



## Scalable Distributed Erlang

*Natalia Chechina*  
and RELEASE Team



June 10, 2014



# Outline

- 1 RELEASE Project
- 2 Distributed Erlang
- 3 Scalable Distributed (SD) Erlang
  - Design Approach
  - Network Scalability
  - Preliminary Validation
  - Orbit
  - Semi-Explicit Placement
- 4 Operational Semantics
  - S\_group Operational Semantics
  - Validation of SD Erlang Semantics and Implementation
- 5 Future Plans

## 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).

## 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).

# Erlang

## 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).

# Erlang

- **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**

## RELEASE Aim

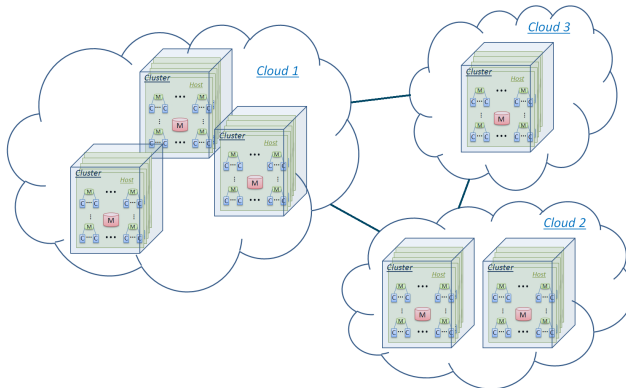
To scale the radical actor (concurrency-oriented) paradigm to build reliable general-purpose software, such as server-based systems, on massively parallel machines ( $10^5$  cores).

# Erlang

- **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**

# Typical Target Architecture - $10^5$ cores

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

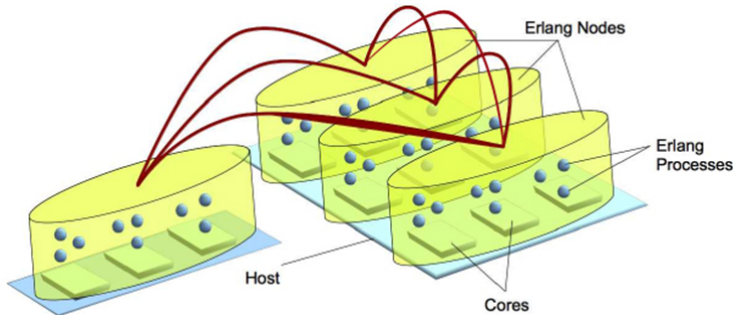


# Distributed Erlang



# Distributed Erlang

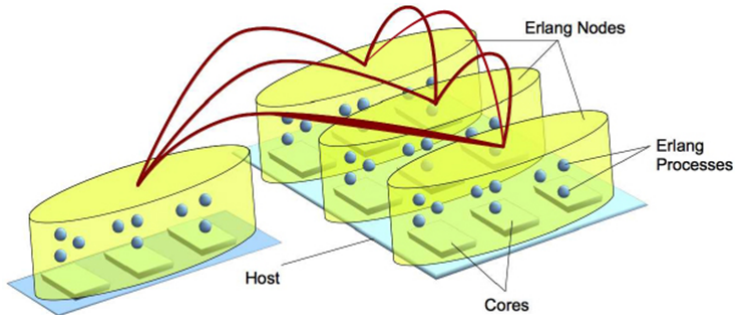
- Transitive connections



# Distributed Erlang

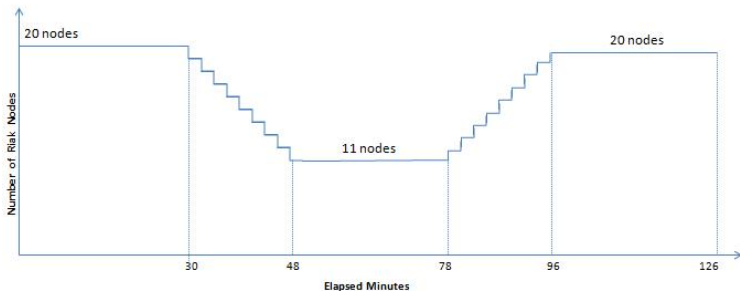
- Transitive connections
- Explicit Placement, i.e.

