ICT-287510<br>
RELEASE<br>
A High-Level Paradigm for Reliable Large-Scale Server Software<br>
A Specific Targeted Research Project (STReP)

# D3.2 (WP3): Scalable SD Erlang Computation Model

Due date of deliverable: 31st July 2013
Actual submission date: 8th August 2013

Start date of project: 1st October 2011 Duration: 36 months

Lead contractor: The University of Glasgow Revision: 2.0

**Purpose:** To implement and validate a scalable computation model and a high-level process placement control for Scalable Distributed (SD) Erlang.

**Results:** The main results of this deliverable are as follows.

- To improve the scalability of distributed Erlang we have split the global namespace and reduced the number of connections following the design presented in deliverable D3.1. That is we group Erlang nodes into s_groups where each s_group has its own namespace.
- We have implemented sixteen functions to support node grouping and name registration.
- To support semi-explicit placement we have added node attribute parameter and implemented choose_nodes/1 function. The function returns a list of nodes that satisfy given restrictions.

**Conclusion:** We have implemented and started the validation of a scalable computation model and a high-level process placement control for SD Erlang.

| Project funded under the European Community Framework 7 Programme (2011-14) | | | |
|---|---|---|---|
| **Dissemination Level** | | | |
| PU | Public | | * |
| PP | Restricted to other programme participants | (including the Commission Services) | |
| RE | Restricted to a group specified by the consortium | (including the Commission Services) | |
| CO | Confidential only for members of the consortium | (including the Commission Services) | |

# Scalable SD Erlang Computation Model

## Contents

## 1 Executive Summary

The deliverable presents implementation of s_groups and semi-explicit placement of the Scalable Distributed (SD) Erlang introduced in D3.1 The Design of Scalable Distributed Erlang deliverable. The implementation is done on the basis of Erlang/OTP R15B03. The source code can be found in `https://github.com/natalia-chechina/otp`. The first version of D3.2 deliverable, i.e. an implementation and validation of a scalable computation model, was delivered in month 18.

We start with a discussion of differences between distributed Erlang global_groups and SD Erlang s_groups (Section 2). Then we discuss the implementation of s_groups and the features of sixteen functions that were modified and introduced in `global` and `s_group` modules (Section 3). After that we discuss semi-explicit placement, node attributes and `choose_node/1` function (Section 4). The functions were unit tested, and their demonstration will be presented at the yearly review (Section 5). Finally, we discuss future work (Section 6).

## 2 Introduction

The objectives of Task 3.2 are to "implement and validate a scalable computation model for Scalable Distributed (SD) Erlang using a combination of layering, controlling connection locality, and high-level process placement control". The lead participant is the University of Glasgow.

SD Erlang is implemented as a small conservative extension of distributed Erlang [REL12]. In distributed Erlang node connections and namespace are defined by both the node belonging to a global_group and by the node type, i.e. hidden or normal. A namespace is a set of names replicated on a group of nodes and treated as global in that group. Thus, if a node is free, i.e. it does not belong to a global_group, the connections and namespace only depend on the node type. A free *normal* node has transitive connections and 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

| 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** | |
| 3 | No grouping | All-to-all connections | Common |
| 4 | S_groups | Transitive connections within an s_group, non-transitive connections with other nodes | Overlapping |

Table 1: Types of Connections and Namespace

global_group node can belong to only one global_group and has transitive connections and common namespace with nodes that belong to the same global_group. The type of a global_group node – *normal* or *hidden* – only defines the types of connections with nodes outside its own global_group. Thus, a *normal* global_group node forms non-transitive visible connections with other nodes, and a *hidden* global_group node forms non-transitive hidden connections with other nodes. A global_group can also be one of the following two types: normal or hidden. Global_groups of both types may have normal and hidden nodes; however, in a hidden global_group all nodes act as hidden independently of the type they were started with. For example, in Figure 1 nodes N1, N2, N3, N5, N7, N8 are normal, and nodes H4, H6, H9 are hidden. Nodes N1, N2, N3 are in global_group G1, nodes H4, N5, H6 are in global_group G2, and nodes N7, N8, H9 are free. The lines between the nodes represent different types of connections, i.e. a solid line denotes a visible transitive connection, a wavy line denotes a non-transtitive visible connection, and a dotted line denotes a non-transitive hidden connection.

The type of connection between two nodes is recorded in `net_kernel` module when the connection is established. To view the types of connections of the connected nodes the following functions are used: `nodes()` function returns a list of connected nodes with visible type of connection, and `nodes(hidden)` returns a list of nodes with hidden type of connection.

The SD Erlang s_groups are *similar* to the distributed Erlang hidden global_groups in the following: 1) each s_group has its own namespace; 2) transitive connections are only with nodes of the same s_group. The *differences* with hidden global_groups are in that 1) a node can belong to an unlimited number of s_groups, and 2) information about s_groups and nodes is not globally collected and shared [REL12, Section 6.1]. Table 1 provides a summary of types of connections and a division of the namespaces in distributed Erlang and SD Erlang. In SD Erlang behaviour and functionality of free nodes remains the same as in distributed Erlang.

