
ICT-287510

RELEASE

A High-Level Paradigm for Reliable Large-Scale Server Software

A Specific Targeted Research Project (STReP)

# D5.4 (WP 5): Interactive SD Erlang Debugging

Due date of deliverable: 31 January 2015

Actual submission date: 28 February 2015

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

Lead Contractor: University of Glasgow Revision: 0.1

**Purpose:** This deliverable provides functionality to support debugging and monitoring systems written in SD Erlang, complementing the tools already provided in Erlang/OTP and in earlier RELEASE deliverables.

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

- We have developed a new tool, SD-Mon, designed specifically to monitor SD-Erlang systems. (Tasks 5.6, 5.7)
- We have made a thorough revision of the Devo tool, so that it now provides a more scalable visualisation of SD-Erlang systems. (Task 5.7)
- We have provided patches for the Erlang/OTP distribution to deliver greater tracing scalability. (WP5).

**Conclusion:** We have augmented Erlang with tools – Percept2, Devo and SD-Mon – designed to support the development, monitoring and debugging the 'Scalable Distributed' aspects of SD Erlang systems.



| 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) | |

# Contents

# Interactive SD Erlang Debugging

Maurizio Di Stefano, Simon Thompson and Stephen Adams, University of Kent

# 1 Executive Summary

The standard Erlang/OTP distribution comes "out of the box" with a range of tools for tracing, profiling and debugging Erlang systems. These systems are loosely federated, and usable together from the Unix command line and Erlang shell. Indeed many are based on the Erlang built-in tracing system (and more recently on the OS-level tracing tools DTrace and SystemTap).

In order for the deliverable to add value to this set of tools, the RELEASE project has chosen not to replicate this functionality, but instead to develop tooling that complements what already exists and is already used. In particular, we have added functionality in three ways.

- A new tool, SD-Mon, a tool designed specifically to monitor SD-Erlang systems; this purpose is accomplished by means a "shadow" network of agents, mapped on the running system. The shadow network follows system changes so that agents are started and stopped at runtime according to the evolving structure of the system. SD-Mon can be used to understand the correct partition of nodes in groups. At an initial stage, just by looking at the inter-node messages flow, it can drive the initial partition itself. After that it can be used to trim the network architecture and to monitor the system, revealing anomalies like intergroup messages bypassing the "bridge" nodes. (Tasks 5.6, 5.7)

- A thorough revision of the Devo tool for online monitoring of SD-Erlang systems, so that it now provides a more scalable visualisation of SD-Erlang systems, as well as a substantial refactoring of the overall architecture of the tool, making it more adaptable of use in conjunction with other tools, especially SD-Mon. (Task 5.7)

- Patches for the Erlang/OTP distribution to deliver greater tracing scalability. (WP5).

In this deliverable we outline the tooling context for the deliverable before describing SD-Mon, the changes to Devo and finally the Erlang/OTP patches.

# 2 Introduction

Deliverable 5.4 concerns two tasks from Work Package 5.

***Task 5.6 Debugging tool***
*Task 5.1 will have delivered the infrastructure to making post-mortem observations on systems. This task will be to integrate these results as a post mortem debugging tool into existing Erlang debugging infrastructure.*

***Task 5.7 Interactive debugging tool***
*Integration of information gathered in D5.2 into the Erlang debugging infrastructure will provide a tool for the interactive debugging of systems.*

## Partner Contributions.

The principal effort for this deliverable is from the University of Kent. Staff from Erlang Solutions Ltd gave feedback on the SD-Mon implementation and on the presentation of the results in this deliverable.

# 3 The tooling landscape

Deliverable D5.1 gives an extensive overview of the tools available for tracing, profiling and debugging Erlang systems (as of mid 2012). We summarise that and bring it up to date in this section of the deliverable, thus providing the context in which the work reported here has been done.

## 3.1 Erlang built-in tracing

The Erlang runtime system has built-in support for tracing many types of events, and this infrastructure forms the basis of a number of tools which build on or specialise the services offered by the Erlang virtual machine, through a number of built-in functions (or BIFs).

With the built-in tracing, an Erlang program can be traced while being executed, and no special compilation or instrumentation of the program is needed. Events that can be traced include: global and local function calls, process-related activities, message passing, garbage collection and memory usage.

However powerful it is, Erlang built-in tracing has some limitations. First, tracing adds a considerable overhead; this situation can be mitigated to some extend on linux/unix systems by means of OS-level tracing facilities such as SystemTap / DTrace, and such a back end has been provided for the Percept2 tool delivered in D5.1/5.2. Second, because a traced process can only have one tracer process at any one time, it is impossible to have several tools requiring trace information run concurrently on a node; this remains a problem. Two other limitations have been tackled by RELEASE.

- It is possible to use match specification to have fine-grained control over the particular function call or return trace events are generated. This is a much more efficient mechanism that generating a more coarse-grained set of trace events, only to have them filtered post hoc. In this project we have extended this filtering process - see Section 6 for more details.

- Out of the box there is no support for remote or distributed tracing, that is all settings have to be executed on the node and the trace process has to be node local; the SD-Mon tool reported in this deliverable (Section 4) allows tracing to be made across an SD Erlang system, automatically.