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

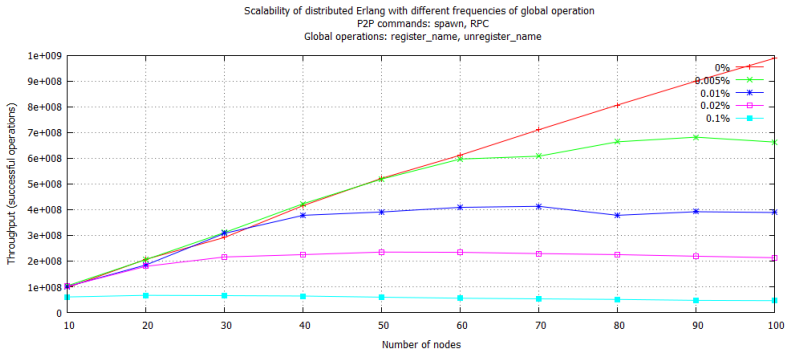


# Distributed Erlang

- **Reliability:** multiple hardware and software redundancy means that if one Host or Node fails, other Nodes can continue to deliver service

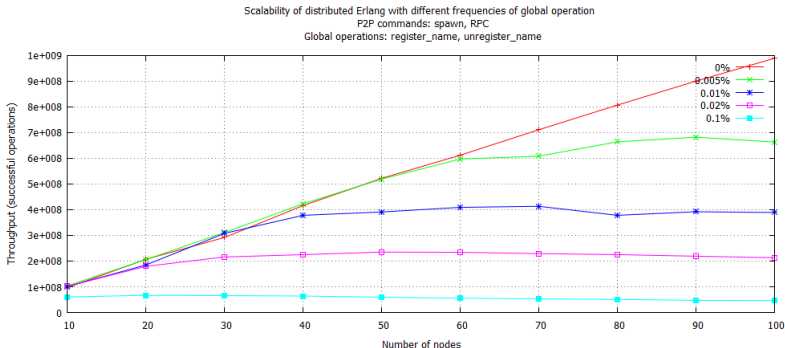


# Distributed Erlang Scalability Limitations



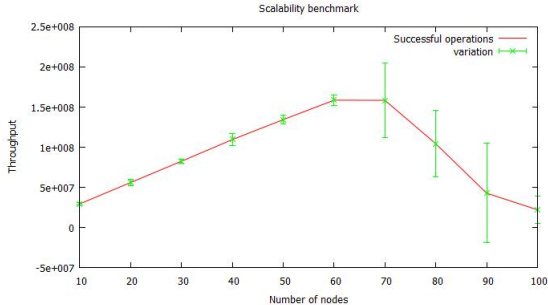
- Global operations, i.e. registering names using `global` module

# 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

# Design Approach & Principles

## Need to scale

- Persistent data structures (Riak, Casandra)
- In-memory data structures (Uppsala University, Ericsson)
- Computation

# Design Approach & Principles

## Need to scale

- Persistent data structures (Riak, Casandra)
- In-memory data structures (Uppsala University, Ericsson)
- **Computation**

## SD Erlang design principles

- Working at Erlang level as far as possible
- Preserving the Erlang philosophy and programming idioms
- Keeping Erlang reliability model unchanged as far as possible



# 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 and shared name space

- **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

## Grouping nodes into s\_groups

### Types of nodes

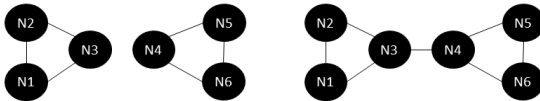
- Free nodes (normal or hidden) belong to *no s\_group*
- S\_group nodes belong to *at least one s\_group*

Nodes in an s\_group have **transitive** connections only with nodes from the same s\_groups, but **non-transitive** connections with other nodes

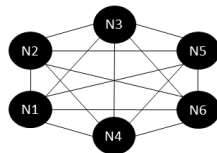
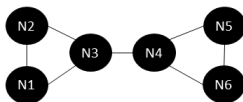
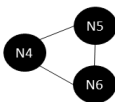
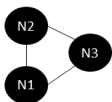
# Free Node Connections vs. S\_group Node Connections