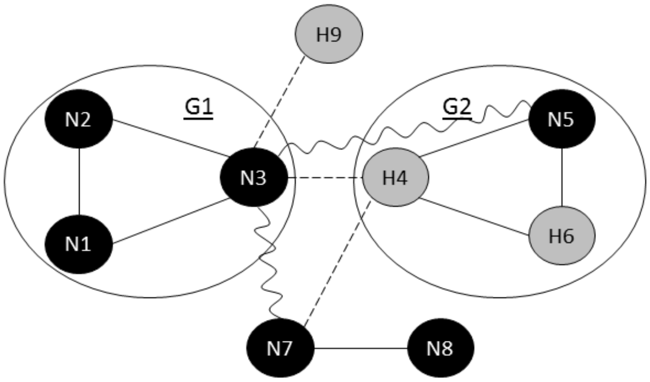


Figure 1: Types of Connections between Different Types of Nodes in distributed Erlang

**Partner Contributions to D3.2.** The University of Kent contributed to the implementation of SD Erlang. The Ericsson team contributed to the identifying of the main components in Erlang/OTP responsible for connections and namespace in distributed Erlang. ESL and ICCS contributed to the review of the implementation design. We had a series of teleconferences with Ericsson, University of Kent, and Erlang Solutions, and a week visit to the University of Kent. There were two multi-partner face-to-face meetings in Denmark and Greece.

We discuss the implementation design of s_groups in Section 3 and semi-explicit placement in Section 4. The validation of the functions is discussed in Section 5. The conclusion and future work are covered in Section 6.

# 3 S_group Implementation Design

## 3.1 Why S_groups?

The main reason we have introduced s_groups is to reduce the number of connections a node has, and reduce the size of name spaces, i.e. replication of information to a smaller number of nodes [REL12]. Before introducing grouping nodes in s_groups we had considered the following approaches: distributed hash tables, hierarchical structure, partitioning, and overlapping. When deciding on the approach we followed the following principles.

- Present 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.

Grouping nodes according to their hash values is a dynamic approach, but it would contradict the Erlang philosophy that states that any node can be connected to any other node. It would also become complicated for a node to belong to multiple groups, and node leaving an s_group would mean changing of its hash value. A hierarchical approach also prevents a node to be a member of different groups and to have direct connections between the nodes. Therefore, we decided to implement overlapping s_groups as they seem to satisfy the Erlang philosophy and our goals the best. Furthermore, using overlapping s_groups all the above structures can be implemented.

## 3.2 Overview

S_groups are implemented on the basis of global_groups of Erlang/OTP R15B03. Modifications are made in the following files:

- `lib/kernel/src/global.erl`
- `lib/kernel/src/global_search.erl`
- `lib/kernel/src/kernel.erl`
- `lib/kernel/src/net_kernel.erl`

In SD Erlang connections and data replication between nodes that belong to the same s_group are handled on all nodes by the following two processes: `global_name_server` and `s_group`. The processes are started at the node launch. `S_group` process is started from `s_group` module and is

| No. | ETS Table | Distributed Erlang | SD Erlang |
|---|---|---|---|
| 1 | global_names | {Name, Pid, Method, RPid, Ref} | {{SGroupName, Name}, Pid, Method, RPid, Ref} |
| 2 | global_names_ext | {Name, Pid, RegNode} | {{SGroupName, Name}, Pid, RegNode} |
| 3 | global_pid_names | {Pid, Name}<br>{Ref, Name} | {Pid, {SGroupName, Name}}<br>{Ref, {SGroupName, Name}} |

Table 2: Modifications in ETS Tables of `global_name_server` Process

responsible for keeping information about s_groups the node belongs to. `Global_name_server` process is started from `global` module, and is responsible for keeping connections and common data on the nodes identified by `s_group` process.

`Global_name_server` process keeps s_group registered names in a number of ETS tables, e.g. `global_names`, `global_pid_names`. In SD Erlang the types of all `global_name_server` ETS tables are the same as in distributed Erlang but entry $Name$ was replaced by $\{Name, SGroupName\}$ in the following ETS tables: `global_names`, `global_names_ext`, `global_pid_names` (Table 2). This was done to support overlapping of s_group namespaces. Thus, a name registered on a free node has $SGroupName = 'undefined'$. On free nodes the functionality of functions from module `global` was preserved.

In SD Erlang an s_group has the following parameters: a name, a list of nodes, and a list of registered names [REL12]. A node can be a member of a number of s_groups. When s_groups are started statically, i.e. an s_group configuration is defined at the launch of a node, the s_group configuration can be either common or individual. For example, we start eight nodes grouped in three s_groups as it is shown in Figure 2. Nodes N1, N3, N5, N6 are normal, and nodes H2, H4, H7, H8 are hidden. A common configuration contains information about all s_groups (Listing 1), and individual configuration contains information about some s_groups, e.g. s_groups the current node belongs to (Listing 2). In both cases nodes run and communicate successfully. An advantage of nodes having information about the configuration of other s_groups is that it is possible to send messages to names and find names of