## 3.2 Erlang tracing tools

The Erlang built-in functions for tracing are powerful, but they are also very low-level and are not very user-friendly. In practice, a sequence of function calls are needed to set up an interesting trace. Apart from that, there are no ready-to use BIFs from Erlang for the display and analysis of trace results. Unsurprisingly, a collection of tracing/profiling tools have been built on top of Erlang's built-in trace, and most of these tools are part of the standard Erlang distribution. These include the Observer, The DBG tracer, ET, the Event Tracer; TTB, The Trace Tool Builder and ETop, The Erlang Top. These are described in more detail in D5.1/5.2, and in Section 3.5 below.

## 3.3 Erlang profiling tools

Erlang/OTP contains a number of server profiling tools for finding performance bottlenecks in Erlang programs, including fprof (function profiling based on tracing, and so quite “heavyweight”), eprof (process profiling based on the erlang:trace_info/3 BIF, more “lightweight‘ than fprof), cprof (counts number of function calls; “lightweight”), lcnt (the Lock Profiler) and Percept, which gives a tracing-based overview of

concurrency in Erlang; Percept has been enhanced to Percept2 by the RELEASE project (see next subsection).

## 3.4 Percept2

Percept has been enhanced to Percept2 by the RELEASE project, as reported in D5.1 and D5.2. These enhancements include:

- User-command interface improved to allow the profiling of a particular aspect of the execution.
- Distinguish between running and runnable time for each process.
- Selective function profiling of processes.
- Improved dynamic function call graph.
- Process communication graph.
- Inter-node message passing.
- Tracing of s_group activities in a distributed system.

The system is available from https://github.com/RefactoringTools/percept2

## 3.5 Observer

The process manager Pman was a tool for viewing processes executing locally or on remote nodes. Its main purpose was to locate erroneous code by inspecting the state of the processes and by tracing events. Processes could be inspected individually in a process trace window, in which a user could see trace output for sent and received messages as well as for called functions and some other process events. It was noted that Pman had some effect on the real time behaviour of a running system. The Pman application was superseded by the Observer application, and Pman was removed from OTP in R16B, February 2013.

Observer is a graphical tool for observing the characteristics of Erlang systems. Observer displays system information, application supervisor trees, process information, ets or mnesia tables and contains a frontend for Erlang tracing (cf dbg and other interfaces to built-in Erlang tracing). Observer can be used on multi-node systems.

## 3.6 Debugging infrastructure

The Erlang system has an integrated debugging facility, the debugger application (not to be confused with the dbg tracing tool!) The debugger is a graphical user interface for the Erlang interpreter, which can be used for debugging and testing of Erlang programs. For example, breakpoints can be set, code can be single stepped and variable values can be displayed and changed. The Erlang interpreter can also be accessed programmatically via the interface module int.

More details about the debugger application and int module are given in the online Erlang documentation at:

http://www.erlang.org/doc/apps/debugger/debugger_chapter.html

http://www.erlang.org/doc/man/int.html

## 3.7 Wombat

Erlang Solutions – as a part of the RELEASE project – have implemented WombatOAM as a commercial product. [Quotes from the WombatOAM website.]

*"WombatOAM is an operations and maintenance framework for Erlang based systems. It gives … full visibility on what is going on in … Erlang clusters either as a stand-alone product or by integrating into … existing OAM infrastructure."*

*"WombatOAM is designed to help developers and operators administer and monitor clusters of Erlang nodes in heterogeneous public and private clouds. Using hidden nodes and distributed Erlang, WombatOAM connects to your Erlang cluster. WombatOAM collects and stores metrics, logs and notable events from the managed Erlang nodes. Its Web Dashboard displays this data in an aggregated manner and provides interfaces to feed the data to other OAM tools."*

Wombat is a commercial product of Erlang Solutions Ltd.

# 4 SD-Mon

This section describes the SD-Mon tool which provides the scalable infrastructure to support offline and online monitoring of SD-Erlang systems through a “shadow network” of nodes designed to collect, analyse and transmit monitoring data from active nodes and s_groups.

## 4.1 Introduction

SD-Mon is a tool designed to monitor SD-Erlang systems.

This purpose is accomplished by means a shadow network of agents, mapped onto the running system. The network is deployed on the base of a configuration file describing the network architecture in terms of hosts, Erlang nodes, global group and s_group partitions. SD-Mon can be used to understand the correct partition of nodes in groups. At an initial stage, just by looking at the inter-node messages flow, it can drive the initial partition itself. After that it can be used to trim the network architecture and to monitor the system, revealing anomalies like intergroup messages bypassing the "bridge" nodes.

The tracing to be performed on monitored nodes is also specified within the configuration file. An agent is started by a master SD-Mon node for each s_group and for each free node. Configured tracing is applied on every monitored node, and traces are stored in binary format in the agent file system.

The shadow network follows system changes so that agents are started and stopped at runtime according to the needs. Such changes are persistently stored so that the last configuration can be reproduced after a restart. Of course the shadow network can be always updated via the User Interface.

As soon as an agent is stopped the related tracing files are fetched across the network by the master and they are made available in a readable format in the master file system.

## 4.2 Overall Architecture

SD-Mon is an Erlang application based on a master-agent architecture.

Currently sub-masters are not supported so each agent is directly linked to the master.

Agents run as Erlang hidden nodes in order not to propagate node connections across the network.

### 4.2.1 Master

The Master is a gen_server started and supervised by a supervisor process at application startup.

The shadow network is deployed according to a specified configuration. In order to limit network usage the host running the maximum number of Erlang nodes is chosen to run the agent as well. Once the agents are started they ask for configuration data, consisting of:

- nodes to be monitored
- traces to be run on them.