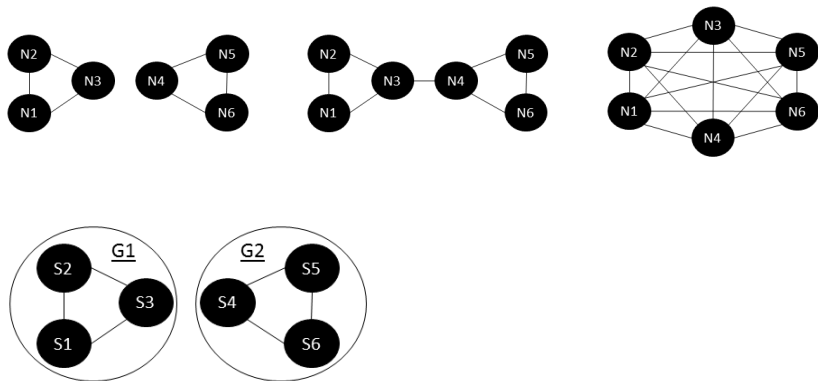
# Free Node Connections vs. S\_group Node Connections



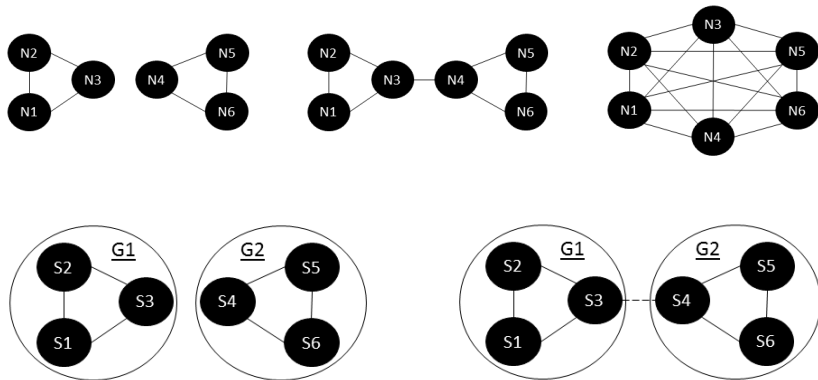
# Free Node Connections vs. S\_group Node Connections



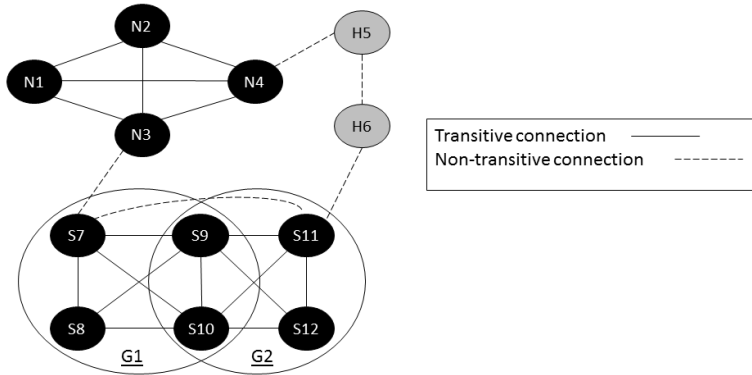
# Free Node Connections vs. S\_group Node Connections