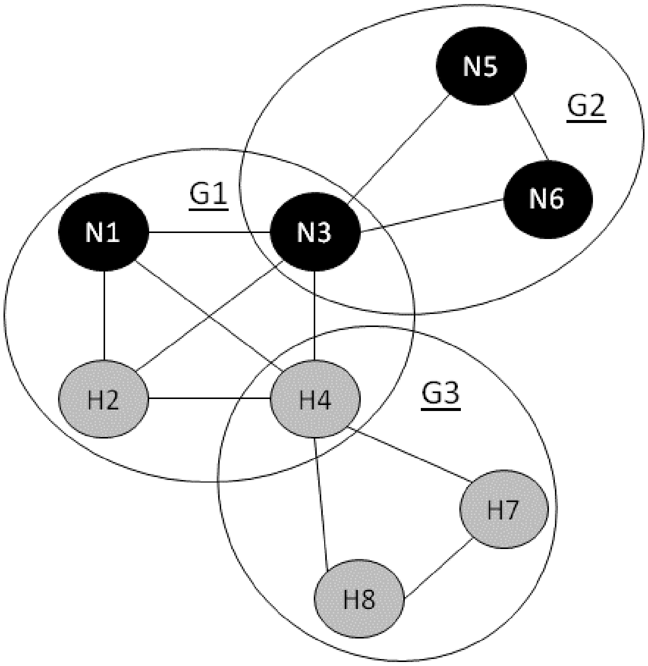


Figure 2: Connections and Namespace between Overlapping S_group Nodes

processes which are not registered in the own s_groups. A drawback is that information about remote s_groups is currently static and is not renewed in case a remote s_group is deleted or its members are changed.

**Listing 1: Common S_group Configuration – commonconf.config**

```
[{kernel,
  [{s_groups,
   [{group1, normal, [’node1@glasgow.ac.uk’, ’node2@glasgow.ac.uk’,
                      ’node3@glasgow.ac.uk’, ’node4@glasgow.ac.uk’]},
    {group2, normal, [’node3@glasgow.ac.uk’, ’node5@glasgow.ac.uk’,
                      ’node6@glasgow.ac.uk’]},
    {group3, normal, [’node4@glasgow.ac.uk’, ’node7@glasgow.ac.uk’,
                      ’node8@glasgow.ac.uk’]}]}]}].
```

**Listing 2: Individual S_group Configuration for Node H4 – individconf.config**

```
[{kernel,
  [{s_groups,
   [{group1, normal, [’node1@glasgow.ac.uk’, ’node2@glasgow.ac.uk’,
                      ’node3@glasgow.ac.uk’, ’node4@glasgow.ac.uk’]},
    {group3, normal, [’node4@glasgow.ac.uk’, ’node7@glasgow.ac.uk’,
                      ’node8@glasgow.ac.uk’]}]}]}].
```

Static s_groups are started at the launch of the nodes using flag ’-config’, e.g.

```
erl -name node1@glasgow.ac.uk -config groupconf
```

where groupconf is a .config file either from Listing 1 or 2.

A summary of the functions we discuss in Sections 3.3 and 3.4 is presented in Table 3.

## 3.3 S_group Functions

In this section we discuss s_group functions that include functions related to grouping Erlang nodes into s_groups, such as creating a new s_group, deleting an s_group, adding nodes to an s_group, removing nodes from an s_group, listing own and known s_groups, synchronisation of nodes, and providing node information. The data types of arguments in the functions are as follows [Eri13]: `Name::term()`, `Pid::pid()`, `Node::node()`, `SGroupName::group_name()`, `Reason::term()`, `Msg::term()`.

**Creating an S_group.** Function `s_group:new_s_group/2` is used to create new s_groups dynamically (Listing 3). The function creates a new s_group on the initiating node and then adds remaining nodes. In case the initiating node either is not included in the list of s_group nodes or is already a member of the defined s_group the function fails and a corresponding error is returned.

**Listing 3: New S_Group**

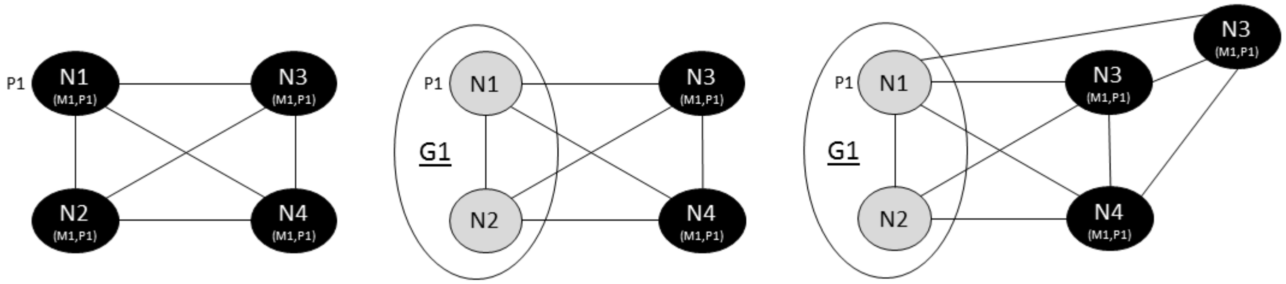
```
s_group:new_s_group(SGroupName, [Node]) -> {SGroupName, [Node]} | {’error’, Reason}
```

