Scaling Distributes Systems

Natalia Chechina
and RELEASE Team

June 11, 2015

Who am I?

- 2011: Received PhD degree in Computer Science from Heriot-Watt University, UK
- 2011-2015: WP3 lead in the EU RELEASE Project at Glasgow University, UK
- March 2015: Research Fellow at Glasgow University, UK

Main research interest: Scaling distributed computations on commodity hardware

Sources

- Research findings
- Experience from the RELEASE project
  - Funded by EU FP7 Framework
  - 5 academic & 3 industrial partners
  - 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 programming language
- Experience of other researches and developers

Scaling a Sysem

Scaling ALL aspects of computation

- Application
- Language
- Virtual Machine
- In-memory data structures
- Persistent data structures
- Tools (debugging, monitoring, etc)

Scaling a Sysem

Scaling ALL aspects of computation

- Application
- Language
- Virtual Machine
- In-memory data structures
- Persistent data structures
- Tools (debugging, monitoring, etc)

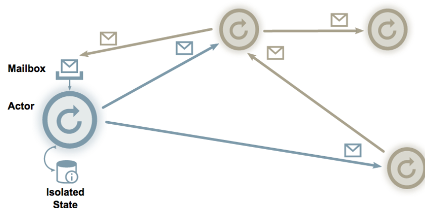
Scaling on language level

- Actor model
- Functional programming

Language – Actor Model

Built-in concurrency

- Actors have own states and don't share them
- Communication between actor happens only via message passing
- Actors can spawn new actors



Language – Functional programming

Fundamental operation – application of functions to arguments

- Higher-order functions – well-structured software
- Modules – independent, reusable
- Lazy evaluations
- Variables given values only once

Fault Tolerance

- 10^5 cores – approx. failure of 1 core per hour
- Non-defensive approach – Supervision & "Let it crash"

Philosophy

- Principles
- Ideas
- Core values

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

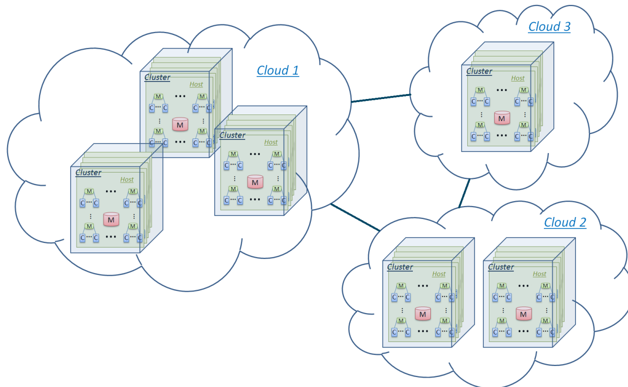
To scale the radical actor (concurrency-oriented) paradigm to build reliable general-purpose software, such as server-based systems, on massively parallel machines (10^5 cores).

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)



Erlang Overview

Erlang

- is a functional general purpose concurrent programming language developed in 1986 at Ericsson
- is dynamically typed
- was designed for distributed, fault-tolerant, massively concurrent, and soft-real time systems
- follows *let it crash* and *share nothing* philosophy

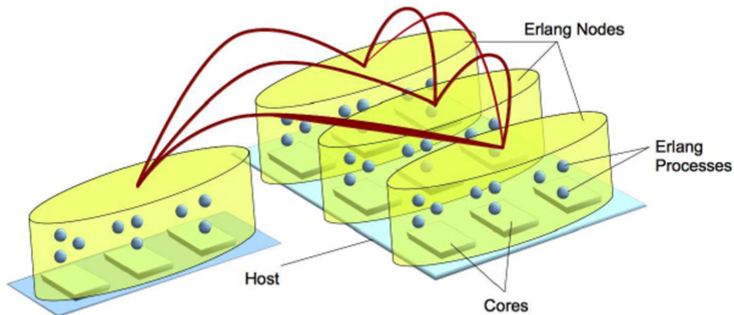
The language primitives are processes.
Erlang concurrency is handled by the language and not by the operating system [Arm10].