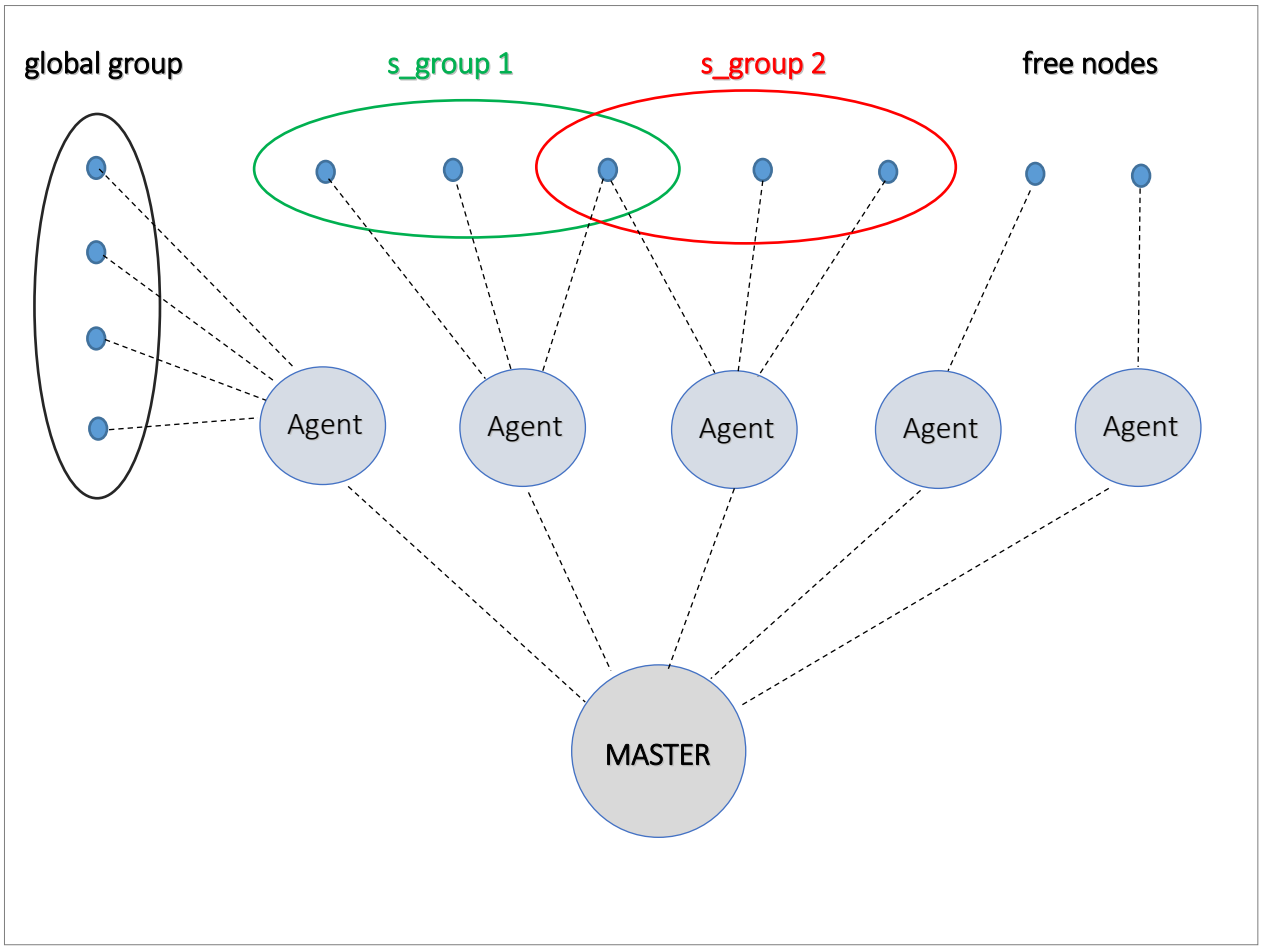



**Figure 1: SD-Mon shadow network**

Each network change in the s_group structure is cached by agents and notified to the master, which is the only one having a global network view. It takes care to restructure the shadow network accordingly: for instance if a new s_group is created a new agent is started, if an s_group is deleted the related agent is stopped, the tracing files are gathered from its host, and new agents are started for its nodes not controlled by any other agent.

Moreover, changes are dumped in new configuration files (the originals are saved) in order to allow the tool to be able to access the last known configuration in case of restart.

User can start and stop the sdmon application, start and stop single agents and ask for the master status.

Trace files related to an agent, controlling a group or a free node, are gathered each time an agent is closed at runtime or when the whole application is closed. Binary files are moved at the master host under the `BASEDIR/SD-Mon/traces/` directory and decoded on the fly, from now on being `BASEDIR` the base directory where the tool is installed. A text file is produced for each controlled node and a summary of statistics is created for each agent and at global system level.

System events are logged by the master under `BASEDIR/SD-Mon/logs`.

### 4.2.2 Agents

Each agent takes care of an s_group or of a free node. At startup it tries to get in contact with its nodes and apply the tracing to them as stated by the master. Binary files are stored in the host file system.

Tracing is internally used in order for the system to be aware of the s_group operations – namely create or delete a s_group, add or remove nodes to a s_group – that happen at runtime. An asynchronous message is sent to the master whenever one of these changes occurs.