When a node becomes a member of an s_group the node keeps its existing connections and the global group keeps its registered names. For example, there are four interconnected free normal nodes $N1$, $N2$, $N3$, and $N4$ (Figure 3(a)). Globally registered process $P1$ with name $M1$ is on node $N1$. After becoming members of s_group $G1$ nodes $N1$ and $N2$ unregister name $M1$ (Figure 3(b)), but the remaining free nodes $N3$ and $N4$ keep the name and share the connection to node $N2$ with other free

| **global:** | **s_group:** |
|---|---|
| S_group Functions | |
| | `new_s_group(SGroupName, [Node])`<br>Creates a new s_group |
| | `delete_s_group(SGroupName)`<br>Deletes an s_group |
| | `add_nodes(SGroupName, [Node])`<br>Adds nodes to an s_group. |
| | `remove_nodes(SGroupName, [Node])`<br>Removes nodes from an s_group. |
| | `s_groups()`<br>Returns a list of all s_groups known to the node |
| | `own_s_groups()`<br>Returns a list of s_group tuples of the s_groups the node belongs to |
| | `own_nodes()`<br>Returns a list of nodes the node shares namespaces with<br>`own_nodes(SGroupName)`<br>Returns a list of nodes from the given s_group |
| `sync()`<br>Synchronises connected free normal nodes | `sync()`<br>Synchronises s_group nodes |
| `info()`<br>Returns global state information | `info()`<br>Returns s_group state information |
| Registered Name Functions | |
| `register_name(Name, Pid)`<br>Registers a name on the connected free normal nodes | `register_name(SGroupName, Name, Pid)`<br>Registers a name in the given s_group |
| `re_register_name(Name, Pid)`<br>Re-registers a name on the connected free normal nodes | `re_register_name(SGroupName, Name, Pid)`<br>Re-registers a name in the given s_group |
| `unregister_name(Name)`<br>Unregisters a name on the connected free normal nodes | `unregister_name(SGroupName, Name)`<br>Unregisters a name in the given s_group |
| `registered_names()`<br>Returns a list of all registered names on the node | `registered_names(node, Node)`<br>Returns a list of all registered names on the given node<br>`registered_names(s_group, SGroupName)`<br>Returns a list of registered names in the given s_group |
| `whereis_name(Name)`<br>Returns the pid of a name registered on a free node | `whereis_name(SGroupName, Name)`<br>Returns the pid of a name registered in the given s_group<br>`whereis_name(Node, SGroupName, Name)`<br>Returns the pid of a name registered in the given s_group. The name is searched on the given node |
| `send(Name, Msg)`<br>Sends a message to a name registered on a free node | `send(SGroupName, Name, Msg)`<br>Sends a message to a name registered in the given s_group<br>`send(Node, SGroupName, Name, Msg)`<br>Sends a message to a name registered in the given s_group. The name is searched on the given node |

Table 3: Summary of the New and Modified Functions from Sections 3.3 and 3.4

nodes until process $P1$ is registered and alive. Thus, new free normal node $N5$ after connecting to nodes $N3$ and $N4$ and getting globally registered names is also connected to node $N1$ (Figure 3(c)). However, this connection to node $N1$ is not transitive, i.e. node $N1$ does not share its connection to node $N2$ with node $N5$. If a node has no globally registered processes then after it becomes a member of an s_group free nodes do not share a connection to it with new free nodes, e.g. node $N2$ has no globally



(a) Free Normal Nodes (b) Adding an S_group (c) Adding a Free Normal Node

Figure 3: Connections when Creating a New S_group

registered processes (Figure 3(b)), therefore, free nodes $N3$ and $N4$ do not share the connection to node $N2$ with new free node $N5$ (Figure 3(c)).

**Deleting an S_group.** Function `s_group:delete_s_group/1` is used to dynamically delete an existing s_group (Listing 4). The function is similar to `s_group:remove_nodes/2` function when all nodes are removed from the s_group. A node can only delete s_groups it is a member of.

**Listing 4: Deleting an S_group**

```
s_group:delete_s_group(SGroupName) -> 'ok'
```

**Adding Nodes to an S_group.** Function `s_group:add_nodes/2` is used to dynamically add nodes to an existing s_group (Listing 5). In case the current node does not belong to the given s_group an error is returned. The function is similar to `s_group:new_s_group/2` function.

**Listing 5: Adding Nodes to an S_group**

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

**Removing Nodes from an S_group.** Function `s_group:remove_nodes/2` is used to dynamically remove nodes from an existing s_group (Listing 6). The initiating node should be a member of the given s_group, and a node cannot remove itself from an s_ group.

**Listing 6: 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_groups it becomes free. The free node type, i.e. 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 existing connections the node synchronises with other free normal nodes with which it has connections and shares their name space. In case the node has a process registered in the s_group it left, other new s_group members will be also connected to the node while the process is registered and alive.

**Listing S_groups.** Function `s_group:s_groups/0` is used to list s_groups the node is aware of (Listing 7). When called on an s_group node the function returns two lists: a list of s_groups the current node belongs to and a list of other known s_groups. When the function is called on a free node 'undefined' is returned.

**Listing 7: List of Own and Known S_Groups**

```
s_group:s_groups() -> {[OwnSGroupName], [OtherSGroupName]} | 'undefined'
```

**Listing Own S_groups.** Function `s_group:own_s_groups/0` is used to list the node's own s_groups together with their nodes (Listing 8). On an s_group node the function returns a list of *SGroupTuples*, i.e. an s_group name together with a list of nodes from that s_group. On a free node the function returns an empty list.