# Free Node Connections vs. S\_group Node Connections

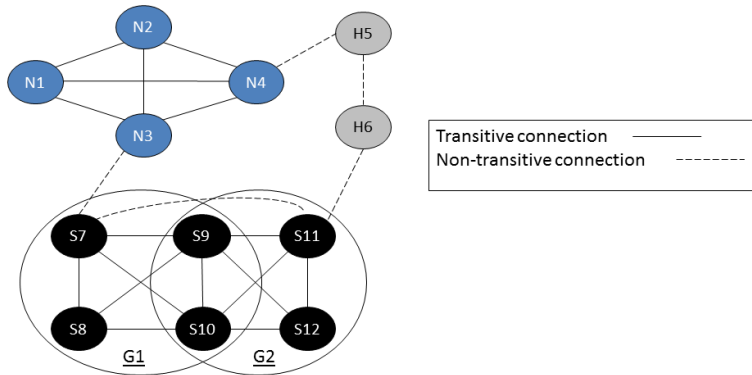


# Connections between Different Types of Nodes

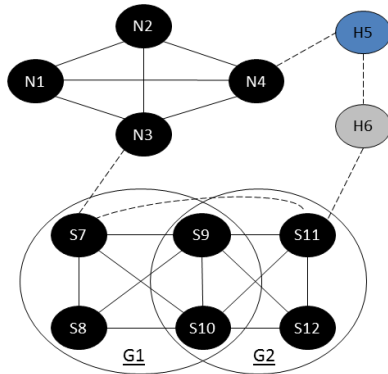




# Connections between Different Types of Nodes

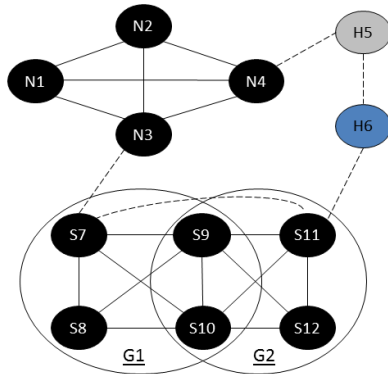


# Connections between Different Types of Nodes



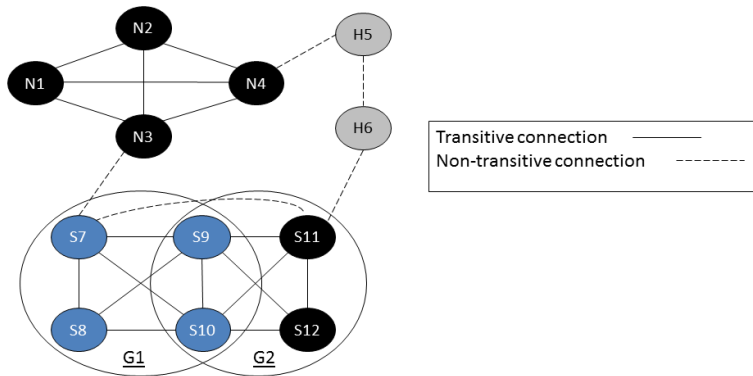
Transitive connection ———  
Non-transitive connection - - - - -

# Connections between Different Types of Nodes

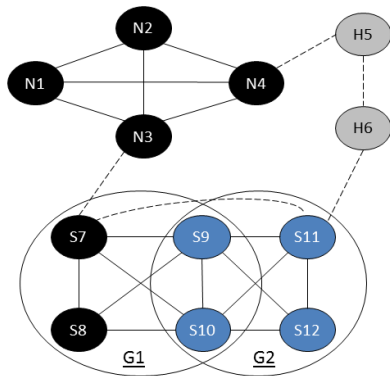


Transitive connection ———  
Non-transitive connection - - - - -

# Connections between Different Types of Nodes



# Connections between Different Types of Nodes



Transitive connection ———  
Non-transitive connection - - - - -

# Why *s\_groups*?

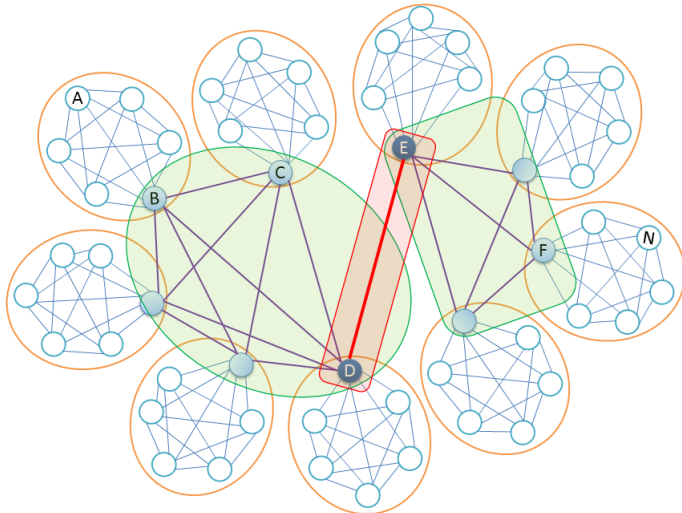
## Requirements to the node grouping approach

