Improving the Network Scalability of Erlang

Natalia Chechina[1,a], Huiqing Li[b], Amir Ghaffari[a], Simon Thompson[b], Phil Trinder[a]

[a] *School of Computing Science, The University of Glasgow, Glasgow, UK, G12 8QQ*
[b] *School of Computing, University of Kent, Canterbury, UK, CT2 7NF*

Abstract

As the number of cores grows in commodity architectures so does the likelihood of failures. A distributed actor model potentially facilitates the development of reliable and scalable software on these architectures. Key components include lightweight processes which 'share nothing' and hence can fail independently. Erlang is not only increasingly widely used, but the underlying actor model has been a beacon for programming language design, influencing for example Scala, Clojure and Cloud Haskell.

While the Erlang distributed actor model is inherently scalable, we demonstrate that it is limited by some pragmatic factors. We address two network scalability issues here: globally registered process names must be updated on every node (virtual machine) in the system, and any Erlang nodes that communicate maintain an active connection. That is, there is a fully connected $O(n^2)$ network of n nodes.

We present the design, implementation, and initial evaluation of a conservative extension of Erlang – Scalable Distributed (SD) Erlang. SD Erlang partitions the global namespace and connection network using s_groups. An s_group is a set of nodes with its own process namespace and with a fully connected network within the s_group, but only individual connections outside it. As a node may belong to more than one s_group it is possible to construct arbitrary connection topologies like trees or rings.

*Corresponding author. URL: `http://www.release-project.eu/`, Fax: +44 (141) 330-4913.

Email addresses: `Natalia.Chechina@glasgow.ac.uk` (Natalia Chechina), `H.Li@kent.ac.uk` (Huiqing Li), `Amir.Ghaffari@glasgow.ac.uk` (Amir Ghaffari), `S.J.Thompson@kent.ac.uk` (Simon Thompson), `Phil.Trinder@glasgow.ac.uk` (Phil Trinder)

We present an operational semantics for the s_group functions, and outline the validation of conformance between the implementation and the semantics using the QuickCheck automatic testing tool. Our preliminary evaluation in comparison with distributed Erlang shows that SD Erlang dramatically improves network scalability even if the number of global operations is tiny (0.01%). Moreover, even in the absence of global operations the reduced connection maintenance overheads mean that SD Erlang scales better beyond 80 nodes (1920 cores).

Keywords: distributed system, Erlang, actor model, operational semantics, validation, conformance, QuickCheck, testing

---

1. Introduction

Erlang (1) is a distributed actor-based functional programming language. The actor model dates from 1973 (2). In the model a system is represented by community of actors, where actors are independent and interactive entities whose interactions are defined by asynchronous message passing. The model is inherently concurrent due to actors being *self-contained* – every actor has a state that is not shared with other actors, and hence may fail independently from each other. Some of the early actor-based languages are as follows: E (3), Erlang (4), and Smalltalk (5). An overview of, and discussion on, the first actor languages can be found in (6).

The Erlang concurrency model is based on *share nothing* message passing between independent actors or processes. Because of this, it can support parallelism and reliability directly. The uptake of Erlang is increasing around the world and broadening from its telecom base into other sectors including finance, database, messaging, and embedded systems. Erlang concurrency differs from many other programming languages in that it is handled by the language and not by the 'host' operating system (7). The concurrency is based on the light-weight processes that are easy to create and destroy, and inexpensive message passing between processes. The language follows the functional paradigm in that variables are single-assignment: once a value is assigned to an instance of a variable it cannot subsequently be changed. Distributed actor programming is distinctive as it is based on highly-scalable lightweight processes that share nothing. Erlang/OTP provides high-level coordination with concurrency and robustness built-in: it can readily support 10,000 processes per core, with transparent distribution of processes across

multiple machines, using message passing for communication. Moreover, the robustness of the Erlang distribution model is provided by hierarchies of supervision processes which manage recovery from software or hardware errors.

Currently, Erlang/OTP (8) has inherently scalable computation and support for building reliable systems, but in practice network scalability is constrained by the default model of full connectivity between all distributed Erlang Virtual Machines (VMs, also called *nodes*) in a system. This limits network scalability as the system must maintain live network connections quadratic in the number of nodes.

In the RELEASE project (9) we aim to scale Erlang's radical distributed actor programming paradigm to build reliable general-purpose software, such as server-based systems, on massively parallel machines. We target reliable scalable general purpose heterogeneous platforms. Our application area is that of general server-side computation, e.g. web or messaging servers. This form of computation is ubiquitous, in contrast to more specialised forms such as traditional high-performance computing. Moreover, we target computation on stock platforms, with standard hardware, operating systems and middleware, rather than on more specialised software stacks on specific hardware, e.g. highly reliable HPC hardware.

To extend the distributed actor paradigm to large-scale reliable parallelism we have designed and implemented a conservative extension to the Erlang language, Scalable Distributed (SD) Erlang, for reliable network scalability. The paper is the first published description of SD Erlang s_groups, and makes the following research contributions.

1. We demonstrate the network scalability limitations of distributed Erlang (Section 2).
2. We define and implement s_groups to reduce network connectivity by dividing large sets of distributed nodes in reliable actor systems (Section 3).
3. We provide an operational semantics for s_groups and validate the implementation against it using QuickCheck (10) (Sections 4 and 5).
4. We demonstrate that SD Erlang provides improved network scalability on up to 257 distributed nodes (6168 cores) (Section 6).

2. Distributed Erlang & Scalability Limitations

2.1. Distributed Erlang

Distributed Erlang was introduced to enable Erlang nodes placed on the same or different physical machines to work together. By default the system aims to maintain a fully connected network of nodes by means of *transitivity*, i.e. when node $N1$ connects to node $N2$ it will also automatically connect to all nodes $N2$ is connected to, and visa versa.

It is possible to override the default distributed Erlang connection transitivity and namespace management policies, for example, by using `-connect_all false` flag. Many applications need to do so to enhance performance when scaling, e.g. Spapi-router (11), Megaload (12). However, transitive connections and shared namespace are an important feature of distributed Erlang because they support fault tolerance and elasticity. We first explain the type of *fault tolerance* we mean here. If a process (let us call it a master process) is globally registered then other processes that want to send it a message do not need to know its pid, only the name. In case the master process fails it will be immediately unregistered, so all nodes will be notified of the failure. When the master process is re-started and re-registered using the same name, its pid changes but other processes can continue to use its name to send messages. So, the frequency of global name registration usually depends on the frequency of the failure of globally registered processes. By *elasticity* we mean an effortless scaling (from a programmer's point of view) of the number of nodes up and down. That is, if a node fails, all connected nodes are notified of the failure. When a new node is added to the system it gets connected to all its nodes and automatically receives information about globally registered names.

In this paper we discuss network scalability limitations of the *default* set-up for distributed Erlang. The s_groups we propose in Section 3 are designed to preserve the transitivity and the shared namespace of distributed Erlang while enabling scalability of applications.

In distributed Erlang the connections and namespace of a node are defined by both the node affiliation to a global_group and by the node type, namely hidden or normal. By a *namespace* we mean a set of names of processes replicated on a group of nodes and treated as global in that group. The name is either registered on all nodes or on none. Global name registration is mainly used to provide reliability. For example, a master process that communicates with worker processes from different nodes may need to be

globally registered, then worker processes communicate with it by name, rather than by process id (pid). If the master process fails, we restart and re-register it using the same name, and hence the worker processes can still communicate with the new master.

If a node is free, i.e. it belongs to no global_group, the connections and the namespace only depend on the node type. A *free normal node* has transitive connections and shares a common namespace with all other free normal nodes. A *free hidden node* has non-transitive connections with all other nodes and every hidden node has its own namespace. A *global_group node* can belong to only one global_group. Independently of its type – normal or hidden – a global_group node has transitive connections with the nodes from the same global_group and non-transitive connections with other nodes.

In Figure 1 we show transitive and non-transitive connections between different types of nodes where nodes N1, N2, N3, N4 are free normal nodes, nodes H5, H6 are free hidden nodes, and nodes S7, S8, S9, S10, S11, S12, S13, S14 are global_group nodes. Nodes S7, S8, S9, S10 are in global_group G1 and nodes S11, S12, S13, S14 are in global_group G2. The lines between the nodes represent different types of connections: a solid line denotes a transitive connection, and a dotted line denotes a non-transitive connection.