**Listing 8: List of Own S_groups with Nodes**

```
s_group:own_s_groups() -> [{SGroupName, [Node]}]
```

**Listing Own Nodes.** Functions `s_group:own_nodes/1,2` are used to list nodes with which the current node shares namespaces (Listing 9). On an s_group node function `s_group:own_nodes()` returns nodes from all s_groups the current node belongs to including the current node. On a free node the function returns a list of nodes with which the current node shares a namespace.

**Listing 9: List of Own Nodes**

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

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

**Synchronisation.** To synchronise nodes and update name spaces `global:sync/0` and `s_group:sync/0` functions are used (Listing 10). On a free node function `global:sync()` synchronises the node with all other known free nodes, and on an s_group node the function returns an error. On an s_group node function `s_group:sync()` synchronises the node with all s_group nodes the current node belongs to, and on a free node no synchronisation occurs.

**Listing 10: List of Own S_groups with Nodes**

```
global:sync() -> 'ok' | {'error', Reason}
s_group:sync() -> 'ok' | {'error', Reason}
```

**Node Information.** Functions `global:info/0` and `s_group:info/0` provide node state information. The functions work on both s_group and free nodes.

### 3.4 Registered Name Functions

In this section we discuss registered name functions that include functions related to manipulating registered names, such as name registration, re-registration, and unregistration, listing registered names, search for registered names, and sending messages to names.

**Name Registration.** A name can be registered using `register_name` functions presented in Listing 11. 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)`. A node can only register a name in a group the node belongs to. Therefore, when a free node attempts to register a name using `s_group:register_name(SGroupName, Name, Pid)` the function returns 'no' because a free node belongs to no s_group; similarly, when an s_group attempts to register a name using `global:register_name(Name, Pid)` the function also returns 'no'.

**Listing 11: Name Registration**

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

When registering a name `register_name` function first checks whether the node belongs to the defined group. If so the function checks whether *Name* and *Pid* are already registered in that group. In case the name and pid are new in that group the name is registered.

**Name Re-registration.** The purpose of `re_register_name` function is to register a pid using the name that is already taken for a different pid in the defined group. The name re-registration works similarly to the name registration. A name can be re-registered using `re_register_name` functions presented in Listing 12. A node can only re-register a name in the group the node belongs to. Therefore, when a free node attempts to re-register a name using `s_group:register_name(SGroupName, Name, Pid)` the function returns 'no' because a free node belongs to no s_group; similarly, when an s_group attempts to re-register a name using `global:register_name(Name, Pid)` the function also returns 'no'.

**Listing 12: Name Re-registration**

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

The function first checks whether the node belongs to the given group. In case the response is positive and the pid is not registered in the group the name is re-registered. In case the pid is already registered under a different name the re-registration fails.

**Name Unregistration.** A name can be unregistered from a group using `unregister_name` functions presented in Listing 13.

**Listing 13: Name Unregistration**

```
global:unregister_name(Name) -> 'ok'
s_group:unregister_name(SGroupName, Name) -> 'ok'
```

The function first checks whether the node belongs to the given group. If so the name is unregistered from the group. If the name is not registered in the group then 'ok' is returned.

**Listing Registered Names.** A list of registered names can be accessed by calling `registered_names` functions presented in Listing 14. Function `global:registered_names()` works on both s_group and free nodes, and returns all names registered on the node, i.e. on a free node the function returns a list of globally registered names, and on an s_group node the function returns a list of names from all s_groups the node belongs to. Function `s_group:registered_names({node, Node})` also works on both s_group and free nodes. It works similarly to `global:registered_names()` but returns all registered names from the target node *Node*. In case the current node is not connected to node *Node* a new connection is established.

**Listing 14: List of Registered Names**

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

Function `s_group:registered_names({s_group, SGroupName})` returns a list of registered names in s_group *SGroupName*. The function returns registered names only from s_groups that the current node is aware of. In case the current node is not a member of s_group *SGroupName* the node establishes a connection with one of the nodes of s_group *SGroupName*. In case the node is not aware of s_group *SGroupName* an empty list is returned.

**Searching for a Name.** A registered name can be found using `whereis_name` functions presented in Listing 15. The name search is done sequentially, and as soon as the name is found its pid is returned. The functions first check name *Name* in the node own registry. If the name is not found locally then the name is searched in other known s_groups by picking a node from the defined s_group *SGroupName*, then establishing a connection with that node, and then checking whether name `{SGroupName, Name}` is registered on that node.

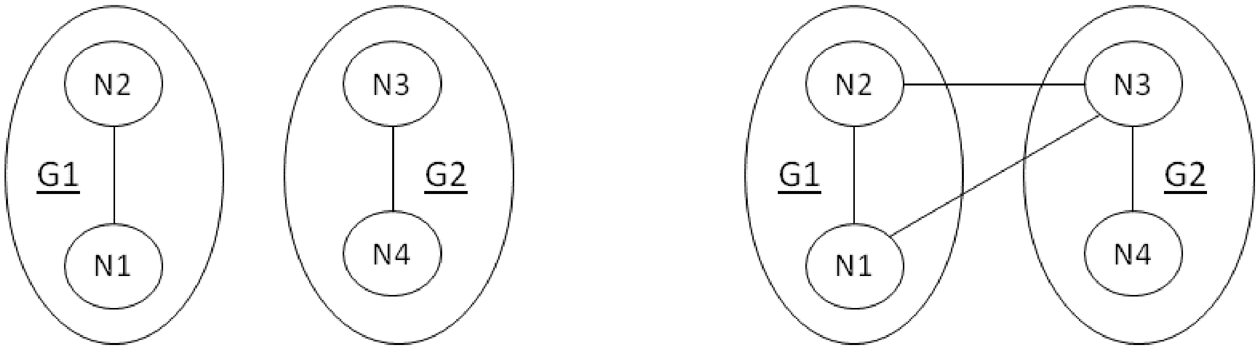
**Listing 15: Search of a Registered Name**