- 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

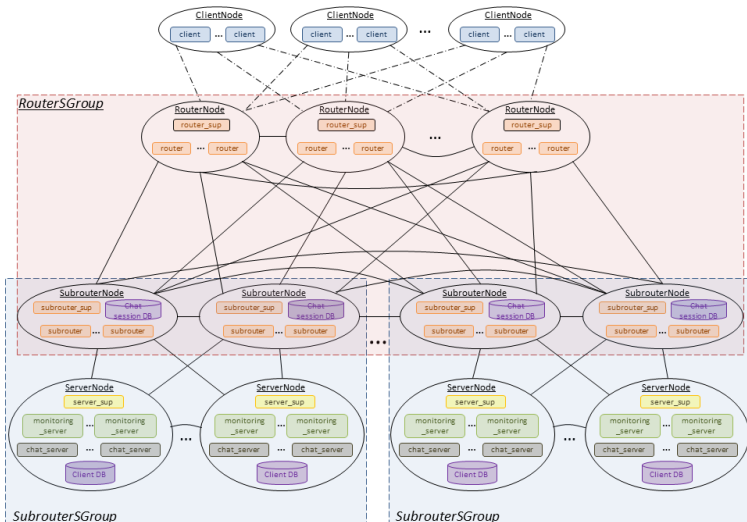
## A list of considered approaches

- Grouping nodes according to their hash values
- A hierarchical approach
- *Overlapping s\_groups*

