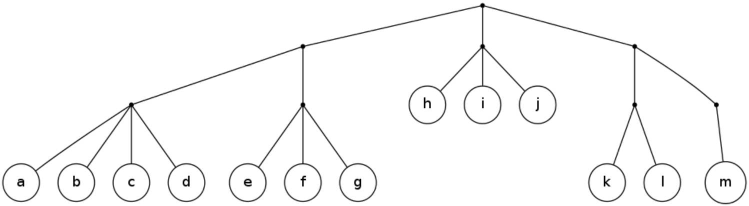
Speed Up of Distributed Erlang Orbit & SD Erlang Orbit



Semi-Explicit Placement

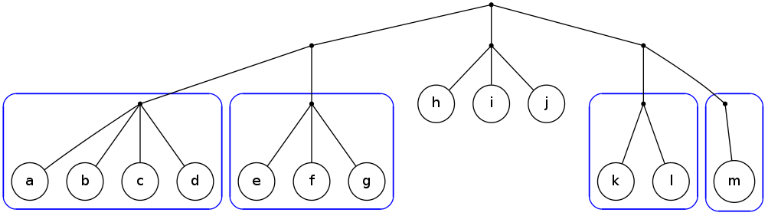
- In a distributed system, communication latencies between nodes may vary according to relative positions of the nodes in the system.
- Some nodes may be "nearby" in terms of communication time, while others may be further away (in a different cluster within a cloud, for example).
- We may wish some tasks to be close together because they're communicating with each other a lot.
- If we have a task which performs only a small amount of computation, we may wish to spawn it nearby to reduce communication overhead.
- Conversely, if we have a computationally intensive task we may wish to spawn it on a distant node which is lightly loaded.

Example



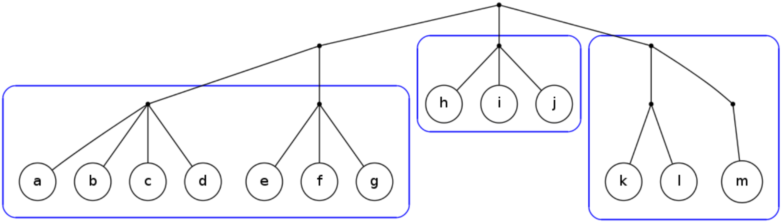
System structure

Example: system structure



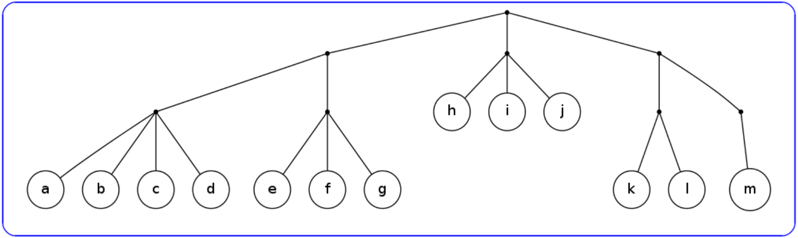
Racks

Example: system structure



Clusters

Example: system structure



Cloud

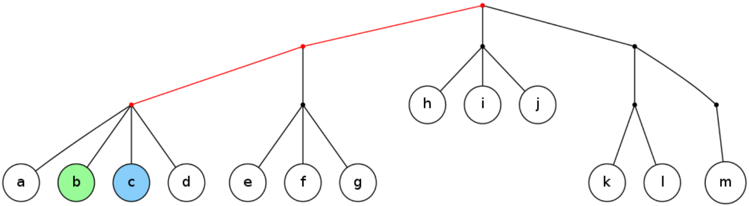
Measuring communication distance

Using an idea of Patrick Maier, Rob Stewart, and Phil Trinder, we can define a *distance function* d on the set V of Erlang VMs in a distributed system by

$$d(x,y) = \begin{cases} 0 & \text{if } x = y \\ 2^{-\ell(x,y)} & \text{if } x \neq y. \end{cases}$$

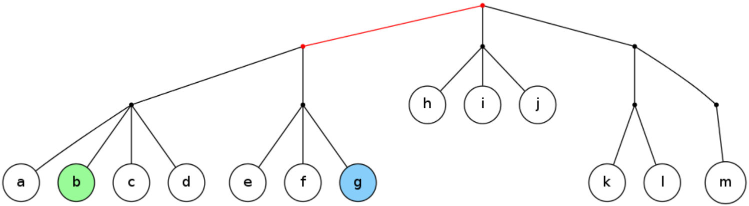
where $\ell(x,y)$ is the length of the longest path which is shared by the paths from the root to x and y.