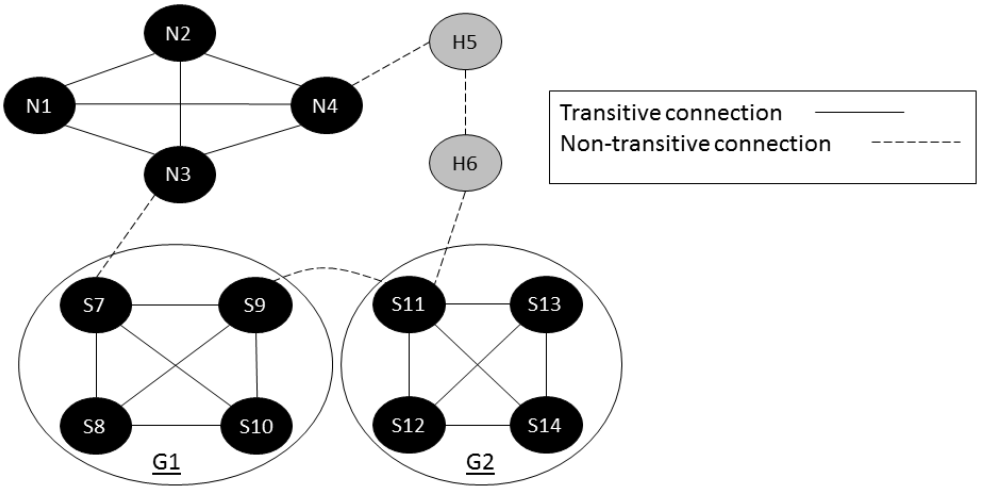


Figure 1: Types of Connections between Different Types of Nodes in Distributed Erlang

2.2. Scalability Limitations of Distributed Erlang

The main two network scalability limitations of distributed Erlang that led to introducing s_groups are global name sharing and transitive connections.

Global Name Sharing. To analyse the effect of global operations on the network scalability of distributed Erlang systems we have conducted experiments using the DEbench benchmarking tool (13) a P2P scalability tool based on Basho Bench (14). Nodes are interconnected, and every node runs on a separate host and has its own copy of DEbench. In the experiments we measure throughput depending on the number of nodes up to 100 (Figure 2), and we only use four operations: two local operations (spawn and RPC), and two global operations (name registration and unregistration). The percentage of global operations ranges from 0% to 0.1%. The results show that even a very small percentage of global operations significantly reduces system throughput, e.g. 0.005% of global operations prevents a linear increase of the throughput beyond 60 nodes.

Fully Connected Network. Maintaining a fully connected network between N nodes requires $N(N-1)/2$ or $O(N^2)$ connections. In distributed Erlang these are both live TCP/IP keepalive messages to maintain the connection, and distributed Erlang heartbeats to monitor the nodes. Clearly, as the number of nodes grows this places a load on the communication infrastructure. We analyse the network scalability of the Riak reliable NoSQL DataBase

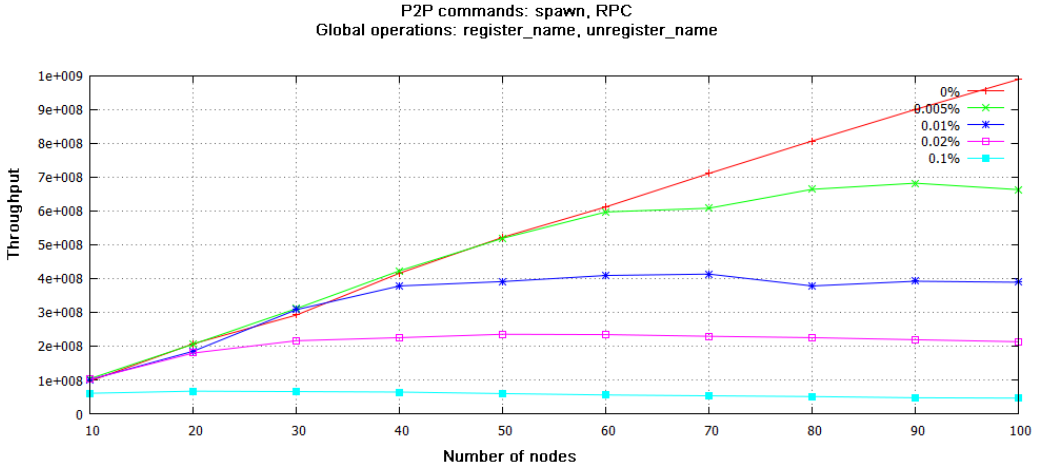


Figure 2: Impact of Percentage of Global Operations on Scalability

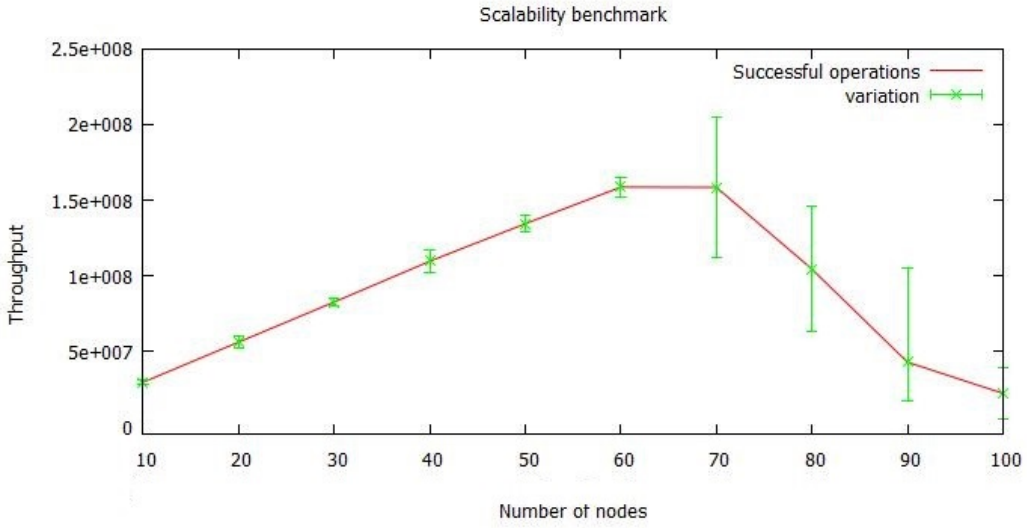


Figure 3: Network Scalability of Riak DBMS

Management System (DBMS) (15), and found that Riak 1.1.1 does not scale beyond 60 nodes. Figure 3 shows Riak 1.1.1 throughput as we increase the number of nodes from 10 to 100 (one node per host). We have investigated possible causes of the network scalability limitations and shown that neither disc, nor RAM, nor network limit scalability. Although Riak is very complex we find good reasons to believe that the number of connections limit scalability (16).

3. Scalable Group Design and Implementation

Scalable Distributed (SD) Erlang is a modest, conservative extension of distributed Erlang. SD Erlang introduces the following two concepts to improve scalability of distributed Erlang applications: scalable groups (s_groups) and semi-explicit placement. S_groups aim to reduce the number of connections maintained by nodes and hence the size of shared namespace. Semi-explicit placement aims to semi-automate the choice of an appropriate target node when spawning a process by introducing node attributes and communication distances. In this paper we only cover research related to the s_group part of the SD Erlang: essentially we address the question of how to scale a network of Erlang nodes by reducing the number of connections

between the nodes. A discussion of semi-explicit placement can be found in (17).

The design of SD Erlang is guided by the following principles for reliable network scalability. The principles include concepts that we want to either preserve or avoid when scaling distributed Erlang, namely (a) preserving the Erlang philosophy and programming idioms; (b) minimal language changes, by minimizing the number of new constructs and reusing the existing constructs; (c) keeping the Erlang/OTP reliability model unchanged as far as possible, so maintaining concepts of linking, monitoring and supervision.

3.1. Scalable Group Design

Not only does a fully connected graph of Erlang nodes imply a quadratic $O(N^2)$ number of active TCP/IP connections, but globally registered names are replicated on all nodes, and the name registration operations are global and synchronous, e.g. `register_name/2` is performed either on all nodes or on none. The larger the network of Erlang nodes the more expensive it becomes for each node to periodically check connected nodes and keep up-to-date replications of global names. We propose overlapping scalable groups (s_groups), where nodes transitively connected with other nodes within their s_groups, and non-transitively with other nodes.

In SD Erlang nodes with no asserted s_group membership belong to a notional group *G0* that follows distributed Erlang rules and hence allows backward compatibility. By backward compatibility we mean that when nodes run the same version of Erlang VM independently of their usage of s_groups the nodes are able to communicate with each other. Therefore, s_groups may be introduced to improve the network scalability of existing distributed Erlang systems.