Distributed Erlang

Distributed Erlang

Distributed Erlang

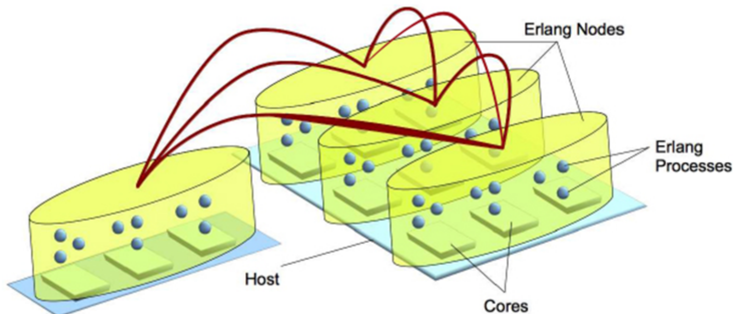
- Transitive connections



Distributed Erlang

- Transitive connections
- Explicit Placement, i.e.

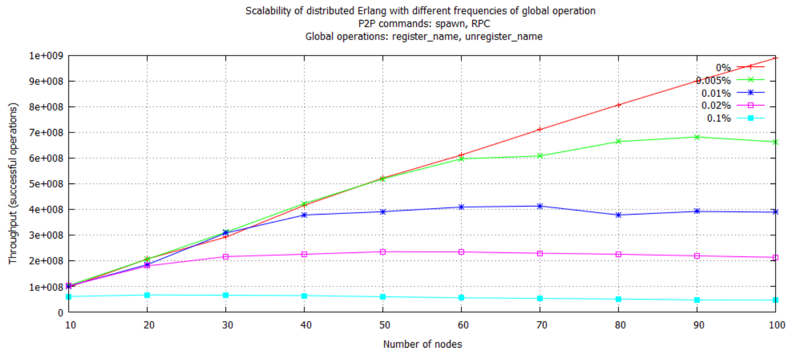
```
spawn(Node, Module, Function, Args) → pid()
```



Distributed Erlang Scalability Limitations

Global operations

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



Distributed Erlang Scalability Limitations

Global operations

- 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

Global operations

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

All-to-all transitive connections

Distributed Erlang Scalability Limitations

Global operations

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

All-to-all transitive connections

But... aren't global operations and transitivity are optional in distributed Erlang? Why use them if they are a bottleneck?

Distributed Erlang Scalability Limitations

Global operations

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

All-to-all transitive connections

But... aren't global operations and transitivity are optional in distributed Erlang? Why use them if they are a bottleneck?

- *Reliability and fault tolerance* – when a process or a node fail, the remaining nodes know about that. The same holds for the recovery
- *It's already there* – no extra effort to connect nodes and distribute information
- *Easy to scale* – a new node knows about running nodes, and vice versa

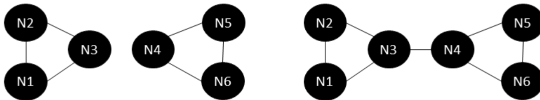
Scalable Distributed (SD) Erlang

SD Erlang is a small conservative extension of Distributed Erlang

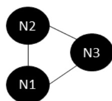
- **Network Scalability**
  - All-to-all connections are not scalable onto 1000s of nodes
  - *Aim:* Reduce connectivity
- **Semi-explicit Placement**
  - Becomes not feasible for a programmer to be aware of all nodes
  - *Aim:* Automatic process placement in groups of nodes

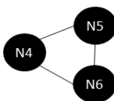
Free Node Connections vs. S_group Node Connections

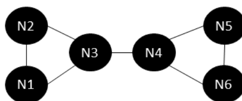
Free Node Connections vs. S_group Node Connections