Distances



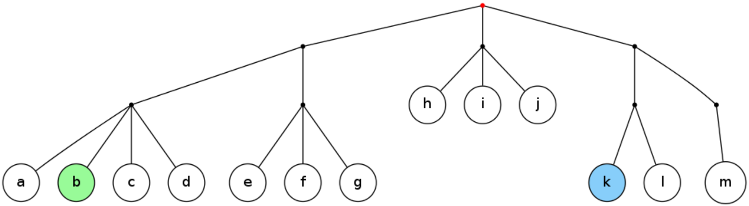
$$\ell(b, c) = 2$$
$$d(b, c) = 2^{-2} = 1/4$$

Distances



$$\ell(b, g) = 1$$
$$d(b, g) = 2^{-1} = 1/2$$

Distances



$$\ell(b,k) = 0$$
$$d(b,k) = 2^{-0} = 1$$

Measuring communication distance

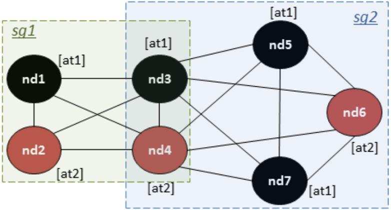
The function d has properties similar to the usual distance function in Euclidean space, and makes V into a *metric space*.
We can define the *closed disc* of radius r about a point x to be

$$D(x, r) = \{y \in V : d(x, y) \leq r\}$$

This is just the set of all nodes whose distance from x is less than or equal to r. We can use such discs to select sets of nodes within specified communication distances.

choose_nodes/1

- Every node may have a list of attributes



- choose_nodes/1 function returns a list of nodes that satisfy given restrictions

```
s_group:choose_nodes([Parameter]) -> [Node]
where
     Parameter = {s_group, SGroupName} | {attribute, AttributeName}
               | {distance, Distance}
     SGroupName = group_name()
     AttributeName = term()
```

S_group Operational Semantics

- Defined an abstract state of SD Erlang systems
- Presented the transitions of fifteen SD Erlang functions
  - Nine functions change their state after the transition: `register_name/3, re_register_name/3, unregister_name/2, whereis_name/3, send/2, new_s_group/2, delete_s_group/1, add_nodes/2, remove_nodes/2`
  - Six functions do not change the state after the transition: `send/3, whereas_name/2, registered_names/1, own_nodes/0, own_nodes/1, own_s_groups/0`

SD Erlang State

$$(grs, fgs, fhs, nds) \in \{state\} \equiv$$
$$\equiv \{(\{s\_group\}, \{free\_group\}, \{free\_hidden\_group\}, \{node\})\}$$

$$gr \in grs \equiv \{s\_group\} \equiv \{(s\_group\_name, \{node\_id\}, namespace)\}$$
$$fg \in fgs \equiv \{free\_group\} \equiv \{(\{node\_id\}, namespace)\}$$
$$fh \in fhs \equiv \{free\_hidden\_group\} \equiv \{(node\_id, namespace)\}$$
$$nd \in nds \equiv \{node\} \equiv \{(node\_id, node\_type, connections, gr\_names)\}$$

Property. Every node in an SD Erlang state is a member of one of the three classes of groups: *s_group*, *free_group*, or *free_hidden_group*. The three classes of groups partition the set of nodes.

Transitions

$$(\mathit{state}, \mathrm{command}, \mathit{ni}) \longrightarrow (\mathit{state}', \mathit{value})$$

Executing $\mathrm{command}$ on node *ni* in *state* returns *value* and transitions to state'.

register_name/3

SD Erlang function

s_group:register_name(SGroupName, Name, Pid) → yes | no

$$((grs, fgs, fhs, nds), \mathrm{register\_name}(s, n, p), ni)$$

$$\longrightarrow ((\{(s, \{ni\} \oplus nis, \{(n, p)\} \oplus ns)\} \oplus grs', fgs, fhs, nds), \mathrm{True})$$

$$\text{If } (n, \_) \notin ns \wedge (\_, p) \notin ns$$

$$\longrightarrow ((grs, fgs, fhs, nds), \mathrm{False})$$

$$\text{Otherwise}$$

where

$$\{(s, \{ni\} \oplus nis, ns)\} \oplus grs' \equiv grs$$

Validation of Semantics and Implementation

- Validate the consistency between the formal semantics and the SD Erlang implementation
- Use Erlang QuickCheck tool developed by QuviQ
- Behaviour is specified by properties expressed in a logical form
- `eqc_statem` is a finite state machine in QuickCheck

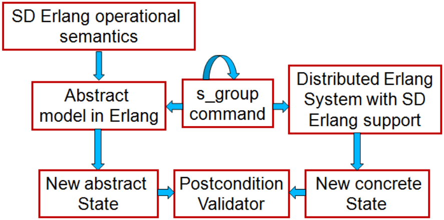


Figure: Testing SD Erlang Using QuickCheck `eqc_statem`

Precondition for `new_s_group` operation

$$
\begin{aligned}
&precondition(\_State, \{call, ?MODULE, new\_s\_group,\\
&\qquad [\{\_SGroupName, NodeIds, \_CurNode\},\\
&\qquad \_AllNodeIds]\}) \rightarrow\\
&NodeIds /= [];
\end{aligned}
$$