Since each process can only be traced by a single process at the time, each node (included those belonging to more than one s_group) is controlled by only one agent.

When a node is removed from a group or when the group is deleted, another agent takes over, as shown in Figure 2.

When an agent is stopped, all traces on the controlled nodes are switched off.

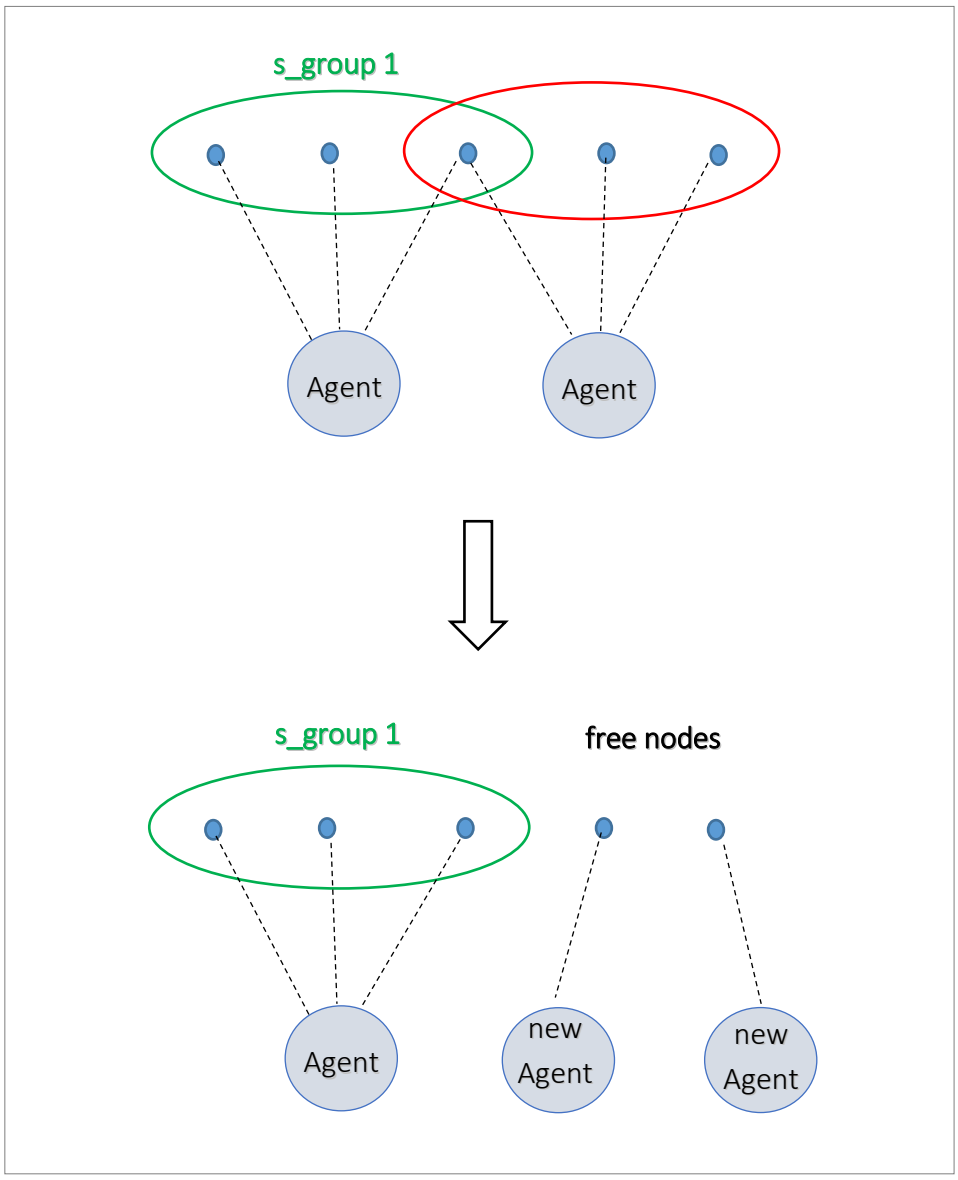



**Figure 2: Delete s_group 2**

### 4.2.3 Robustness

A double robustness mechanism is adopted against network turbulence and failures.

First of all every agent node is monitored so that as soon as a nodedown message is received a direct monitoring for that node is initiated: a dedicated process is spawned to poll the missing node and when it comes up again a nodeup notification is sent to the master who checks the agent status and align the system to any changes performed in the meantime.

This is done with the second mechanism which is based on a configuration token (see Figure 3).

A token is an integer identifying the current valid configuration: it is sent to the agent (who stores it) every time something changes in the related part of the network. After a nodeup event the master sends the current token to the agent and if it does not match, the agent is configured from scratch. This strategy ensures fast recovery from network failures.