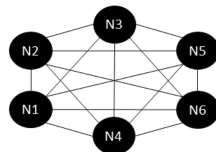


Free Node Connections vs. S_group Node Connections

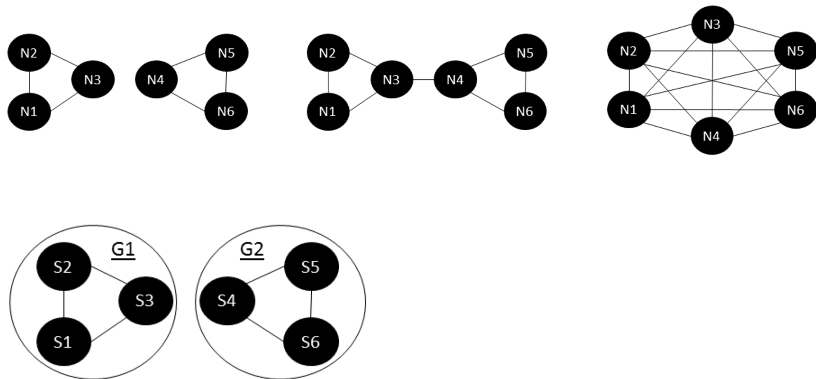




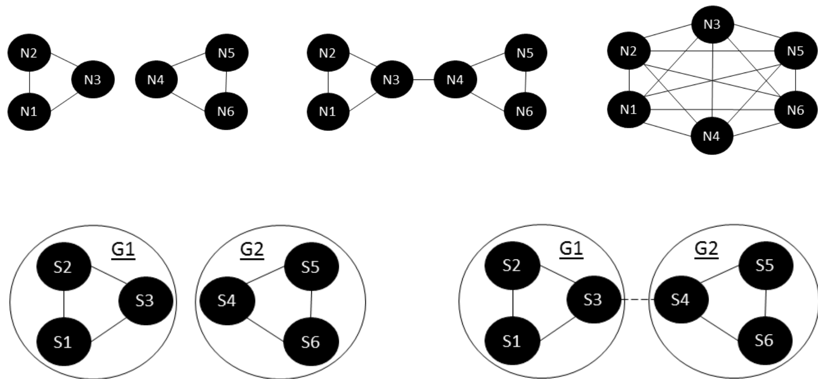




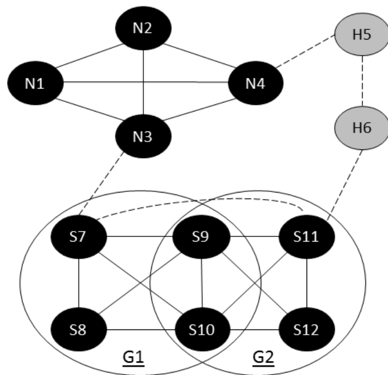
Free Node Connections vs. S_group Node Connections



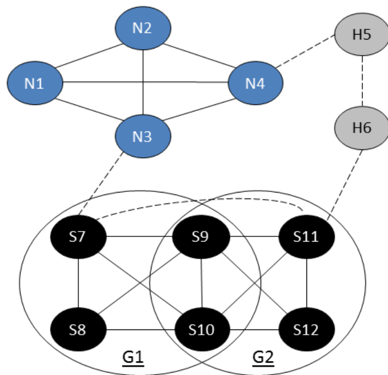
Free Node Connections vs. S_group Node Connections