To demonstrate transitive and non-transitive connections in SD Erlang we consider the following example. Assume we start six free normal nodes: A, B, C, D, E, F, then the nodes belong to the notional group $G0$ (Figure 4(a)). Note that a node belongs to the group $G0$ only when this node does not belong to any s_group. Assume also that nodes are interconnected; here, the fully connected network of Erlang nodes is not compulsory and is for a demonstration of transitive connections only. First, on node A we create a new s_group $G1$ that consists of nodes A, B, and C. When nodes A, B, and C become members of the s_group they keep connections with nodes D, E, F but now these connections are non-transitive. We then disconnect the nodes of s_group $G1$ from the nodes of group $G0$ using the function

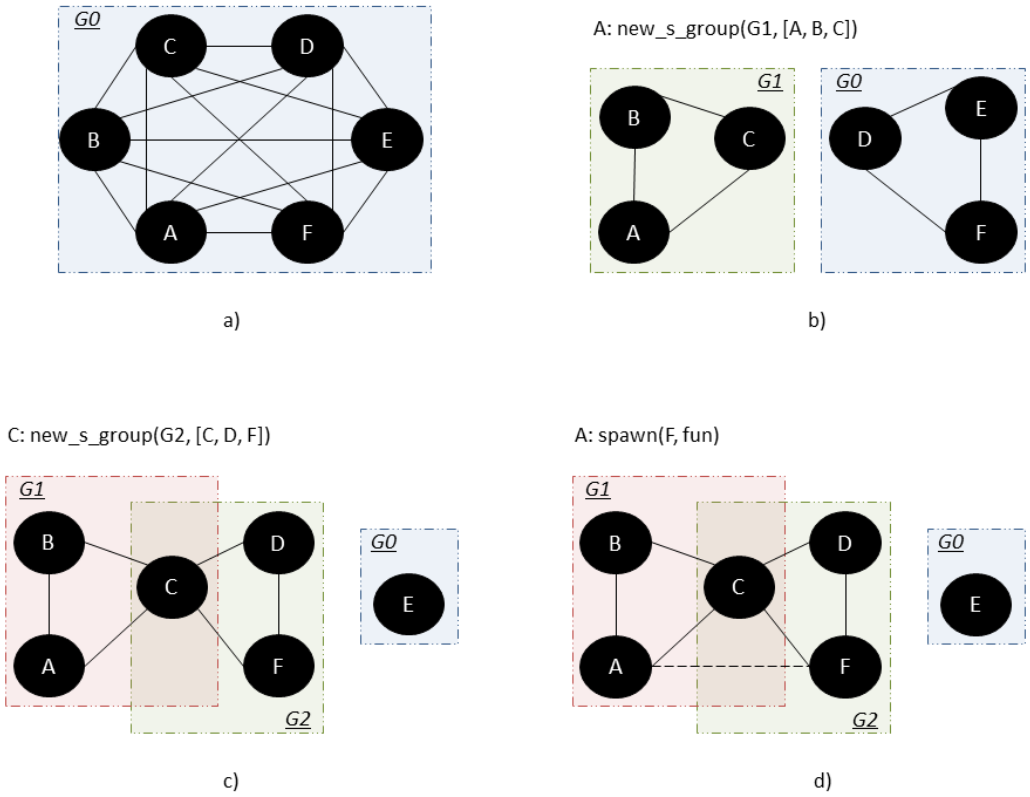


Figure 4: Connections in s_groups

`erlang:disconnect_node(Node)` (Figure 4(b)). After that on node C we create an s_group $G2$ that consists of the nodes C, D, and F. The nodes D and F now have non-transitive connections with the node E. We disconnect nodes D and F from node E. Figure 4(c) shows that node C does not share information about nodes A and B with nodes D and F. Similarly, when nodes A and F establish a direct connection they do not share the connection information with each other (in Figure 4(d) a dotted line represents a non-transitive connection). Note, however, that node disconnections are not compulsory, and are included here for the purposes of demonstration.

3.1.1. Design Alternatives

Before introducing s_groups we considered grouping nodes in hierarchical, overapping and partitioned groups. To choose the most appropriate approach we took into account the following principles.

- Preserving the distributed Erlang philosophy that any node can be directly connected to any other node.
- Dynamic adding and removing nodes from groups.
- Enabling nodes to belong to multiple groups.
- A simple mechanism.

A hierarchical approach prevents a node from being a member of different groups and also prevents there being direct connections between nodes from different levels and subgroups. We have therefore implemented *overlapping* s_groups, as this approach seems to best satisfy the Erlang philosophy and our goals. In addition, both overlapping and partitioned groups can be implemented using overlapping s_groups, so that we have enough generality with this choice.

Joe Armstrong (18) speculated about storing global data using an approach based on Distributed Hash Tables (DHTs), e.g. Kademlia (19) and Tapestry (20). In this case a reduction of the namespace and the number of connections would be achieved through a change of routing algorithm. That is, instead of establishing direct connections, nodes would communicate with each other via "hash close" nodes, and global names would be also stored on "hash close" nodes. However, implementing this approach would mean going against established Erlang philosophies such as "any node can be directly connected to any other node". It would also mean putting a restriction on developers, forcing them to use a particular network configuration. But most importantly, we do not know in advance how effective remote supervision will be (i.e. supervising a process via other processes due to a lack of direct connection between nodes on which the processes reside) and impact of extra load on routing nodes on the performance. Whereas using the SD Erlang s_groups the DHT approach could be implemented and analysed systematically before the actual implementation.

The idea of SD Erlang s_groups is similar to the distributed Erlang hidden global_groups in two ways: (a) each s_group has its own namespace, and (b) transitive connections are only with nodes from the same s_group. The differences from hidden global_groups are that (a) a node can belong to multiple s_groups which implies a different synchronisation mechanism, and (b) s_groups can be modified dynamically (21). The functionality of free nodes in SD Erlang is the same as it is in distributed Erlang. Table 1 provides a summary of types of connections and a division of the namespaces in distributed Erlang and SD Erlang.

No.	Grouping	Type of Connections	Namespace
Distributed Erlang			
1	No grouping	All-to-all connections	Common
2	Global_groups	Transitive connections within a global_group, non-transitive connections with other nodes	Partitioned
Scalable Distributed Erlang			
1	No grouping	All-to-all connections	Common
2	S_groups	Transitive connections within an s_group, non-transitive connections with other nodes	Overlapping

Table 1: Types of Connections and Namespace

The notion of s_groups is also similar to that of *MPI communicators* (22) but while an MPI communicator groups processes, an s_group groups Erlang nodes. Another difference is that s_groups aim to reduce common namespace and transitive connections, but, unlike MPI communicators, impose no other limitations or restrictions on node communications. In addition when s_groups are arranged in a hierarchical manner one can find similarities between Erlang nodes that belong to a number of s_groups (let us call them gateway nodes) and *super-peers*, i.e. nodes that act simultaneously as a server and a peer (23). However, in SD Erlang a gateway node having a super-peer functionality depends on an application and this role is not imposed by s_groups.

3.1.2. S_group Implementation

In SD Erlang connections and data replication between nodes that belong to the same s_group are handled by the following two Erlang processes: `global_name_server` and `s_group`. These processes are present on every node and are started when the node is launched. The `s_group` process is started from `s_group` module and is responsible for keeping information about s_groups. The `global_name_server` process is started from `global` module, and is responsible for keeping connections and common data on the nodes identified by the `s_group` process.

A node can become a member of an s_group either dynamically using `s_group:new_s_group/1,2` functions (Section 3.1.3) or at launch using the `-config` flag and the `.config` file. For example, Listing 1 presents the configuration of node C if nodes in Figure 4(d) join the s_groups at launch.

Listing 1: S_group configuration for node C in Figure 4(d)

```
[{kernel,[{s_groups,[
{group1,normal,['nodeA@glasgow.ac.uk','nodeB@glasgow.ac.uk',
        'nodeC@glasgow.ac.uk']},
{group2,normal,['nodeC@glasgow.ac.uk','nodeD@glasgow.ac.uk',
        'nodeE@glasgow.ac.uk']}]}]}].
```

The configuration file may contain information either about the s_groups of a particular node or about the whole system. In the latter case the node is aware of the remote s_groups and may interact with processes registered there (Section 3.1.4). That said, information from `.config` file about remote s_groups must used with caution because it is not updated during runtime and may be inconsistent with the actual group structure. We have introduced this functionality to explore the opportunities and challenges of dynamic configuration update but this has not been implemented yet. See further discussion in Section 8.2.

The SD Erlang implementation and measurements we present in this paper are based on Erlang/OTP 17.0 and 17.4. We call SD Erlang an extension because it only makes some changes in Erlang/OTP modules, but does not change Erlang VM. In particular in `lib/kernel/src/` directory SD Erlang replaces `global_group.erl` module with `s_group.erl` module to group nodes and modifies the following four modules: `global.erl`, `global_search.erl`, `kernel.erl`, `net_kernel.erl`. Instructions on how to build SD Erlang can be found in `http://www.dcs.gla.ac.uk/research/sd-erlang/`.

Erlang code that uses neither global_groups nor s_groups can be run on both distributed Erlang and SD Erlang. However, it is not advisable to use both types of nodes in the same application due to a modification of Erlang/OTP modules that handle connections.

3.1.3. S_group Functions

In this section we discuss three s_group functions related to grouping Erlang nodes: creating a new s_group, removing nodes from an s_group, and listing own and known nodes. The types of arguments in the functions below are as follows (24): `Name::term()`, `Pid::pid()`, `Node::node()`, `SGroupName::group_name()`, `Reason::term()`, `Msg::term()`. The description of the remaining six functions, such as deleting an s_group, adding

Function	Description
`new_s_group([Node])` `new_s_group(SGroupName, [Node])`	Creats new s_groups
`delete_s_group(SGroupName)`	Deletes an s_group
`add_nodes(SGroupName, [Node])`	Adds nodes to an s_group
`remove_nodes(SGroupName, [Node])`	Removes nodes from an s_group
`s_groups()`	Returns a list of all s_groups known to the node
`own_s_groups()`	Returns a list of s_group tuples of the s_groups the node belongs to

`own_nodes()`
 	
`own_nodes(SGroupName)`
	Returns a list of nodes the node shares namespaces with
	Returns a list of nodes from the given s_group

Table 2: Summary of S_group Specific Functions

nodes to an s_group, listing the nodes of own and known s_groups, synchronisation of nodes, and providing node information can be found in (21).