Postcondition for `new_s_group` operation

- **AbsRes** – abstract result; **AbsState** – abstract state
- **ActRes** – actual result; **ActState** – actual state

$$postcondition(State, \{call, ?MODULE, new\_s\_group, [\{SGroupName, NodeIds, CurNode\}, \_AllNodeIds]\}, \{ActResult, ActState\}) \rightarrow$$

$$\{AbsRes, AbsState\} = new\_s\_group\_next\_state(State, SGroupName, NodeIds, CurNode),$$

$$(AbsResult == ActResult) \text{ and } is\_the\_same(ActState, AbsState);$$

Future work

Semi-explicit Placement

- Instead of describing the system structure in a configuration file, we will look into the possibility of discovering it at runtime.
- We also want to look into questions of robustness: it would be useful to have some means of dynamically adjusting our view of the system if new nodes join it, or if existing ones fail.

Future Plans

- Continue the work on SD Erlang Semantics
- Run **Sim-Diasca** simulation engine on massively parallel supercomputer **Blue Gene/Q** with approx. 65,000 cores
- SD Erlang to become standart Erlang
- Methodology, i.e. portability princiles, scalability principles

Sources

- RELEASE Project http://www.release-project.eu/
- RELEASE github repos
  - SD Erlang https://github.com/release-project/otp/tree/dev
  - DEbench, Orbit https://github.com/release-project/benchmarks
  - Percept2 https://github.com/release-project/percept2
- BenchErl http://release.softlab.ntua.gr/bencherl/index.html
- Sim-Diasca simulation engine http://researchers.edf.com/software/sim-diasca-80704.html

Thank you!

State Components

$$
\begin{aligned}
gs &\in \{gr\_names\} \equiv \{NoGroup, \{s\_group\_name\}\} \\
ns &\in \{namespace\} \equiv \{\{(name, pid)\}\} \\
cs &\in \{connections\} \equiv \{\{node\_id\}\} \\
nt &\in \{node\_type\} \equiv \{Normal, Hidden\} \\
s &\in \{NoGroup, s\_group\_name\} \\
n &\in \{name\} \\
p &\in \{pid\} \\
ni &\in \{node\_id\} \\
nis &\in \{\{node\_id\}\} \\
m &\in \{message\}
\end{aligned}
$$

new_s_group/2

SD Erlang function

s_group:new_s_group(SGroupName, [Node]) → ok | error

$$((grs, fgs, fhs, nds), \text{new\_s\_group}(s, nis), ni)$$

$$\longrightarrow ((grs', fgs', fhs', nds''), \text{Ok}) \qquad \text{If } ni \in nis$$

$$\longrightarrow ((grs, fgs, fhs, nds), \text{Error}) \qquad \text{Otherwise}$$

where

$$nds' \equiv \text{InterConnectNodes}(nis, nds)$$

$$nds'' \equiv \text{AddSGroup}(s, nis, nds')$$

$$grs' \equiv grs \oplus \{(s, nis, \{\})\}$$

$$(fgs', fhs') \equiv \text{RemoveNodes}(nis, fgs, fhs)$$

new_s_group/2 – Auxiliary Functions (1)

$$\text{InterConnectNodes}(nis, nds)$$
$$= nds \ \cup \ \{(ni, nt, (cs \oplus nis) - \{ni\}, gs) \mid (ni, nt, cs, gs) \in nds, ni \in nis\}$$

$$\text{AddSGroup}(s, nis, nds) = nds \cup nds''$$
where
$$nds' \equiv \{(ni, nt, cs, gs) \mid (ni, nt, cs, gs) \in nds, ni \in nis\}$$
$$nds'' \equiv \{(ni, nt, cs, \text{AddSGroupS}(s, gs)) \mid (ni, nt, cs, gs) \in nds'\}$$

$$\text{AddSGroupS}(s, gs)$$
$$= \{s\} \qquad \text{If } gs \equiv NoGroup$$
$$= gs \oplus \{s\} \qquad \text{Otherwise}$$

new_s_group/2 – Auxiliary Functions (2)

$$\mathrm{RemoveNodes}(nis, fgs, fhs) = (fgs'', fhs')$$

where

$$fgs' \equiv \{(\{ni\} \oplus nis', ns') \mid (\{ni\} \oplus nis', ns') \in fgs, ni \in nis\}$$

$$fgs'' \equiv (fgs - fgs') \oplus$$

$$\oplus \{(nis', ns') \mid nis' \neq \{\},\ (\{ni\} \oplus nis', ns') \in fgs', ni \in nis\}$$

$$fhs' \equiv fhs - \{(ni, ns) \mid (ni, ns) \in fhs, ni \in nis\}$$

BashoConcepts.
Concepts, 2013.

Frank Lubeck and Max Neunhoffer.
Enumerating Large Orbits and Direct Condensation.
Experimental Mathematics, pages 197–205, 2001.

Jeff Motocha and Tracy Camp.
A taxonomy of distributed termination detection algorithms.
The Journal of Systems and Software, pages 207–221, 1998.