Connections between Different Types of Nodes

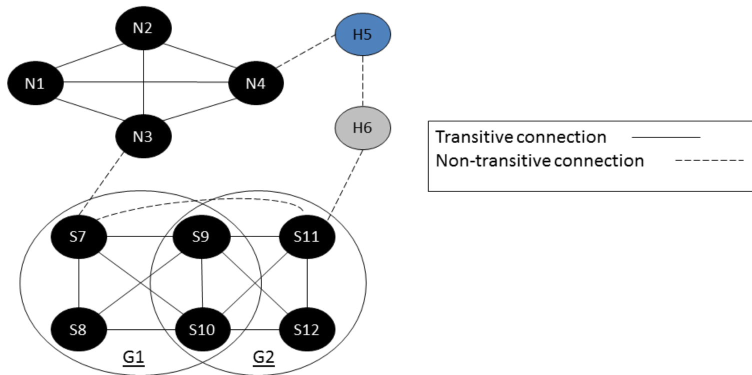


Connections between Different Types of Nodes

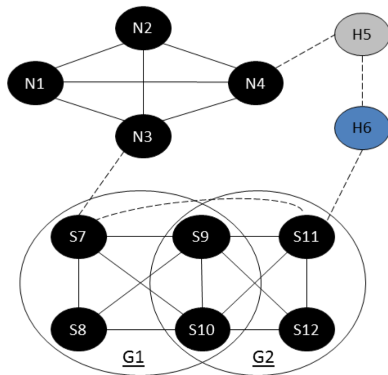


Transitive connection	————
Non-transitive connection	--------

Connections between Different Types of Nodes

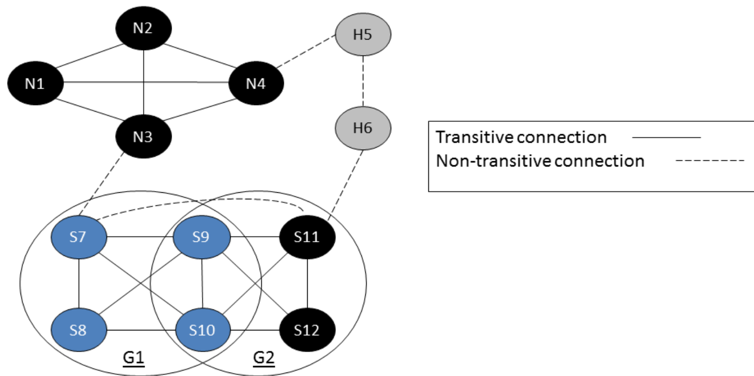


Connections between Different Types of Nodes

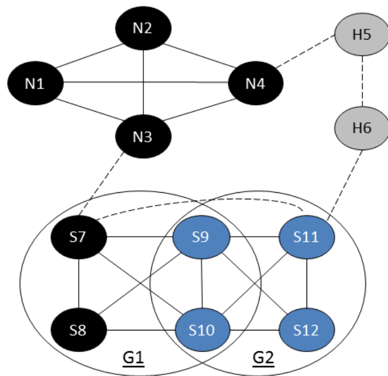


Transitive connection ————
Non-transitive connection - - - - - - - -

Connections between Different Types of Nodes



Connections between Different Types of Nodes



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

Why S_groups?

Preserve Erlang phylosophy & transitivity and scale

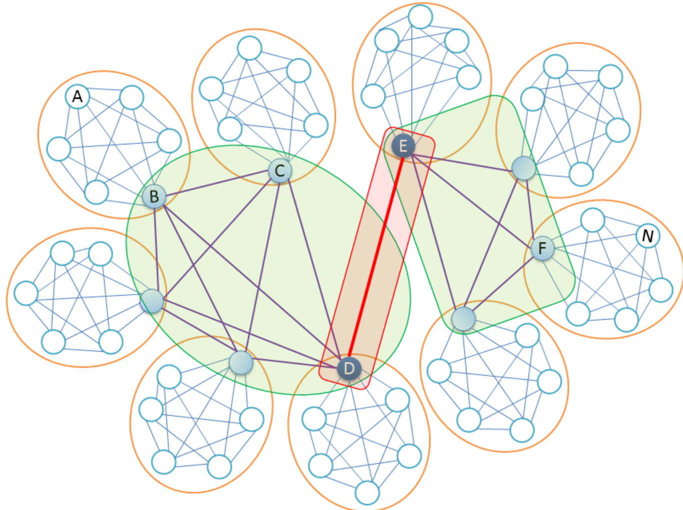
Considered approaches

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

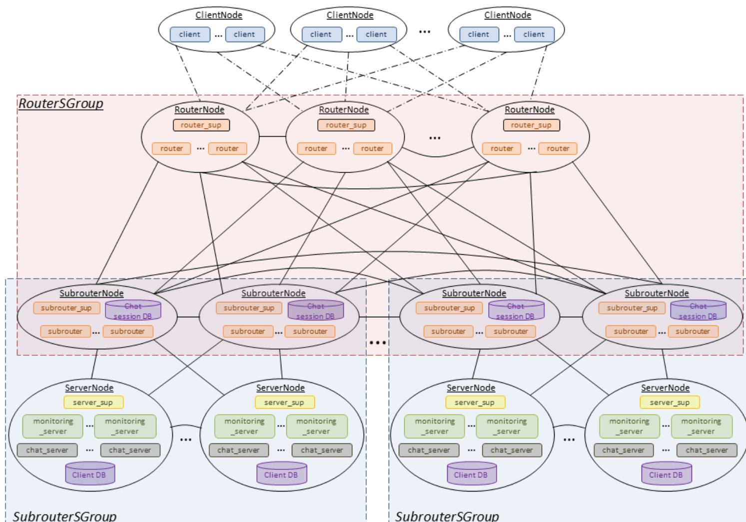
Other approaches

- Distributed Erlang global_groups
- Spapi Router (SpilGames)
- Custom routing on non-transtively connected normal or hidden nodes