A summary of modified and new functions from global and s_group modules is presented in Tables 2 and 3; some of these functions we discuss in detail in Sections 3.1.3 and 3.1.4. Functions from module `global` have identical functionality on free nodes in distributed Erlang and SD Erlang.

Creating an S_group. The `s_group:new_s_group/1,2` functions are used to create new s_groups dynamically (Listing 2). A new s_group is created first on the initiating node and then the remaining nodes are added. If the initiating node either is not in the list of s_group nodes or is already a member of the s_group the function fails and an error is returned. When an s_group name is not provided the `crypto:strong_rand_bytes(30)` function is used to generate a random s_group name. The particular function was chosen as a proof of concept, and may be replaced by an alternative one that also guarantees high probability of name uniqueness.

Listing 2: New S_Group

```
s_group:new_s_group([Node]) -> {SGroupName,[Node]} |
                               {'error',Reason}
s_group:new_s_group(SGroupName,[Node]) -> {SGroupName,[Node]} |
                                          {'error',Reason}
```

global:	**s_group:**
`info()` Returns global state information	`info()` Returns s_group state information
`register_name(Name,Pid)` Registers a name on the connected free normal nodes	`register_name(SGroupName,Name,Pid)` Registers a name in the given s_group
`re_register_name(Name,Pid)` Re-registers a name on the connected free normal nodes	`re_register_name(SGroupName,Name,Pid)` Re-registers a name in the given s_group
`unregister_name(Name)` Unregisters a name on the connected free normal nodes	`unregister_name(SGroupName,Name)` Unregisters a name in the given s_group

`registered_names()`
Returns a list of all registered names on the node
	`registered_names(node,Node)`
	Returns a list of all registered names on the given node
	`registered_names(s_group,SGroupName)`
	Returns a list of registered names in the given s_group

`whereis_name(Name)`
Returns the pid of a name registered on a free node
	`whereis_name(SGroupName,Name)`
	Returns the pid of a name registered in the given s_group
	
	`whereis_name(Node,SGroupName,Name)`
	Returns the pid of a name registered in the given s_group. The name is searched on the given node

`send(Name,Msg)`
Sends a message to a name registered on a free node
	`send(SGroupName,Name,Msg)`
	Sends a message to a name registered in the given s_group
	
	`send(Node,SGroupName,Name,Msg)`
	Sends a message to a name registered in the given s_group. The name is searched on the given node

Table 3: Summary of Global and S_group Functions

Removing Nodes from an S_group. The `s_group:remove_nodes/2` function is used to dynamically remove nodes from an existing s_group (Listing 3). The initiating node cannot remove itself, and to remove other nodes it should be a member of the target s_group.

Listing 3: Removing Nodes from an S_group

```
s_group:remove_nodes(SGroupName,[Node]) -> 'ok'
```

After leaving an s_group the node unregisters the s_group names. In case the node belongs to no other s_group it becomes free. Which free node type it is – hidden or normal – depends on the flag with which the node was launched. If the node becomes a free hidden node then it just keeps its existing connections. If the node becomes a free normal node then apart from keeping its existing connections the node synchronises with other free normal nodes with which it has connections, and as a result shares their connections and namespace.

Listing Own Nodes. The `s_group:own_nodes/0,1` functions are used to list nodes with which the node shares namespaces (Listing 4). On an s_group node `s_group:own_nodes()` function returns a list of nodes from all s_groups the node belongs to. On a free node the function returns a list of connected free normal nodes.

Listing 4: List of Own Nodes

```
s_group:own_nodes() -> [Node]
s_group:own_nodes(SGroupName) -> [Node]
```

The `s_group:own_nodes(SGroupName)` function returns a list of nodes of the given s_group. In case the node does not belong to the s_group an empty list is returned. On a free node the function returns an empty list.

3.1.4. Name Registration Functions

In this section we discuss the following three functions related to manipulating registered names: name registration, listing registered names, and searching for registered names. The functions called on s_group nodes treat free nodes as if they belong to an `'undefined'` s_group. The detailed description of the remaining functions from Table 3 can be found in (21).

Name Registration. A name is registered with one of the `register_name/2,3` functions (Listing 5). On free nodes names are registered using `global:register_name(Name,Pid)`, and on s_group nodes names are registered using `s_group:register_name(SGroupName,Name,Pid)`. Neither name nor pid should be already registered in the given group, and only a node that belongs to that group can register a name in it.

Listing 5: Name Registration

```
global:register_name(Name,Pid) -> 'yes' | 'no'
s_group:register_name(SGroupName,Name,Pid) -> 'yes' | 'no'
```

If for some reason we want a name to be known to the whole network, then we cannot simply register it in every s_group, because when registering a process globally the node on which the process information is replicated establishes a link to the node on which the process resides. This is due to a mechanism of process monitoring. So, to avoid establishing direct connections between nodes from different s_groups a programmer needs to introduce a mechanism of forwarding messages to the s_group in which the process is registered, for example, via gateway nodes.

Listing Registered Names. A list of registered names is returned by the `registered_names/0,1` functions (Listing 6). The `global:registered_names()` function can be used on both s_group and free nodes; it returns a list of all names registered on the calling node.

The `s_group:registered_names/1` function can be used with one of the following two arguments: {`node,Node`} and {`s_group,SGroupName`}. With {`node,Node`} argument the `s_group:registered_names({node,Node})` function can be used on both s_group and free nodes; it works similarly to `global:registered_names()` function but returns registered names from the given node. If the node that owns the calling process is not connected to the target node then a new connection is established between the nodes. This connection will remain until, for example, it is decided to disconnect the nodes or one of the nodes fails. With {`s_group,SGroupName`} argument if the node that owns the calling process belongs to s_group `SGroupName` the `s_group:registered_names({s_group,SGroupName})` function returns a list of names registered in this s_group; if the node does not belong to s_group `SGroupName` but has information about it then the node establishes a connection with one of the nodes of the s_group. A node may have information about an s_group but not belong to it when s_groups are started at launch (Section 3.1.2), e.g. in Listing 1 node `nodeA@glasgow.ac.uk` has information about `group2` but does not belong to it, and therefore, does not share the group's namespace.

Listing 6: List registered names

```
global:registered_names() -> [Name]
s_group:registered_names({s_group,SGroupName}) ->
                                              [{SGroupName,Name}]
s_group:registered_names({node,Node}) -> [{SGroupName,Name}]
```

Searching for a Name. A registered name can be found using `whereis_name/1,2,3` functions presented in Listing 7. The name search is done sequentially, and as soon as the name is found its pid is returned. The `global:whereis_name(Name)` function on a free node returns a pid in case the name is found, otherwise it returns `'undefined'`. On an s_group node the function returns `'undefined'` because the s_group name is not specified.

Listing 7: Searching for a Registered Name

```
global:whereis_name(Name) -> Pid | 'undefined'
s_group:whereis_name(SGroupName,Name) -> Pid | 'undefined'
s_group:whereis_name(Node,SGroupName,Name) -> Pid | 'undefined'
```

The `s_group:whereis_name(SGroupName,Name)` function first checks the name in the node own registry. If the name is not found locally then it is searched in other known s_groups by picking a node from the given s_group, then establishing a connection with that node, and checking whether the name is registered on that node. The function returns a pid if the name is registered in the given group and the node is aware of that group.

The `s_group:whereis_name(Node,SGroupName,Name)` function searches the name only on the defined node independently of the type of the initiating node. If the initiating node and the target node are not connected, then the connection is established.

4. Operational Semantics

To provide a formal basis for program understanding and enable reasoning we introduce an operational semantics for the s_group operations, and validate the library against the semantics in the following section. The semantics also provides an intuition for the functions that enabled us to improve implementation of a number of functions.

We start by defining an abstract state of SD Erlang systems (Section 4.1), before defining each function as a transition between states (Section 4.2).

4.1. SD Erlang State

We define the SD Erlang system state and associated abstract syntax variables as shown in Figure 5. The state of a system is modelled as a four tuple comprising a set of *s_group*s, a set of *free_group*s, a set of *free_hidden_group*s, and a set of *node*s. Each type of groups is associated with nodes and has a namespace. An *s_group* additionally has a name, whereas a *free_hidden_group* consists of only one node, i.e. a hidden node simultaneously acts as a node and as a group, because as a group a hidden node has a namespace but does not share it with any other node. Free normal and hidden groups have no names, and are uniquely defined by the nodes associated with them. Therefore, group names, *gr_names*, are either *NoGroup* or a set of *s_group_name*s. A *namespace* is a set pairs of *name*s and process ids, *pid*s, and is replicated on all nodes of the associated group.