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

Function `global:whereis_name(Name)` returns a pid on a free node in case the name is found, otherwise returns ’undefined’. Function `s_group:whereis_name(SGroupName, Name)` returns a pid on an s_group node if the name is registered in the given *SGroupName* and the node is aware of that s_group; on a free node the function returns ’undefined’.

Function `s_group:whereis_name(Node, SGroupName, Name)` works on both s_group and free nodes and searches only `{SGroupName, Name}` registered on that node. In case the target node is free *SGroupName* should be ’undefined’. If initiating node $N1$ and target node $N2$ are not connected, then the connection is established. In case the process actually resides on node $N3$ no connection between nodes $N1$ and $N3$ is established.

**Sending a Message.** A message can be sent to a registered name using `send` functions presented in Listing 16. From a free node a message can be sent to name *Name* using `global:send(Name, Msg)`. From an s_group node that has information about s_group *SGroupName* a message can be sent to any registered *Name* in that s_group using `s_group:send(SGroupName, Name, Msg)`. To send a message to $\{SGroupName, Name\}$ registered on node *Node* function `s_group:send(Node, SGroupName, Name, Msg)` is used. In case node *Node* is free a message to *Name* registered on that



(a) Before

(b) After

Figure 4: Connections when Sending a Message

node should be sent for an undefined s_group, i.e. `s_group:send(Node, 'undefined', Name, Msg).`

**Listing 16: Sending Messages**

```
global:send(Name, Msg) -> Pid
s_group:send(SGroupName, Name, Msg) -> Pid | {'badarg', Reason}
s_group:send(Node, SGroupName, Name, Msg) -> Pid | {'badarg', Reason}
```

When sending a message to an s_group or a node from a remote node a number of connections can be established. For example, we have two s_groups: $\{G1, [N1, N2]\}$ and $\{G2, [N3, N4]\}$ (Figure 4(a)). Assume that we send a message from node $N3$ to name $\{Name, G1\}$ on node $N2$, i.e.
`s_group:send(N2, G1, Name, Msg).`
If $Name$ registered in s_group G1 actually resides on node $N1$ then node $N3$ also establishes a connection with node $N1$ (Figure 4(b)). Thus, when sending a message to a name on a node or in a remote s_group the initial node establishes a connection with both the defined node and with the node where the process actually resides.

The difference between functions `s_group:send/3` and `s_group:send/4` is in the node that should maintain information about the target s_group. In case of `s_group:send/3` this is the initiating node, and in case of `s_group:send/4` this is the target node.

# 4 Semi-explicit Placement

We implement semi-explicit placement with function choose_nodes/1 presented in Listing 17. The function returns a list of nodes that satisfy all given criteria, i.e. intersection. In case no node satisfies the criteria the function returns an empty list. The initial parameters include s_groups and attributes.

Currently, only nodes from s_groups the initiating node belongs to are considered, i.e. when a parameter is {`s_group, SGroupName`} in case the initiating node belongs to s_group $SGroupName$ the function returns a list of the s_group nodes, otherwise an empty list is returned. When the parameter is {`attribute, AttributeName`} the initiating node collects attributes from all connected nodes, then returns a list of nodes that contain the given attribute.

Attributes are a set of individual node characteristics. Attributes may contain information about hardware and software specification, or a particular node responsibility. We have implemented attributes by adding a corresponding parameter to a node global state. At the node launch the list of attributes is empty. To manipulate node attributes we have implemented the following functions: adding, deleting, and viewing attributes.

**Listing 17: Selecting Nodes using Parameters**

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

**Adding attributes.** Attributes can be added using functions presented in Listing 18. Function `global:add_attribute/1` adds a list of attributes to the own node, and function `s_group:add_attribute/2` adds a list of attributes to a set of nodes.

**Listing 18: Adding Attributes**

```
global:add_attribute([AttributeName]) -> 'ok' | {error, Reason}
s_group:add_attribute([Node], [AttributeName]) -> 'ok' | {error, Reason}
```

**Removing attributes.** To remove attributes functions presented Listing 19 are used. Function `global:remove_attribute/1` removes attributes from the own node, and function `s_group:remove_attribute/2` removes attributes from a list of nodes.