# Hierarchical Grouping

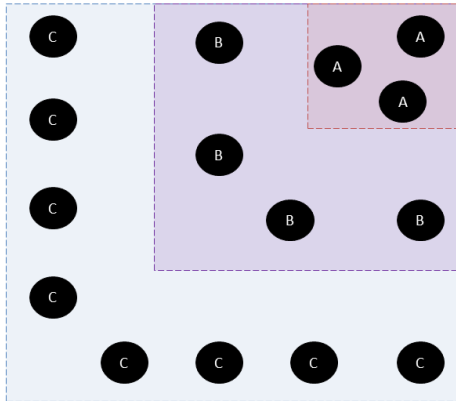


# Free Nodes and S\_groups





# Embedded Grouping



# 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

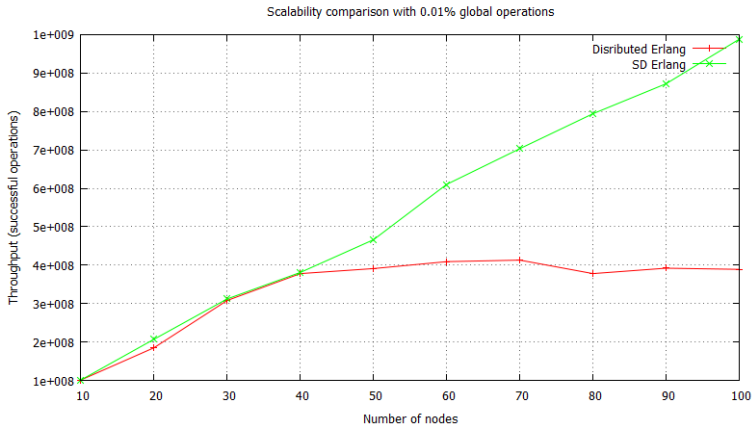
## Main Functions

```
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



## Orbit Example

- Orbit is a symbolic computing kernel and a generalization of a transitive closure computation [LN01]
- To compute Orbit for a given space  $[0..X]$  we apply on the initial vertex  $x_0 \in [0..X]$  a list of generators  $g_1, g_2, \dots, g_n$  that creates new numbers  $(x_1 \dots x_n) \in [0..X]$ . The generator functions are applied on the new numbers until no new number is generated.

## Orbit Example

- Orbit is a symbolic computing kernel and a generalization of a transitive closure computation [LN01]
- To compute Orbit for a given space  $[0..X]$  we apply on the initial vertex  $x_0 \in [0..X]$  a list of generators  $g_1, g_2, \dots, g_n$  that creates new numbers  $(x_1 \dots x_n) \in [0..X]$ . The generator functions are applied on the new numbers until no new number is generated.

## Orbit Example

- Orbit is a symbolic computing kernel and a generalization of a transitive closure computation [LN01]
- To compute Orbit for a given space  $[0..X]$  we apply on the initial vertex  $x_0 \in [0..X]$  a list of generators  $g_1, g_2, \dots, g_n$  that creates new numbers  $(x_1 \dots x_n) \in [0..X]$ . The generator functions are applied on the new numbers until no new number is generated.

## Orbit Example

- Orbit is a symbolic computing kernel and a generalization of a transitive closure computation [LN01]
- To compute Orbit for a given space  $[0..X]$  we apply on the initial vertex  $x_0 \in [0..X]$  a list of generators  $g_1, g_2, \dots, g_n$  that creates new numbers  $(x_1 \dots x_n) \in [0..X]$ . The generator functions are applied on the new numbers until no new number is generated.

## Orbit Example

- Orbit is a symbolic computing kernel and a generalization of a transitive closure computation [LN01]
- To compute Orbit for a given space  $[0..X]$  we apply on the initial vertex  $x_0 \in [0..X]$  a list of generators  $g_1, g_2, \dots, g_n$  that creates new numbers  $(x_1 \dots x_n) \in [0..X]$ . The generator functions are applied on the new numbers until no new number is generated.