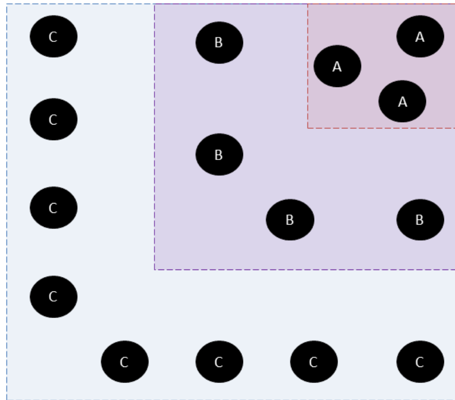
Hierarchical Grouping



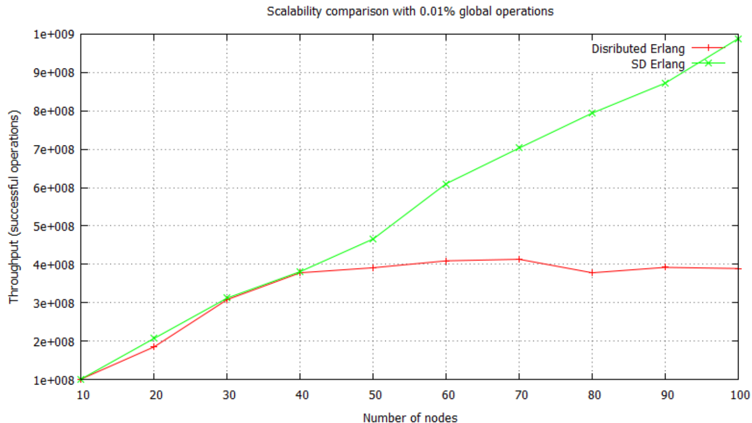
Free Nodes and S_groups



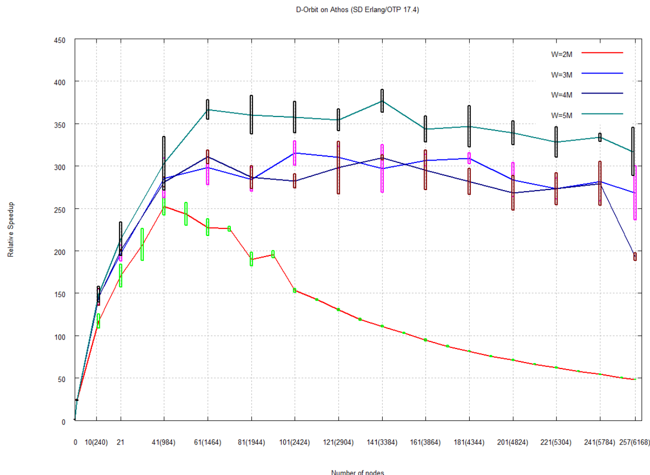
Embedded Grouping



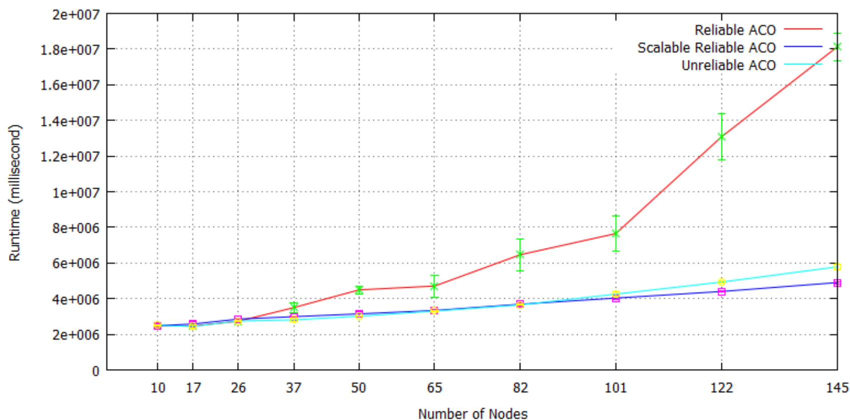
SD Erlang Improves Scalability



Speed Up of Distributed Erlang Orbit & SD Erlang Orbit

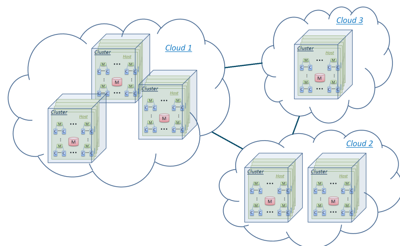


Speed Up of Distributed Erlang ACO & SD Erlang ACO

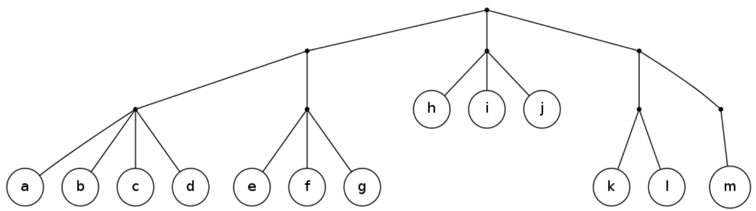


Semi-Explicit Placement

- Communication latencies between nodes may vary according to their relative positions
- In terms of communication time nodes may be "nearby" or "far away"
- We may wish some tasks to be close together because they're communicating with each other a lot