**Listing 19: Removing Attributes**

```
global:remove_attribute([AttributeName]) -> 'ok'
s_group:remove_attribute([Node], [AttributeName]) -> 'ok'
```

**Listing attributes.** To view own node attributes function `global:registered_attributes()` is used (Listing 20).

**Listing 20: Listing Attributes**

```
global:registered_attributes() -> [AttributeName]
```

# 5 Preliminary Validation

The functions have been unit tested. The node state, connections, and name spaces were analysed by combining the following parameters: static & dynamic s_groups, partitioned & overlapping s_groups, common & individual s_group configuration on nodes. Individual function properties, such 'a node can register a name only in an s_group the node belongs to', discussed in [REL12] were also tested. Table 4 provides a list of types of nodes we considered in the unit tests for the functions discussed in Sections 3.3 and 3.4. We also use a feedback from the SD Erlang semantics that we currently work on to modify and improve the functions.

We work on demonstrators to investigate performance benefits of SD Erlang compared to distributed Erlang. In the preliminary performance analysis we use DEbench as a benchmarking tool to measure

| **Functions** | **Types of Nodes** |
|---|---|
| `all` | Free or s_group nodes<br>1. Free node<br>2. S_group node<br>(a) Belongs to a single s_group<br>(b) Belongs to multiple s_groups<br>Normal or hidden nodes<br>1. Normal node<br>2. Hidden node |
| `new_s_group/2`<br>`add_nodes/2` | Connected or not connected to other nodes<br>1. Connected to other nodes<br>(a) Shares a name space with the nodes<br>i. After joining the new s_group continues to share a namespace with the nodes<br>ii. After joining the new s_group does not share a namespace with the nodes<br>(b) Does not share a name space with the nodes<br>2. Not connected to other nodes |
| `delete_s_group/1`<br>`remove_nodes/2` | Becomes free or remains an s_group node after leaving an s_group<br>1. Becomes a free node<br>(a) Connections with other free nodes<br>i. Connected to free nodes<br>ii. Not connected to free nodes<br>(b) Nodes from the same s_group<br>i. No other nodes become free<br>ii. Other nodes also become free<br>2. Remains an s_group node |

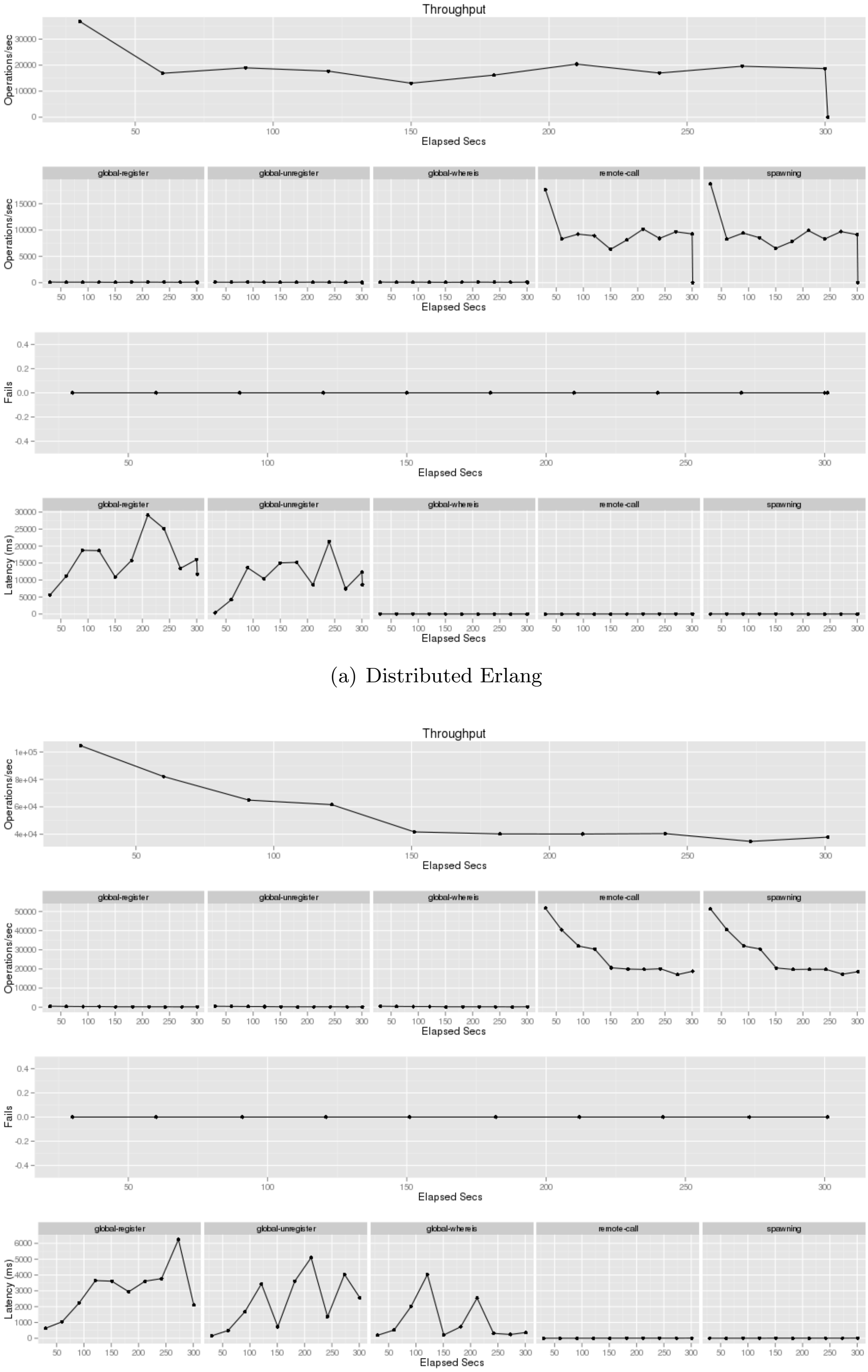
Table 4: Types of Nodes Considered in the Unit Tests

the throughput and the latency of distributed Erlang commands and SD Erlang commands on a cluster. DEbench is a simplified version of open source Basho Bench benchmarking tool [REL13]. Basho Bench was created to conduct performance and stress tests, and to produce performance graphs [Bas13]. All nodes participating in the experiments run their own copy of DEbench. An example of conducted experiments using DEbench is presented below.