## Orbit Example

- Orbit is a symbolic computing kernel and a generalization of a transitive closure computation [LN01]
- To compute Orbit for a given space  $[0..X]$  we apply on the initial vertex  $x_0 \in [0..X]$  a list of generators  $g_1, g_2, \dots, g_n$  that creates new numbers  $(x_1 \dots x_n) \in [0..X]$ . The generator functions are applied on the new numbers until **no new number** is generated.

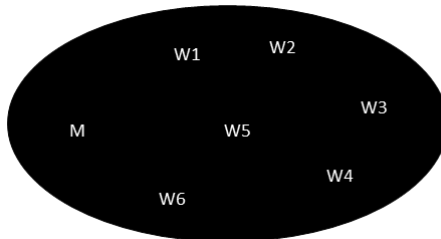
## Why Orbit?

- 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 Non-distributed Erlang

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

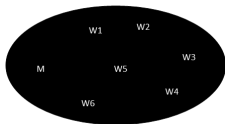
```
Pid = spawn_link(worker, init, [TabSize, TmOut, SpawnImgComp])
```



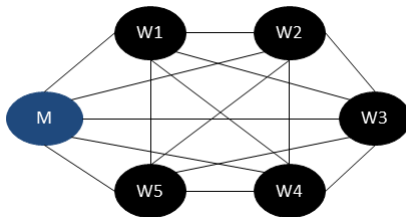
# Orbit in Distributed Erlang

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])`

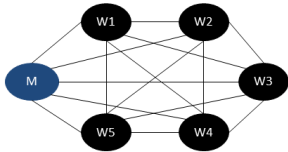


(a)

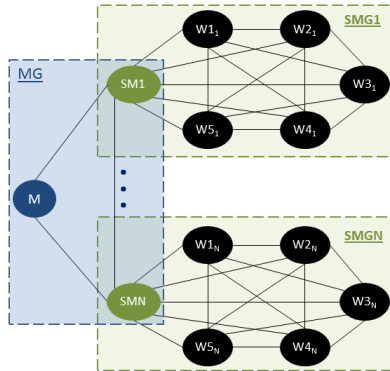


(b)

# Distributed Erlang Orbit vs. SD Erlang Orbit



(c)



(d)

## Distributed Erlang Orbit → SD Erlang Orbit

### Distributed Erlang Orbit:

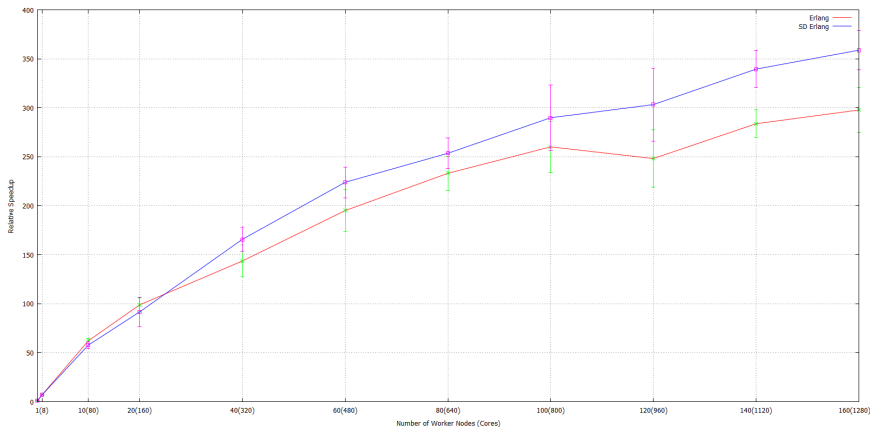
- `master.erl`, `worker.erl`, `table.erl`, `credit.erl`

### SD Erlang Orbit:

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

Kent team works on refactoring mechanisms

# Speed Up of Distributed Erlang Orbit & SD Erlang Orbit

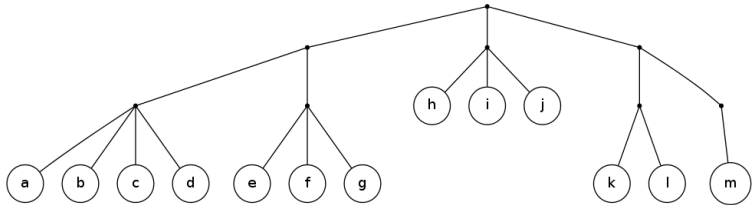


## Semi-Explicit Placement

- 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. computation, we may wish to spawn it nearby to reduce communication overhead. wish to spawn it on a distant node which is lightly loaded.

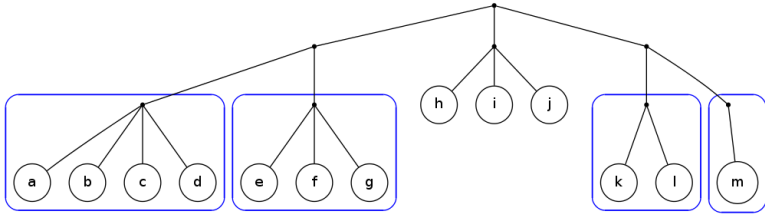


# Example



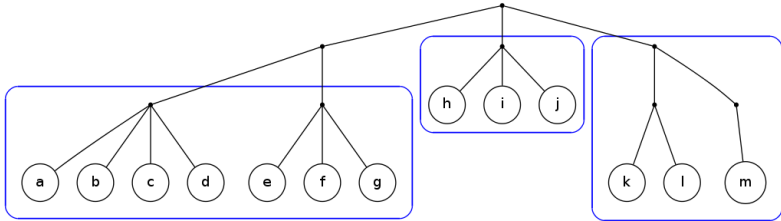
System structure

## Example: system structure



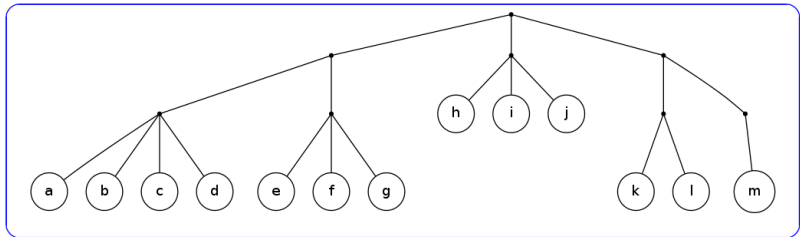
Racks

## Example: system structure



Clusters

## Example: system structure



Cloud

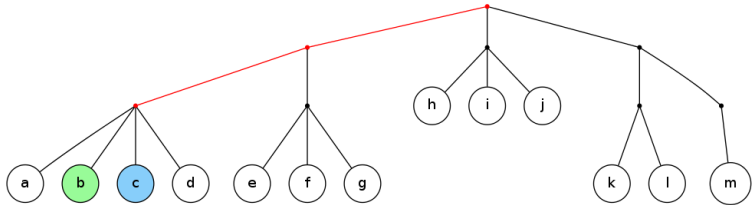
## Measuring communication distance

We can define a **distance function**  $d$  on the set  $V$  of Erlang VMs in a distributed system by

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

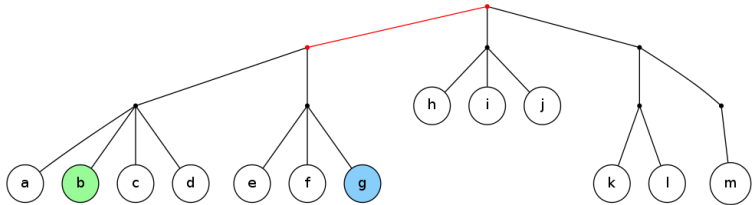
where  $\ell(x, y)$  is the length of the **longest path** which is **shared** by the paths **from the root** to  $x$  and  $y$ .

# Distances



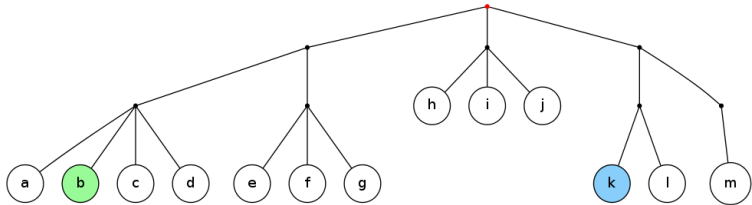
$$\ell(b, c) = 2$$