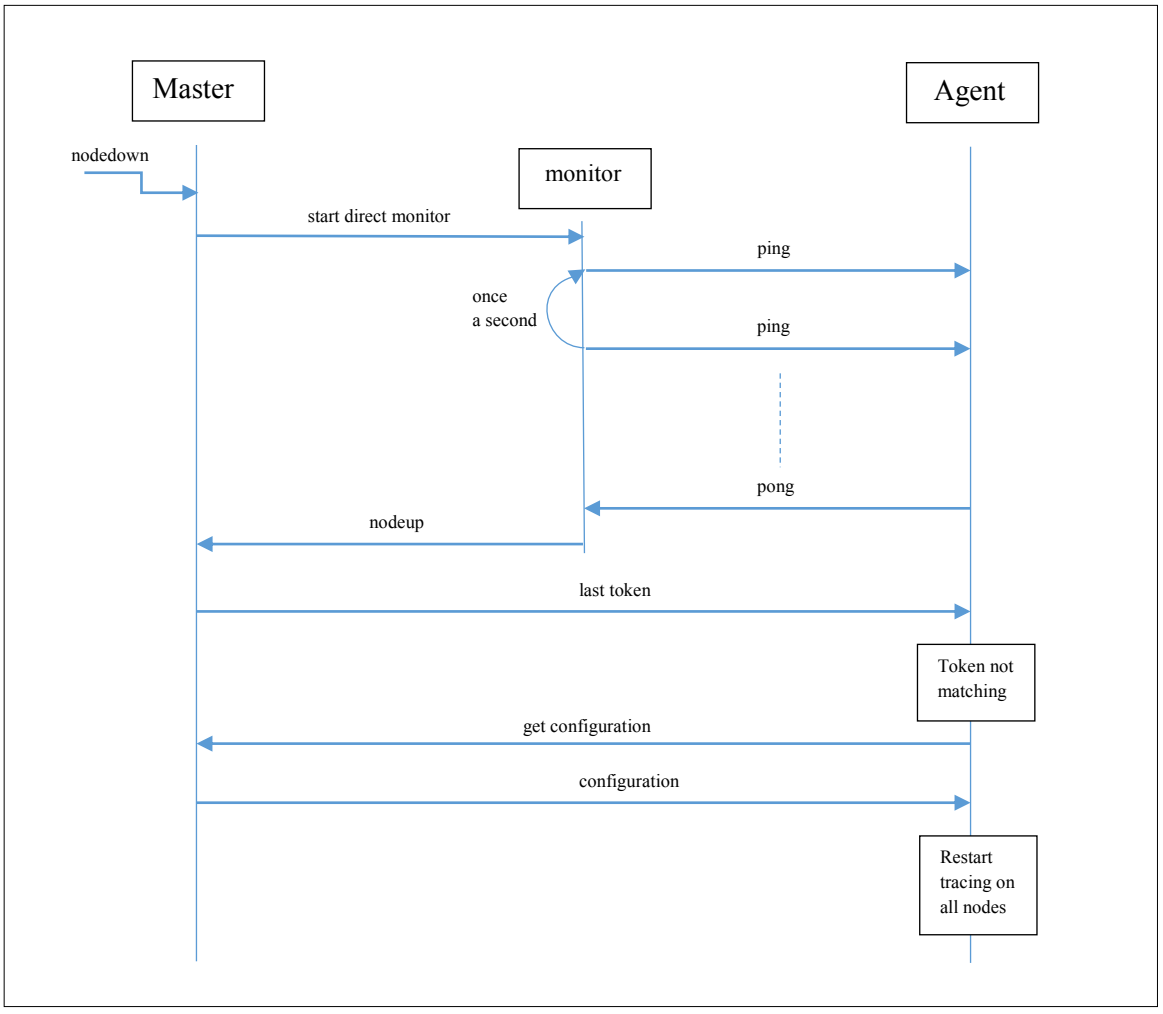



**Figure 3: nodedown use case**

Direct monitoring is also applied by the agents on the controlled nodes when a nodedown event occurs. Tracing is re-established at nodeup.

### 4.2.4 Configuration and Trace

The tool can be launched on a running system by means of 2 configuration files:

- `group.config`
- `trace.config`

The first of these files, which is also used by s_group processes, is defined as a configuration parameter of the kernel application and states the group partition of the system in terms of s_group names and related Erlang nodes.

The second one specifies the tracing to be applied on each group of nodes.

Both files are placed under `BASEDIR/SD-Mon/config/`.

These files can be written by hand or can be generated with the test command `gen_env` starting from a unique, high level configuration file named `test.config` which is placed in `BASEDIR/SD-Mon/test/config/`

This file consists of a set of Erlang terms defining the system to be observed. It is mainly composed of 5 sections:

1) First section is optional. Here you can state the IP address to be used for all Erlang nodes running on the localhost. If left empty, the address will be deduced from the operating system.
2) The second section lists the hosts on which the system is running.
3) The third section is an integer stating how many Erlang nodes (virtual machines) should be started for each host.
4) In this part groups are listed
5) The last part describes the traces to be gathered for each group

Currently three tracing options are supported but more can be added in future: inter-node, garbage collection and scheduler. We describe these in turn now.

- *inter_node*
  Enables tracing of inter-node and inter-group messages, i.e. messages sent by a process to a destination outside its own node and group.
  Note that this feature is always active since it is internally used by the tool. It is reported here for the sake of completeness.

  For each message sent by a monitored process out of its own Erlang node two entries will be reported in the related trace file:

  - The original trace message in the form:
    `{trace, FromPid, send, Msg, ToPid}`

  - The following tuple:
    `{trace_inter_node, FromNode, ToNode, MsgSize}`

  In the event that the destination node ToNode is not part of the same group of the sender node FromNode, a third entry will be reported:

- `{trace_inter_group, FromGroups, ToGroups}`

  where the last two elements are the lists of groups of which the sender and the receiving nodes are part of. The empty list is used for free nodes.