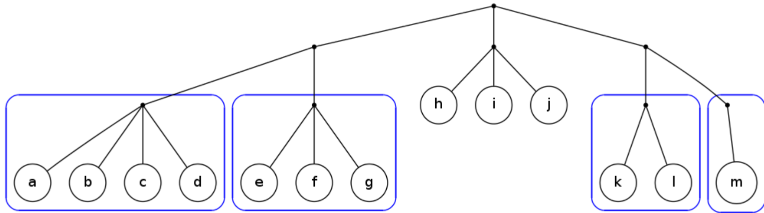


Example



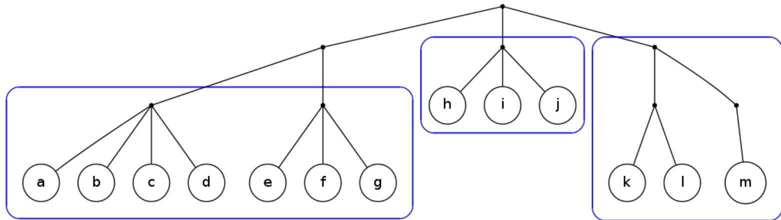
System structure

Example: system structure



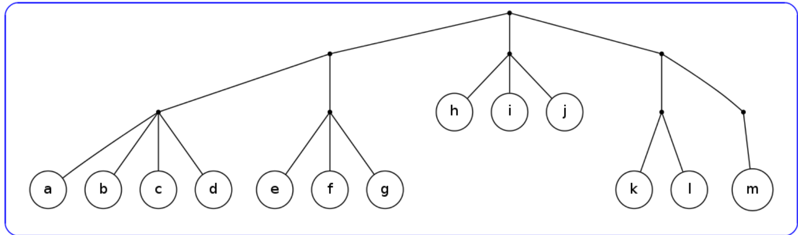
Racks

Example: system structure



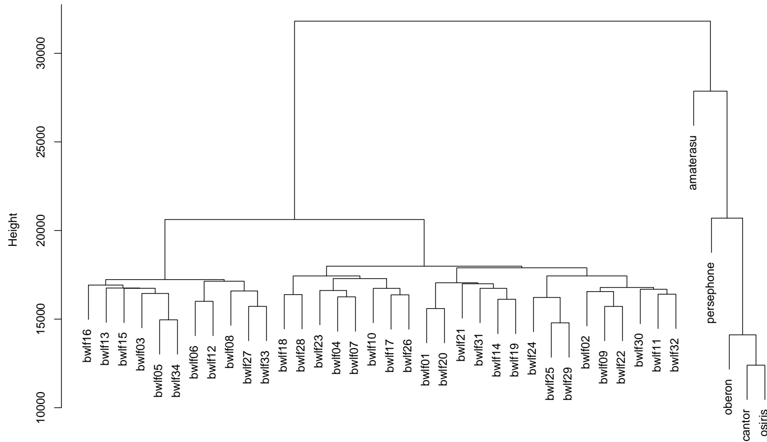
Clusters

Example: system structure



Cloud

Dendrogram



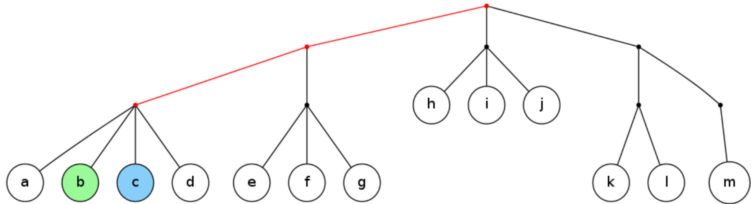
Measuring communication distance

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

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

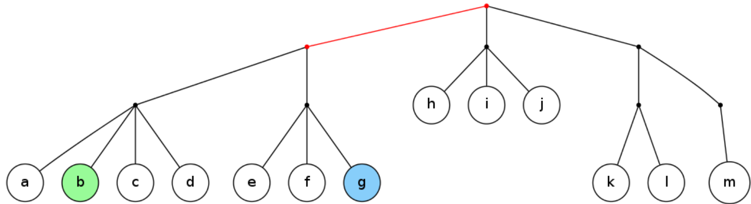
where $\ell(x,y)$ is the length of the longest path which is shared by the paths from the root to x and y.

Distances



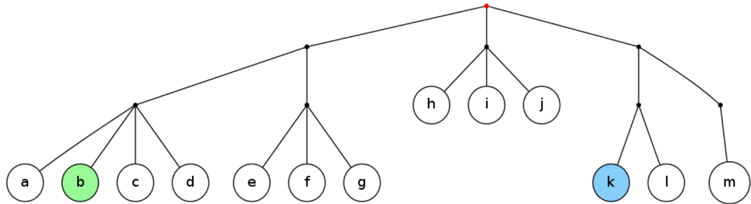
$$\ell(b, c) = 2$$
$$d(b, c) = 2^{-2} = 1/4$$

Distances



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

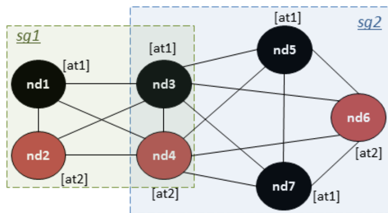
Distances



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

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()
```

Operational Semantics

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