A *node* has the following parameters: *node_id* identifier, *node_type* that can be either hidden or normal, *connections*, and *group_names*, i.e. names of groups the node belongs to. The node can belong to either a list of s_groups or one of the free groups. The type of the free group is defined by the node type. Connections are a set of *node_id*s.

SD Erlang State 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. That is, for any state (grs, fgs, fhs, nds) $\{\Pi_{node\_id} grs, \Pi_{node\_id} fgs, \Pi_{node\_id} fhs\}$ is a partition of $\Pi_{node\_id} nds$ where Π_{node_id} is projection onto the *node_id* attribute, or set of attributes, of the tuples.

Assumptions. We make the following assumptions to simplify the state transitions. It is clearly desiarable to relax some of these assumptions in our future work on the SD Erlang semantics (Section 8.2).

1. No two s_groups have the same name, that is all *s_group_name*s are unique.
2. All *node_id*s identify some node. More formally, for all *node_id*s occurring in some state (grs, fgs, fhs, nds), there exists some node in *nds* with that *node_id*.
3. No failures occur.

$$
\begin{aligned}
&(grs, fgs, fhs, nds) \in \\
&\quad \in \{state\} \equiv \{(\{s\_group\}, \{free\_group\}, \{free\_hidden\_group\}, \{node\})\} \\
&\quad gr \in grs \equiv \{s\_group\} \equiv \{(s\_group\_name, \{node\_id\}, namespace)\} \\
&\quad fg \in fgs \equiv \{free\_group\} \equiv \{(\{node\_id\}, namespace)\} \\
&\quad fh \in fhs \equiv \{free\_hidden\_group\} \equiv \{(node\_id, namespace)\} \\
&\quad nd \in nds \equiv \{node\} \equiv \{(node\_id, node\_type, connections, gr\_names)\} \\
&\quad gs \in \{gr\_names\} \equiv \{NoGroup, \{s\_group\_name\}\} \\
&\quad ns \in \{namespace\} \equiv \{\{(name, pid)\}\} \\
&\quad cs \in \{connections\} \equiv \{\{node\_id\}\} \\
&\quad nt \in \{node\_type\} \equiv \{Normal, Hidden\} \\
&\quad s \in \{NoGroup, s\_group\_name\} \\
&\quad n \in \{name\} \\
&\quad p \in \{pid\} \\
&\quad ni \in \{node\_id\} \\
&\quad nis \in \{\{node\_id\}\} \\
&\quad m \in \{message\}
\end{aligned}
$$

Figure 5: SD Erlang State

4.2. Transitions

The transitions we present in this section have the following form:

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

meaning that executing command on node ni in $state$ returns $value$ and transitions to $state'$. The transitions use a number of auxiliary functions that we also define. In the following $\oplus$ denotes disjoint set union; and by $y' \equiv \bigoplus\{y \mid ...\}$ we mean that elements from all generated y sets are accumulated in one y' set.

In total we have implemented transitions of fifteen SD Erlang functions. Nine of these functions change their state after the transition, whereas the other six functions only return some state information but do not change the state after the transition. To illustrate the semantics we present the

transitions for three functions previously described in Sections 3.1.3 and 3.1.4: `s_group:register_name/3` and `s_group:new_s_group/2` change the state, and `s_group:whereis_name/2` does not. The full semantics is available in (25).

s_group:register_name/3. When registering name n for pid p in s_group s the pair (n, p) is added to the namespace ns of the s_group only if node ni is a member of s_group s and neither n nor p appears in the s_group namespace (Listing 5 in Section 3.1.4).

$$
\begin{aligned}
&((grs, fgs, fhs, nds), \text{register\_name}(s, n, p), ni) \\
&\quad \longrightarrow ((\{(s, \{ni\} \oplus nis, \{(n, p)\} \oplus ns)\} \oplus grs', fgs, fhs, nds), \text{True}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{If } (n, \_) \notin ns \wedge (\_, p) \notin ns \\
&\quad \longrightarrow ((grs, fgs, fhs, nds), \text{False}) \qquad \text{Otherwise}
\end{aligned}
$$

where

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

s_group:whereis_name/2. If node ni belongs to s_group s the function returns pid p registered as name n in the s_group, `undefined` otherwise (Listing 7 in Section 3.1.4). The `IsSGroupSNode` function returns either `True` or `False` depending on whether node ni belongs to s_group s. The `FindName` function searches for the pid p of a registered name n depending on the type of the group in which the name is registered, i.e. s_group, free normal, or free hidden group.

$$
\begin{aligned}
&((grs, fgs, fhs, nds), \text{whereis\_name}(s, n), ni) \\
&\quad \longrightarrow ((grs, fgs, fhs, nds), p) \qquad\qquad \text{If IsSGroupSNode}(ni, s, grs) \\
&\quad \longrightarrow ((grs, fgs, fhs, nds), \text{undefined}) \qquad\qquad \text{Otherwise}
\end{aligned}
$$

where

$$
\begin{aligned}
&\{(s, \{ni\} \oplus nis, ns)\} \oplus grs' \equiv grs \\
&p \equiv \text{FindName}(ni, s, n, grs, fgs, fhs, nds)
\end{aligned}
$$

s_group:new_s_group/2. When we create a new s_group s, the s_group together with its nodes nis are added to the list of s_groups. If before joining the s_group nodes nis are free then the nodes are removed from corresponding free groups fgs and fhs. The new s_group has an empty namespace (Listing 2 in Section 3.1.3). `InterConnectNodes` function interconnects

nodes from nds identified by nis node ids. `AddSGroup` function adds membership of s_group s to all nodes identified by nis node ids. `RemoveNodes` function removes node ids identified by nis from free normal groups fgs and free hidden groups fhs.

$$
\begin{aligned}
&((grs, fgs, fhs, nds), \text{new\_s\_group}(s, nis), ni) \\
&\quad \longrightarrow ((grs', fgs', fhs', nds''), (s, nis)) && \text{If } ni \in nis \\
&\quad \longrightarrow ((grs, fgs, fhs, nds), \text{Error}) && \text{Otherwise}
\end{aligned}
$$

where

$$
\begin{aligned}
nds' &\equiv \text{InterConnectNodes}(nis, nds) \\
nds'' &\equiv \text{AddSGroup}(s, nis, nds') \\
grs' &\equiv grs \oplus \{(s, nis, \{\})\} \\
(fgs', fhs') &\equiv \text{RemoveNodes}(nis, fgs, fhs)
\end{aligned}
$$

5. Validation of Conformance

A specification is of little value if there is no attempt made to check that it corresponds to its implementation. In order to ensure conformance between the SD Erlang semantic specification and the actual implementation, we have implemented an executable version of the semantic specification based on the formal mathematical definition of Section 4. This work is reported in full in (26); in this section we give an introduction to the approach, as well as a full statement of the results. The paper (26) provides a comprehensive account of the work and its background.

5.1. Property-based testing

The executable semantic specification is implemented within the property and model-based random testing framework provided by the Erlang testing tool QuickCheck (10). Property-based testing (PBT) provides a high-level approach to testing: rather than focusing on individual test cases, in PBT the required behaviour is specified by properties, expressed in a logical form. For example, a function without side effects might be specified by means of the full input/output relation using a universal quantification over all the inputs; a stateful system will be described by means of model, which is an extended finite state machine. The system is then tested by checking whether it has the required properties for randomly generated data, which may be

inputs to functions, sequences of API calls to the stateful system, or other representations of test cases. Since SD Erlang has a stateful API, we use the modelling approach here.

The advantage of writing the executable semantics within the QuickCheck testing framework is that it allows us to test the conformance between semantics and implementation as well as, *inter alia*, the correctness of semantic specification and the correctness of implementation. As a link between the formal mathematical specification and the implementation, the executable model makes it more feasible for the co-evolution of specification and implementation; it also provides us with a means to explore the new features to be added to the library without having to provide a full implementation of them.

5.2. The Validation Approach

The architecture of the testing framework is shown in Figure 6. First we define an abstract state machine `eqc_statem` client module that is the executable version of the semantics. The source code can be found at `https://github.com/huiqing/s_group/blob/master/s_group_eqc.erl`.

The state machine defines the abstract state representation and the transition from one state to another when an operation is applied. Test case and data generators are then defined to control the test case generation; this includes the automatic generation of eligible s_group operations and the input

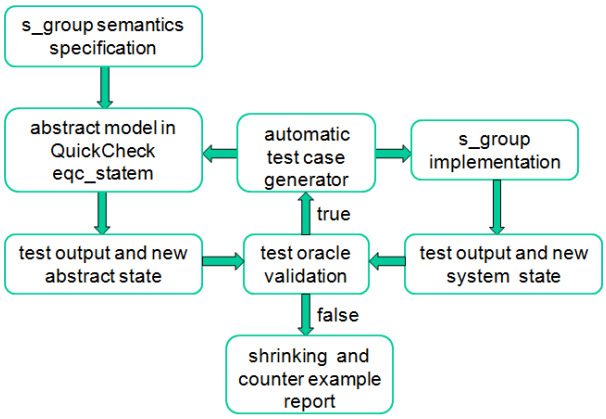


Figure 6: Testing s_groups Using QuickCheck

data to those operations. Test oracles are encoded as the postcondition for s_group operations.