  Example of inter_group message:

```
{trace,<2922.112.0>,send,{vertex,3955,2597,45},<2918.125.0>}.
{trace_inter_node,'node4@129.12.3.211','node4@129.12.3.176',24}.
{trace_inter_group,[group3],[group2]}.
```

  *garbage_collection*
  Enables tracing on garbage collection.
  Original trace messages will be reported in tracing files (see official Erlang documentation)

- *scheduler*
  Produces information about process scheduling.
  Original trace messages will be reported in tracing files (see official Erlang documentation).

It is also possible to define Erlang commands to be executed on the monitored system through a MFA tuple.

### 4.2.5 Post-mortem analysis

When agents are stopped a new directory (whose name is a timestamp in the form `yyyymmdd_hhmmss`) is created in the master file system under the `traces\` directory.

For each closed agent a sub-directory with its name is also created and all binary files related to the agent are moved in it. Then a readable version (.txt) of them is created for each controlled node and stored in the same directory together with a summary of statistical data for that agent (file `STATISTICS.txt`).

Lastly, statistics at system level are also created and stored in `GLOBAL_STATISTICS.txt`.

The following graph gives an example of directory structure.

```
traces/
└── 20150202_141045/
    ├── GLOBAL_STATISTICS.txt
    ├── sdmon_group1@127.0.1.1/
    │   ├── sdmon_group1@127.0.1.1_traces_0_node1@127.0.1.1
    │   ├── sdmon_group1@127.0.1.1_traces_0_node1@129.12.3.176
    │   ├── sdmon_group1@127.0.1.1_traces_0_node1@129.12.3.211
    │   ├── sdmon_group1@127.0.1.1_traces_1_node1@127.0.1.1
    │   ├── …
    │   ├── sdmon_group1@127.0.1.1_traces_node1@127.0.1.1.txt
    │   ├── sdmon_group1@127.0.1.1_traces_node1@129.12.3.176.txt
    │   ├── sdmon_group1@127.0.1.1_traces_node1@129.12.3.211.txt
    │   └── STATISTICS.txt
    ├── sdmon_group2@129.12.3.176/
    │   ├── sdmon_group2@129.12.3.176_traces_0_node1@129.12.3.176
    │   ├── …
    │   └── STATISTICS.txt
    └── sdmon_group3@129.12.3.211/
        ├── sdmon_group3@129.12.3.211_traces_0_node1@129.12.3.211
        ├── …
        └── STATISTICS.txt
```

The most relevant information included in the global statistic file are organized in 4 tables:

1. Node Tab
   Represents sent inter-node messages by any node.
   The format of each entry is:

   ```
   {FromNode, [{ToNode, TotalSize, NumOfMessages} | ..]}
   ```

   where:

   - `FromNode` and `ToNode` are the sender and the receiving nodes
   - `TotalSize` is the total size in bytes of all sent messages
   - `NumOfMessages` is the total number of sent messages

2. Sent Tab
   Reports inter-node and inter-group messages sent by any node.
   The format of each entry is:

   ```
   {FromNode, {Inter_node, Inter_group, IG/IN}
   ```

   where:

   - `FromNode` is the sender node
   - `Inter-node` is the number of inter-node messages sent
   - `Inter-group` is the number of inter-group messages
   - `IG/IN` percentage of inter-group messages

3. Flow Tab
   Reports incoming and outgoing inter-node messages for any node.
   The format of each entry is:

   ```
   {Node, Incoming, Outgoing}
   ```

   where:

   - `Incoming` is the number of incoming inter-node messages
   - `Outgoing` is the number of outgoing inter-node messages

4. Bridge Flow Tab
   Bridges are nodes belonging to more than one group, which are supposed to vehicle all traffic crossing adjacent groups. Reports inter-node and inter-group messages sent by any node. The format of each entry is:

   ```
   {Node, B_Incoming, B_Outgoing}
   ```

   where:

   - `B_Incoming` is the number of incoming inter-node messages sent by nodes belonging to at least one of Node‘s groups

- `B_Outgoing` is the number of outgoing inter-node messages sent to nodes belonging to at least one of Node's groups

NOTE: in the actual Erlang implementation Sent Tab, Flow Tab and Bridge Flow Tab also includes message size or average size for each field, not shown here for the sake of clearness.

An example of global statistic data can be found in Appendix A2.

### 4.2.6 Run-time visualization

A basic function has been introduced in order to follow system evolution at run-time.

It is only related to sent inter-node messages. When inter-node messages are detected by the tracing processes, they sent a message to their own agent in the form:

```
{in, FromNode, ToNode}
```

The agent just forwards the message to a process named `sdmon_db` on the Master node.
In case the process is not started this message is lost and everything keeps going on as usual.

When, instead, the process is started (by calling `sdmon_db:start()`) it creates an ets table, which is an ordered set having the tuple `{FromNode, ToNode}` as key and associated to a counter.

After that the process enters the receiving loop, waiting for inter-node notifications from agents: every message just increases the counter for the involved nodes.

Moreover, every 0.5 second the whole table is dumped on the text file `/tmp/in_tab.txt`.
In a system composed of N nodes the table will hold Nx(N-1) elements in the worst case, represented within the file as a list of tuples `{{FromNode, ToNode}, Counter}`.