In the experiment we measure throughput and latency of the following P2P and global commands.

- P2P commands: spawn and RPC.
- Global commands: registration, unregistration, and search of global names.

The rate with which the commands are called we define in a configuration file, e.g. one global command to a hundred of P2P commands. Each node randomly selects a node and a command from the configuration file and runs that command on the selected node. The experiment is run on 20 nodes for 5 minutes. In SD Erlang we partition the nodes in 5 s_groups, and all s_groups have 4 nodes. Thus, when registering a name in the experiments with distribute Erlang the names are replicated on



(a) Distributed Erlang

(b) SD Erlang

Figure 5: Throughput of 20 Nodes in DEbench Experiments

20 nodes, and in the experiments with SD Erlang the names are replicated on 4 nodes. In commands spawn and RPC the size of an argument is 1000 bytes, and the called function takes $200\mu s$.

Figures 5(a) and 5(b) show the throughput of a 20-node Elang cluster in experiments using distributed Erlang and SD Erlang respectively. The experiments show that the throughput in SD Erlang experiments is larger in comparison with distributed Erlang experiments, i.e. 500,000 operations/sec vs. 200,000 operations/sec; whereas the latency of global operations in SD Erlang experiments is smaller that in the distributed Erlang experiments, i.e. $3{,}000\mu s$ vs. $15{,}000\mu s$. Therefore, in comparison with distributed Erlang SD Erlang reduces the latency of global operations and increases the throughput.

In the next year we plan to use s_groups in the WP6 case studies.

# 6 Implications and Future Work

The deliverable presents the implementation of SD Erlang computation model and semi-explicit placement. We have discussed the main aspects of s_group implementation and covered functionality of the following sixteen functions from `s_group` and `global` modules: creating and deleting an s_group, adding and removing nodes from an s_group, listing own and known s_groups, synchronisation and monitoring of nodes, providing node information, name registration, re-registration, and unregistration, listing registered names, search of registered names, and sending messages. For semi-explicit placement we have implemented `choose_nodes/1` function and node attributes together with five additional function to manipulate those attributes. All functions were unit tested.

To guarantee the uniqueness of s_group names we plan to introduce s_group leader nodes in D3.3 Scalable SD Erlang Reliability model deliverable, and hence we no longer need to maintain global updates of s_group information [REL12, p.11,13].

We plan to build the following on the s_group implementation presented here.

1. *Reliability Model* is scheduled in deliverable D3.3. It includes mechanisms to ensure uniqueness of s_group names when no central information about s_groups is collected. One of the direction proposed in D3.1 is introducing s_group leader nodes. The reliability model also includes restarting nodes in their s_groups.

2. *Semi-explicit Placement.* We plan to add more parameters to `choose_node/1` function as discussed in D3.1 The Design of Scalable Distributed Erlang deliverable. Currently, we consider adding node's load and communication distance parameters, i.e. collecting load information from connected nodes to place processes on the least loaded nodes, and using communication distances to decide how far we want to spawn a process from the initiating node. We may also consider introducing a scalable scheme to collect state information from remote s_groups. This will enable a node to consider placing processes not only on the nodes from its own s_group but also on nodes from remote s_groups.

3. *SD Erlang Semantics.* Together with the Kent team of the RELEASE project we work on the semantics of the basic SD Erlang functions, such as `register_name/3` and `new_s_group/2`.

## Change Log

| Version | Date | Comments |
|---|---|---|
| 0.1 | 15/3/2013 | First Version Submitted to Internal Reviewers |
| 0.2 | 29/3/2013 | Revised version based on comments from T. Hoffmann and R. Aloi submitted to the Commission Services |
| 1.0 | 10/4/2013 | Final version submitted to the Commission Services |
| 1.1 | 4/7/2013 | Version Submitted to Internal Reviewers |
| 1.2 | 8/8/2013 | Revised version based on comments from E. Fernandez submitted to the Commission Services |
| 2.0 | 20/8/2013 | Final version submitted to the Commission Services |

## References

[Bas13] Basho Technologies. Basho Bench, 2013. http://docs.basho.com/riak/latest/ops/building/benchmarking/.

[Eri13] Ericsson AB. Types and Function Specifications, 2013. http://www.erlang.org/doc/reference_manual/typespec.html.

[REL12] RELEASE Project. Deliverable D3.1: Scalable Reliable SD Erlang Design, June 2012.

[REL13] RELEASE Project. DEbench, 2013. https://github.com/release-project/RELEASE/Research/Benchmarks/DEbench.