During testing, each test command is applied to both the abstract model and the actual s_group implementation. The application of the test command to the abstract model takes the abstract model from its current state to a new state as described by the transition functions; whereas the application of the test command to the real system leads the system to a new actual state. The actual state information is collected from each node in the distributed system, then merged and normalised to the same format as the abstract state representation. In order for a test to be successful, after the execution of a test command, the test oracles specified for this command should be satisfied. Various test oracles can be defined for s_group operations; for instance one of the generic constraints that applies to all the s_group operations is that after each s_group operation, the normalised system state should be equivalent to the abstract state.

By default, QuickCheck generates 100 test cases for each run, with each test case consisting of a sequence of test commands. The number of test cases to test can be changed however. Testing is deemed to be successful if all the test cases have been passed, otherwise a test fails and a 'shrunk' counter-example is returned.

5.2.1. Results

The model covering the nine s_group operations contains 1,100 lines of code. So far, thousands of tests have been run using this test model. In this section, we summarise the kinds of errors encountered during testing.

- Errors in the *test code*. Test code is code, hence not immune from errors. As a result, some of the errors encountered, especially in the early stage of the testing, were errors in the test code itself.

- Errors in the *semantic specification*. In this case, the actual state is different from the abstract state after some test execution, and human examination identifies that the actual state represents the expected result.

  We found two semantic errors during testing. One error was that a free normal node was not properly removed from its original free group when the node joins an s_group; the other error was due to erroneous manipulation of the *gr_names* of a node resulting that *gr_names* contains both *NoGroupName* and an s_group name.

- Errors in the *implementation*. An error in the implementation also leads to a disagreement between the actual state and the abstract state, but in this case the abstract state represents the expected result.

  Our testing revealed two errors in the implementation. One error was due to the synchronisation between nodes where one node was expecting a `'nodeup'` message from another node but failed to receive it after a time-out although the other node was actually up; the other error was related to the `remove_nodes` operation, where a mismatch between the expected result and actual value returned by a list search operation happened and crashed the Erlang node.

- *Inconsistency* between semantics and implementation. In this case, although the actual system state and the abstract state are equivalent, the value returned by the implementation and the abstract state machine are not always the same.

  In one case the formal semantics specified that the `send` operation should return `'undefined'` as the result if the message receiving process does not exist, however the actual implementation returned a tuple with the first element as `'badarg'` and the second element being the arguments supplied; in another case the semantics specified that the `unregister_name` operation always returns `'True'`, whereas the implementation could also return {`no, cannot_unregister_from_remote_group`}.

The results show the value of the executable approach, in that we were able not only to debug the implementation, but also to debug the formal semantics itself, as well as the consistency between the semantics and its implementation.

6. Preliminary Evaluation

In this section we discuss the results of the preliminary evaluation of the SD Erlang implementation. The evaluation includes measurements with a test harness where we can control key network scalability aspects, such as percentage of global operations using DEbench (Section 6.1), and an analysis of the impact of transitive connections on the scalability of a distributed application using Orbit benchmark (Sections 6.2).

DEbench is selected for these benchmarks as it enables us to investigate both the impact of reduced number of connections and of smaller namespaces.

In contrast Orbit demonstrates the impact of reduced number of connections alone.

6.1. The DEbench Measurement Harness

To analyse the impact of global operations on network scalability of SD Erlang we again use the DEbench tool (Section 2.2). This time we compare distributed Erlang results with corresponding SD Erlang results when the transaction mix contains 0.01 per cent of global operations. Recall that in context of this paper a global operation is an operation that is applied to all nodes of a group and treated as global in that group (Section 2.1). The experiments are based on Erlang/OTP 17.0.

In the SD Erlang version we partition a set of nodes in such a way that every s_group has 10 nodes. Therefore, when we register a name in distributed Erlang the name is replicated on all nodes, whereas in SD Erlang the name is replicated on 10 nodes of a particular s_group. Recall that by a *global operation* we mean an operation that involves all nodes from a given namespace. We ran the experiments varying the number of nodes between 10 (80 cores) and 100 (800 cores). The results presented in Figure 7 show that up

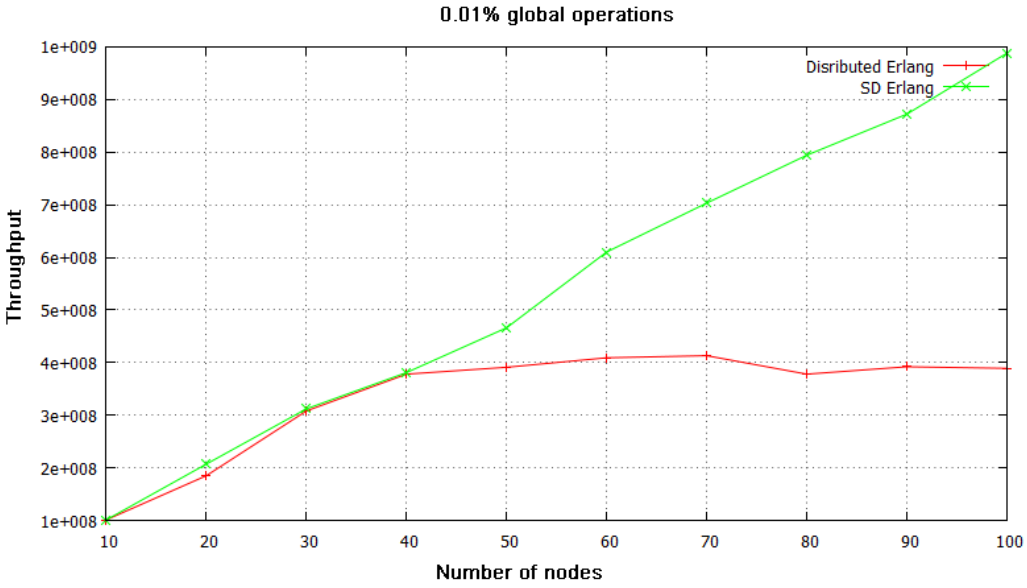


Figure 7: Impact of Global Operations on Network Scalability of Distributed Erlang and SD Erlang

to 40 nodes distributed Erlang and SD Erlang perform similarly, and beyond 40 nodes the throughput of distributed Erlang stops increasing, whereas the throughput of SD Erlang continues to grow linearly.

6.2. Orbit

To evaluate the impact of reduced number of connections when introducing s_groups on the network scalability of Erlang applications we have conducted experiments using the Orbit benchmark (27), i.e. a symbolic computing kernel and a generalization of a transitive closure computation. We have chosen Orbit as a case study as it uses a Distributed Hash Table (DHT) similar to NoSQL DBMS like Riak and standard P2P techniques. Orbit is only a few hundred lines of code, and has a good performance and extensibility. To compute Orbit for a given space $[0..X]$ a list of generators $g1, g2, ..., gn$ are applied on the initial vertex $x_0 \in [0..X]$ that creates new numbers $(x_1...x_n) \in [0..X]$. The generator functions are applied on the new numbers until no new number is generated. We have implemented distributed Erlang and SD Erlang versions of Orbit (28) where neither version uses global operations. The computation is started on the master node, and then is distributed between worker nodes.

We ran Orbit experiment on a cluster located in EDF, France, called Athos. For the experiments we had simulataneous access to up to 257 compute nodes (6168 cores) for up to 8 hours at a time. Each Athos node has 64GB of RAM and an Intel Xeon E5-2697 v2 processor with 24 cores. In the Orbit experiments each worker node has 8 DHTs. The number of nodes varied between 1 (24 cores) and 257 (6168 cores). The experiments are based on Erlang/OTP 17.4.

The *distributed Erlang implementation* of Orbit has one master node and the remaining nodes are workers. All nodes are interconnected.