In this way it is possible to follow run-time updating for instance by means of the OS command:

```
tail -f /tmp/in_tab.txt
```

For demo purposes it is also possible to use the following commands:

| | |
|---|---|
| `watch_internode` | starts a terminal who follows the file (max. 45 entries) |
| `watch_internode2` | starts 2 terminals (max. 45 entries each). See snapshot below. |

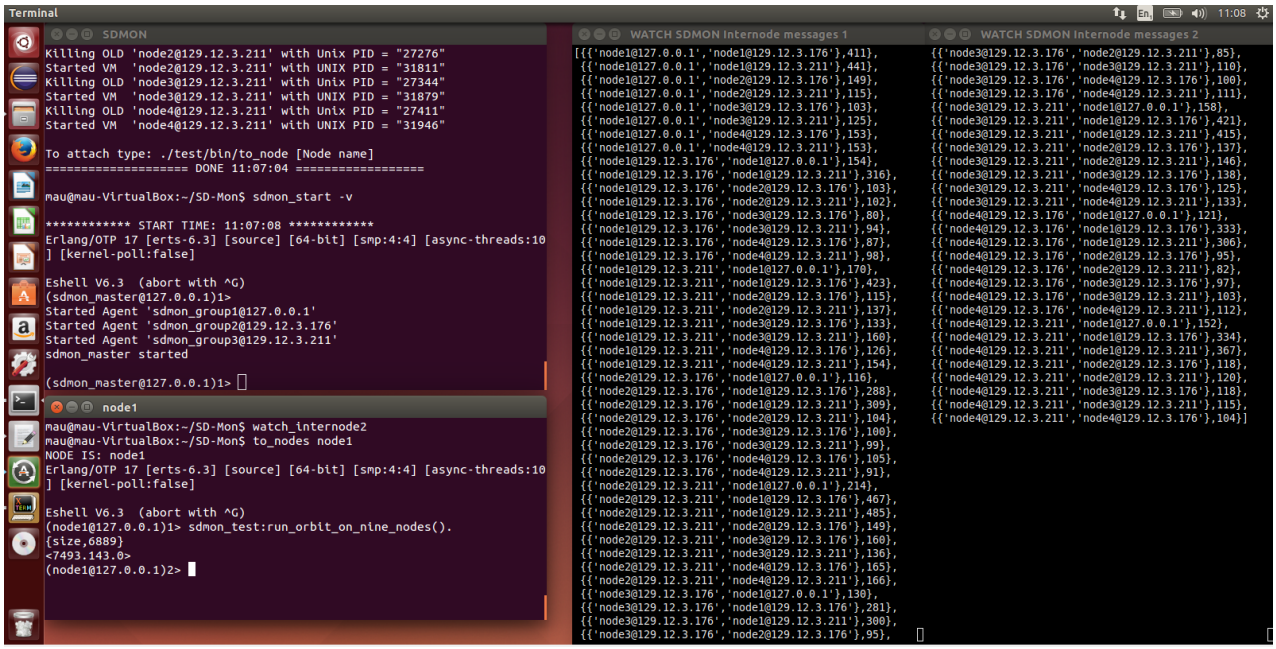



**Figure 4. Run-time visualization**

## 4.3 How to run SD-Mon

The following prerequisite applies to be able to run the tool:

- The user must be able to execute SSH commands on target non-local hosts without the needs to provide a password (use ssh-keygen if needed).
  The userid granted to access remote nodes via SSH without password must be defined in test.config file ('`uid`' tag).

To install the tool, execute the following commands from a terminal:

```
git clone https://github.com/RefactoringTools/SD-Mon
cd SD-Mon
make
```

SD-Mon is started by executing from the installation directory (`BASEDIR/SD-Mon`) the bash script:

```
> sdmon_start
```

configuration files are read and the shadow network is started. SD-Mon is normally executed in detached mode, without a shell. For debugging purposes a '`-v`' option is available: the Erlang shell on the master node is opened and the user can interact with the master or simply follows the system evolution.

By executing:

```
> sdmon_stop
```

SD-Mon is stopped: all tracing is removed, agents are terminated and all tracing files are downloaded in the master file system (`/traces` directory).

### 4.3.1 Example 1: SD-ORBIT on single-host

In this example SD-ORBIT is run on a system composed of 5 nodes, all running on local host and distributed in two s_groups, as depicted in Figure 5.