Executing command on node *ni* in *state* returns *value* and transitions to *state*′.

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

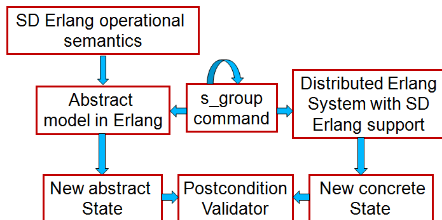


Figure: Testing SD Erlang Using QuickCheck `eqc_statem`

Ongoing and Future Work

S_groups

- Introduce *more patterns*, for example, routing for a tree structure
- Analysis of fault tolerance strategies and features in SD Erlang applications

Semi-explicit Placement

- *Discovering system structure at runtime*
- *Robustness* – dynamically adjusting a view of the system if new nodes join it, or if existing ones fail

Sources

- SD Erlang http://www.dcs.gla.ac.uk/research/sd-erlang/
- RELEASE Project http://www.release-project.eu/

Deployment tool

- Wombat https://www.erlang-solutions.com/products/wombat

Profiling tools

- Percept2 https://github.com/release-project/percept2
- devo https://www.youtube.com/watch?v=Ox30TBDcFPw

Benchmarking

- BenchErl http://release.softlab.ntua.gr/bencherl/index.html
- DEbench, Orbit, ACO https://github.com/release-project/benchmarks

Thank you!

J. Armstrong.
Erlang.
Commun. ACM, 53:68–75, 2010.