The *SD Erlang implementation* of Orbit has one master node and the remaining nodes are submasters and workers. The nodes are grouped into sets of s_groups (Figure 8). Within an s_group nodes communicate directly with each other but to reach a node outside of an s_group the communication is done via the submaster nodes. The s_groups reduce the number of connections between nodes, i.e. the number of connections of a worker node is equal to the number of worker nodes in its s_group, and the number of connections of a sub-master node is equal to the number of connections of a worker node *plus* the number of sub-master nodes. Every s_group has one

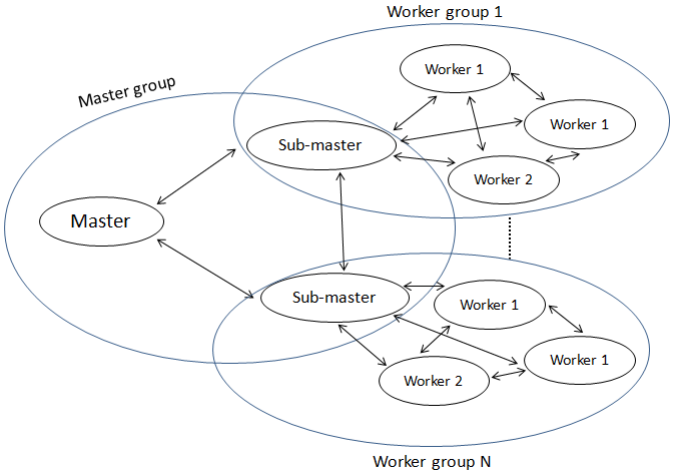


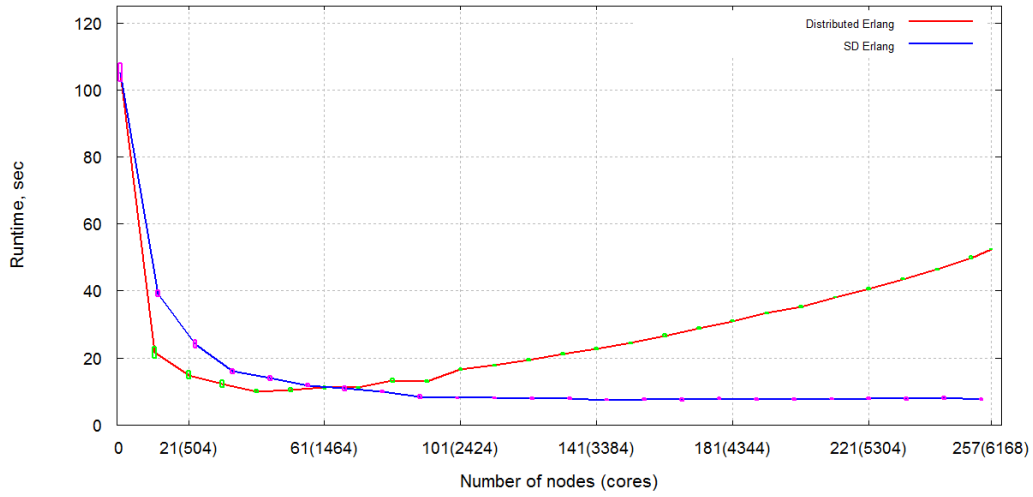
Figure 8: Communication Model in SD Erlang Orbit

submaster node and ten worker nodes. Every sub-master node has 40 gateway processes that perform transferring of messages between worker nodes from different s_groups.

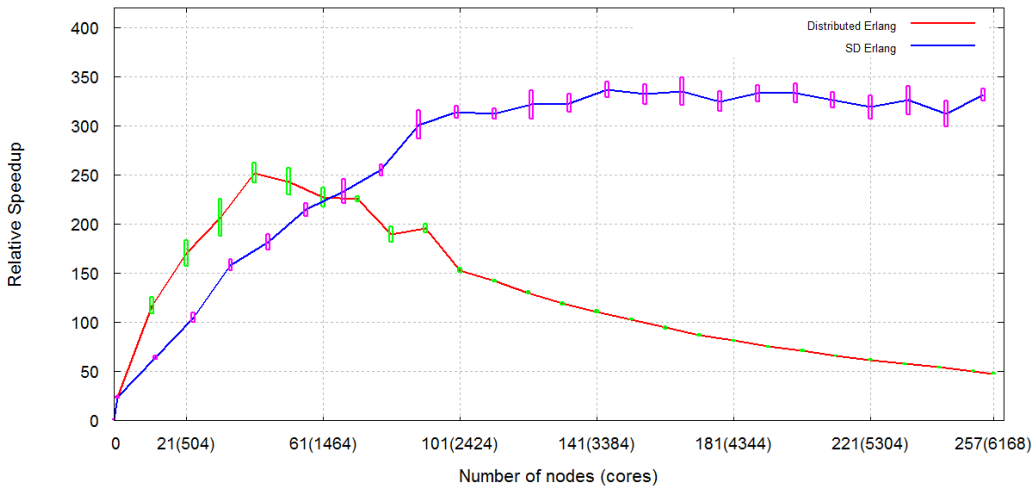
Figures 9(a) and 9(b) compare the runtime and the speed-up of distributed Erlang and SD Erlang implementations. Every experiment was repeated 7 times, and the median results are plotted in the diagrams. The vertical segments depict 95% confidence interval. The speedup is a ratio between execution time on one node with one core and the execution time on corresponding number of nodes and cores. The results show that performance of distributed Erlang version starts degrading after 40 nodes (984 cores). SD Erlang performs better on larger scales – beyond 80 nodes (1920 cores) – and the performance does not degrade as the number of nodes grows. The results confirm our expectations that on a small scale SD Erlang performs a bit worse than distributed Erlang but the larger the scale the better SD Erlang performs in comparison with distributed Erlang.

7. Actor Languages & Frameworks

The Erlang programming model and philosophy is widely acknowledged as very effective. It has influenced and inspired a number of languages and frameworks, including Akka (29), Cloud Haskell (30), APRIL (31), and Kilim (32). The most well-known ones – Akka and Cloud Haskell – we discuss



(a) Network Scalability



(b) Relative Speedup

Figure 9: Impact of Transitive Connections on Network Scalability of Distributed and SD Erlang Orbit

here in more details.

Akka is an event-driven middleware framework to build reliable distributed applications (29). Akka is implemented in Scala, a statically typed programming language that combines features of both object-oriented and functional programming languages. Fault tolerance in Akka is implemented using similar to Erlang 'Let it crash' philosophy and supervisor hierarchies (33). An actor can only have one supervisor which is the parent supervisor but similarly to Erlang actors can monitor each other. Due to the possibility of creating an actor within a different Java VM, two mechanisms are available for accessing an actor: logical and physical. A logical path follows parental supervision links toward the root, whereas, a physical actor path starts at the root of the system at which the actual actor object resides, but cannot reference actors on other Java VMs. Like Erlang Akka does not support guaranteed delivery. As far as we know, cluster support for Akka is only planned to be introduced (34; 35).

Cloud Haskell (30) is a domain specific language embedded as a library in the Haskell functional programming language (36). From Haskell the language inherits purity, types and monads. As a pure functional programming language Haskell provides immutability of data, and types and monads statically guarantee program properties. Similarly to Erlang the processes in Cloud Haskell are lightweight and are central for the concurrency. In contrast to Erlang, Cloud Haskell allows shared-memory concurrency within a process. The language utilises Erlang message-passing mechanism, i.e. processes do not share data and communicate with each other only via message passing. However, in contrast to Erlang, where messages can be sent to process for which the sender has the address (or name) of the recipient, in Cloud Haskell the messages are sent via two types of channels: untyped and typed. Here, the incoming messages are matched by type. The supervision philosophy for the fault tolerance is also borrowed from Erlang, i.e. processes are monitored and can be restarted following a failure. (37) presents network scalability measurements on up to 160 cores.

The above shows that Akka and Cloud Haskell are heavily influenced by Erlang and apply many of Erlang properties and philosophy. When scaling these languages over a set of nodes we believe the programmers will find useful our experience of scaling Erlang. This does not mean though that the functionality or wording should be the same to have a similar impact. For example, such property as shared-memory concurrency within a process should not have an effect on a scalability of a set of nodes. It may have

an impact on a performance of a single node or rubbish collection but as processes are isolated from each other this should not effect processes on remote nodes and scalability of a set of nodes in particular.

On the other hand, both Akka and Cloud Haskell do not support transitive connections, however monitoring a process on a remote node implies a connection between the nodes and a heart-bit signal. As the number of nodes in the system grows nodes likely to maintain a larger number of connections which will have a negative impact on scalability, so a restriction of connections to sub-groups of nodes may be advisable. The same principle applies to global namespace and global operations. Applying global operations to a subset of nodes rather than to all nodes should significantly improve scalability.

8. Conclusion and Future Work

8.1. Conclusion

We address the network scalability limitations of distributed Erlang (Section 2.2) by presenting the design and implementation of SD Erlang – a small conservative extension of distributed Erlang (Section 3). We discuss the main aspects of s_group design and implementation. That is nodes have transitive connections with nodes from the same s_groups and non-transitive connections with other nodes. Free nodes in SD Erlang have the same functionality as in distributed Erlang. In total we introduce sixteen functions of two types: s_group functions that manipulate s_groups, for example, creating an s_group and listing all s_groups of a particular node, and name registration functions that support registration of names in s_groups, for example, unregistration of a name and listing names registered in a particular s_group.

We provide a semantics for s_groups by defining an abstract state of SD Erlang systems and presenting the transitions of fifteen SD Erlang functions (Section 4). Nine of the functions change their state, whereas the remaining six functions do not. We validate the consistency between the formal semantics and the SD Erlang implementation using Erlang QuickCheck testing tool (Section 5). Apart from validating the semantics the test enabled us to validate the implementation of the SD Erlang functions, and the conformance between the semantics and implementation. We provide the details of the testing approach and discuss the errors that we encountered while working on the semantics, the implementation, and the validation.