$$d(b, c) = 2^{-2} = 1/4$$

# Distances



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

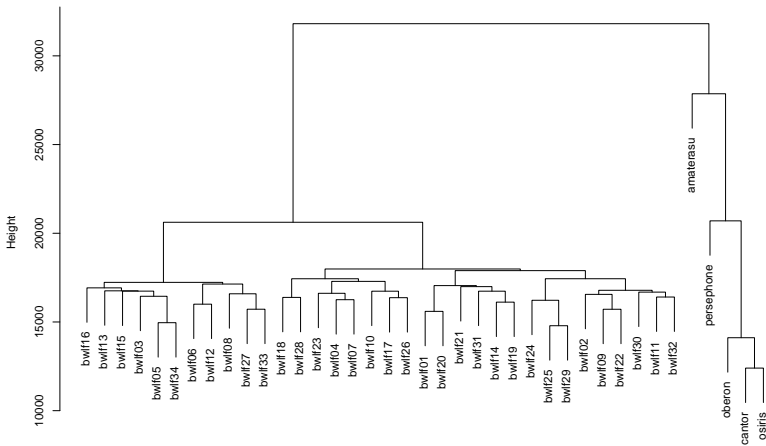
# Distances



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

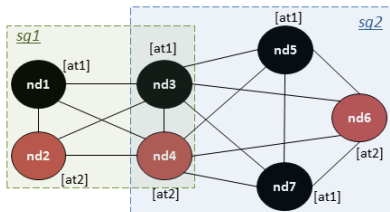


# Dendrogram



## 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}
               | {nearer, 0.4} | {between, 0.5, 0.7}
    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

$$(state, command, ni) \longrightarrow (state', value)$$

Executing *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`

$((grs, fgs, fhs, nds), register\_name(s, n, p), ni)$   
→  $((\{(s, \{ni\} \oplus nis, \{(n, p)\} \oplus ns)\} \oplus grs', fgs, fhs, nds), True)$   
If  $(n, -) \notin ns \wedge (-, p) \notin ns$   
→  $((grs, fgs, fhs, nds), False)$   
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_state` is a finite state machine in QuickCheck

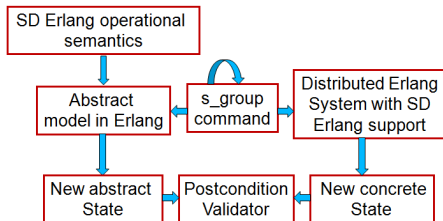


Figure: Testing SD Erlang Using QuickCheck `eqc_state`

## Precondition for `new_s_group` operation

*precondition*(*\_State*, {*call*, ?*MODULE*, *new\_s\_group*,  
[{*\_SGroupName*, *Nodelds*, *\_CurNode*},  
*\_AllNodelds*]}) →

*Nodelds*/ = [];



## 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*, *Nodelds*, *CurNode*},  
                  \_*AllNodelds*}},  
                  {*ActResult*, *ActState*}) →

{*AbsRes*, *AbsState*} =  
= *new\_s\_group\_next\_state*(*State*, *SGroupName*, *Nodelds*, *CurNode*),  
(*AbsResult* == *ActResult*) and *is\_the\_same*(*ActState*, *AbsState*);

## Future work

### Semi-explicit Placement

- For reasons of portability and understandability it might not be desirable to expose too much information about distances to programmers. We may wish to implement a more **abstract interface**, using attributes along the lines of very close, close, medium, distant, very distant.
- We will look into the possibility of **discovering the system structure at runtime**, instead of describing it in a configuration file.
- We also want to look into questions of **reliability** – to have some means of dynamically adjusting our view of the system if new nodes join it, or if existing ones fail.




## Future Plans

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

## 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!

-  [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.