We provide the preliminary evaluation of SD Erlang performance compared with distributed Erlang (Section 6). The results show that introducing s_groups improves network scalability. We analyse the impact of global operations on network scalability of distributed Erlang and SD Erlang applications using DEbench benchmarking tool. The experiments are conducted on 10–100 nodes (80–800 cores). The results show that with 0.01% of global opperations the distributed Erlang version stops scaling beyond 40 nodes (320 cores) whereas the SD Erlang version continues to scale (Figure 7). The impact of all-to-all connections is analysed using the Orbit benchmark. In the experiments we utilise between 1 and 257 nodes (24 and 6168 cores). The results show that on a small scale (up to 40 nodes or 960 cores) distributed Erlang version of Orbit performs better than SD Erlang one, but as the number of nodes grows (beyond 80 nodes or 1920 cores) SD Erlang outperforms distributed Erlang (Figure 9).

8.2. Future Work

We plan to proceed the work on SD Erlang in the directions outlined below. Ultimately, we aim SD Erlang to be included in the standard Erlang/OTP.

Evaluation of SD Erlang reliability. To analyse SD Erlang reliability in comparison with distributed Erlang we develop an Instant Messenger (IM) benchmark. From the IM experiments we expect to get a better understanding if additional features need to be added to ensure application fault tolerance when using s_groups.

SD Erlang semantics. We plan to relax some of assumptions of the SD Erlang semantics discussed in Section 4.1, and in particular to consider failures. We hope this will provide a deeper understanding of Erlang's non-defence approach to fault tolerance.

Dynamic information updating about remote s_groups. In SD Erlang we introduced a possibility for a node to be aware of other s_groups (Section 3.1.2). Currently, this information is static and a node can get it only via the `.config` file at launch. The idea is to introduce a dynamic updating of this information. We conduct this work in conjunction with the work on semi-explicit placement (17) that also requires an up-to-date information about the network of nodes to make reasonable placement decisions.

Patterns and properties. We analyse different SD Erlang applications to identify common patterns and properties of s_groups. The work on the s_group patterns includes introducing functions to group nodes according

to different structures, and identifying refactoring mechanisms for gateway processes that route messages between nodes from different s_groups. We also work on identifying s_group properties, such as the best ratio of the number of worker nodes to the number of submaster nodes (see for example Figure 8).

Acknowledgements. We would like to thank our RELEASE project colleagues for technical insights. This work has been supported by the European Union grant RII3-CT-2005-026133 'SCIEnce: Symbolic Computing Infrastructure in Europe', IST-2011-287510 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software', and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 'HPC-GAP: High Performance Computational Algebra and Discrete Mathematics'.

[1] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007.

[2] C. Hewitt, P. Bishop, R. Steiger, A universal modular ACTOR formalism for artificial intelligence, in: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1973, pp. 235–245.

[3] J. E. Richardson, M. J. Carey, D. T. Schuh, The design of the E programming language, ACM Trans. Program. Lang. Syst. 15 (3) (1993) 494–534.

[4] F. Cesarini, S. Thompson, Erlang Programming: A Concurrent Approach to Software Development, 1st Edition, O'Reilly Media, 2009.

[5] A. Goldberg, D. Robson, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, Boston, MA, USA, 1983.

[6] G. Agha, An overview of actor languages, SIGPLAN Not. 21 (10) (1986) 58–67.

[7] J. Armstrong, Erlang, Commun. ACM 53 (2010) 68–75.

[8] Ericsson AB, Erlang/OTP Efficiency Guide, System Limits, www.erlang.org/doc/efficiency_guide/advanced. html#id67011 (2014).

[9] O. Boudeville, F. Cesarini, N. Chechina, K. Lundin, N. Papaspyrou, K. Sagonas, S. Thompson, P. Trinder, U. Wiger, RELEASE: A high-level paradigm for reliable large-scale server software, in: In Proceedings of the 13th International Symposium on Trends in Functional Programming, Vol. 7829, Springer, 2012, pp. 263–278.

[10] J. Hughes, QuickCheck testing for fun and profit, in: Practical Aspects of Declarative Languages, Springer, 2007, pp. 1–32.

[11] SpilGames, Spapi-router, https://github.com/spilgames/spapi-router (2014).

[12] Erlang Solutions, Megaload - The Age of Load Testing, https://www.erlang-solutions.com/resources/webinars/megaload-age-load-testing (2014).

[13] A. Ghaffari, Investigating the scalability limits of distributed Erlang, in: Proceedings of the 13th ACM SIGPLAN Workshop on Erlang, ACM, 2014, pp. 43–49.

[14] Basho Technologies, Riakdocs. Basho Bench, http://docs.basho.com/riak/latest/ops/building/benchmarking/ (2014).

[15] R. Klophaus, Riak core: Building distributed applications without shared state, in: ACM SIGPLAN Commercial Users of Functional Programming, ACM, New York, NY, USA, 2010, pp. 14:1–14:1.

[16] A. Ghaffari, N. Chechina, P. Trinder, J. Meredith, Scalable persistent storage for Erlang: Theory and practice, in: Proceedings of the 12th ACM SIGPLAN Workshop on Erlang, ACM, New York, NY, USA, 2013, pp. 73–74.

[17] K. MacKenzie, N. Chechina, P. Trinder, Performance portability through semi-explicit placement in distributed Erlang, in: Proceedings of the 14th ACM SIGPLAN Workshop on Erlang, ACM, 2015, pp. 27–38.

[18] Ericsson AB, Inside Erlang – Creator Joe Armstrong Tells His Story, http://www.ericsson.com/news/141204-inside-erlang-creator-joe-armstrong-tells-his-story_244099435_c (2014).

[19] P. Maymounkov, D. Mazieres, Kademlia: A peer-to-peer information system based on the xor metric, in: Peer-to-Peer Systems, Springer, 2002, pp. 53–65.

[20] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, J. D. Kubiatowicz, Tapestry: A resilient global-scale overlay for service deployment, Selected Areas in Communications, IEEE Journal on 22 (1) (2004) 41–53.

[21] N. Chechina, H. Li, P. Trinder, A. Ghaffari, Scalable SD Erlang computation model, Tech. Rep. TR-2014-003, The University of Glasgow (December 2014).

[22] J. J. Dongarra, S. W. Otto, M. Snir, D. Walker, An introduction to the MPI standard, Tech. rep., University of Tennessee, Knoxville, TN, USA (1995).

[23] B. Beverly Yang, H. Garcia-Molina, Designing a super-peer network, in: Proceedings of the 19th International Conference on Data Engineering, 2003, pp. 49–60.

[24] Ericsson AB, Types and Function Specifications, http://www.erlang.org/doc/reference _manual/typespec.html (2013).

[25] N. Chechina, H. Li, S. Thompson, P. Trinder, Scalable SD Erlang reliability model, Tech. Rep. TR-2014-004, The University of Glasgow (December 2014).

[26] H. Li, S. Thompson, Improved semantics and implementation through property-based testing with QuickCheck, in: Proceedings of the 9th International Workshop on Automation of Software Test, AST 2014, ACM, 2014, pp. 50–56.

[27] F. Lübeck, M. Neunhöffer, Enumerating large Orbits and direct condensation, Experimental Mathematics 10 (2) (2001) 197–205.

[28] RELEASE Project, Benchmarks, https://github.com/release-project/benchmarks (2014).

[29] Typesafe Inc., Akka: Event-driven middleware for Java and Scala, www.typesafe.com/technology/akka (2012).

[30] J. Epstein, A. P. Black, S. Peyton-Jones, Towards Haskell in the Cloud, SIGPLAN Not. 46 (12) (2011) 118–129.

[31] F. G. McCabe, K. L. Clark, APRIL – agent process interaction language, in: Proceedings of the Workshop on Agent Theories, Architectures, and Languages on Intelligent Agents, Springer-Verlag New York, Inc., New York, NY, USA, 1995, pp. 324–340.

[32] S. Srinivasan, A. Mycroft, Kilim: Isolation-typed actors for Java, in: ECOOP'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 104–128.

[33] Typesafe Inc., Akka Documentation: Release 2.1 - Snapshot, http://www.akka.io/docs/akka/snapshot/ (July 2012).

[34] D. Trabold, H. Grosskreutz, Parallel subgroup discovery on computing clusters – first results, in: Big Data, 2013 IEEE International Conference on, 2013, pp. 575–579.

[35] J. He, P. Wadler, P. Trinder, Typecasting actors: From Akka to TAkka, in: Proceedings of the Fifth Annual Scala Workshop, SCALA'14, ACM, New York, NY, USA, 2014, pp. 23–33.

[36] S. Thompson, Haskell: The Craft of Functional Programming, 3rd Edition, Addison-Wesley Publishing Company, USA, 2008.

[37] O. Batchelor, R. Green, Cloud Haskell: First impressions and applications to processing large image datasets, in: Proceedings of the 28th International Conference on Image and Vision Computing New Zealand (IVCNZ), 2013, 2013, pp. 412–417.