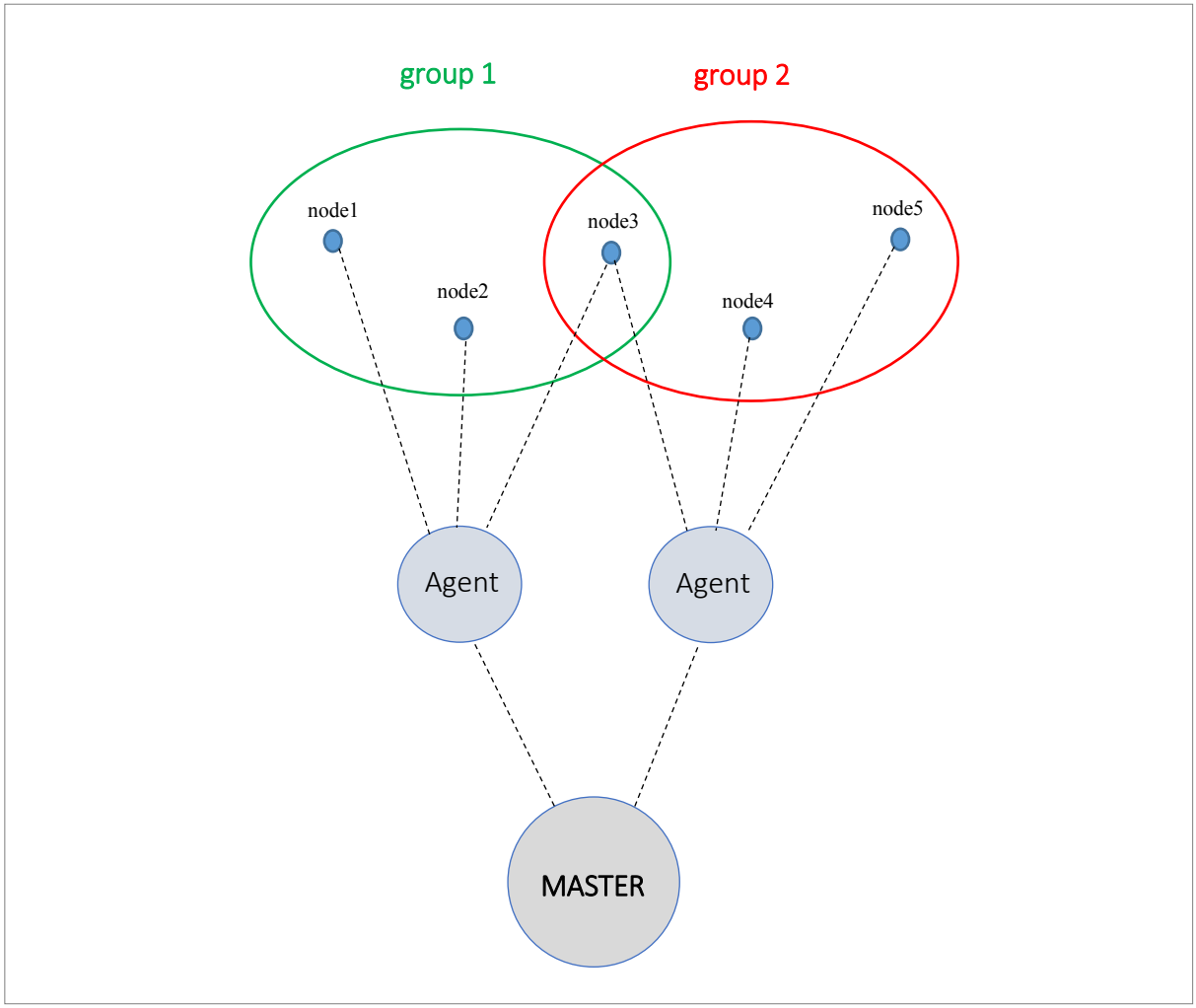



**Figure 5: SD-ORBIT on five nodes**

ORBIT parameters are:

- Generators: bench:g124/1
- Size of space: 50000
- Processors: 8

To execute this test open a terminal and type
(after replacing `<BASEDIR>` with the actual base directory):

```
export PATH=<BASEDIR>/SD-Mon/bin/:<BASEDIR>/SD-Mon/test/bin/:$PATH
cd <BASEDIR>/SD-Mon
cd test/config
rm test.config
ln -s test.config.orbit test.config
```

```
cd ../../
run_env
sdmon_start -v
```

open a new terminal and attach to node1 Erlang shell:

```
export PATH=<BASEDIR>/SD-Mon/bin/:<BASEDIR>/SD-Mon/test/bin/:$PATH
cd <BASEDIR>/SD-Mon
watch_internode
to_nodes node1
sdmon_test:run_orbit_on_five_nodes().
```

The terminal started by the watch_internode command will show internode message counters updating at runtime, each entry in the form `{{FromNode, ToNode}, SentMessages}`.

When Orbit run is completed go back on the first terminal and type:

```
application:stop(sdmon).
```

Find tracing and statistics in `<BASEDIR>/SD-Mon/traces`.

### 4.3.2 Example 2: SD-ORBIT on multi-host

In this case SD-Orbit is run on localhost and on two remote hosts: `myrtle.kent.ac.uk` (129.12.3.176) and `dove.kent.ac.uk` (129.12.3.211).

The orbit parameters are the same as before, with the exception of the size of space, decreased to 10000.

The system is composed of nine nodes grouped in two s_groups, as shown in Figure 6.

To execute this demo, edit the file `BASEDIR/SD-Mon/test/config/test.config.orbit_3h` and replace the string "md504" with the proper user id (see above).

Now open a terminal and (after replacing `<BASEDIR>`) type:

```
export PATH=<BASEDIR>/SD-Mon/bin/:<BASEDIR>/SD-Mon/test/bin/:$PATH
cd <BASEDIR>/SD-Mon
cd test/config
rm test.config
ln -s test.config.orbit_3h test.config
cd ../../
run_env
sdmon_start -v
```

open a new terminal and attach to node1 erlang shell:

```
export PATH=<BASEDIR>/SD-Mon/bin/:<BASEDIR>/SD-Mon/test/bin/:$PATH
cd <BASEDIR>/SD-Mon
watch_internode2
to_nodes node1
sdmon_test:run_orbit_on_nine_nodes().
```

This time message counters will be displayed on 2 terminals.

When Orbit run is completed go back on the first terminal and type:

```
application:stop(sdmon).
```

Find tracing and statistics in `<BASEDIR>/SD-Mon/traces`.

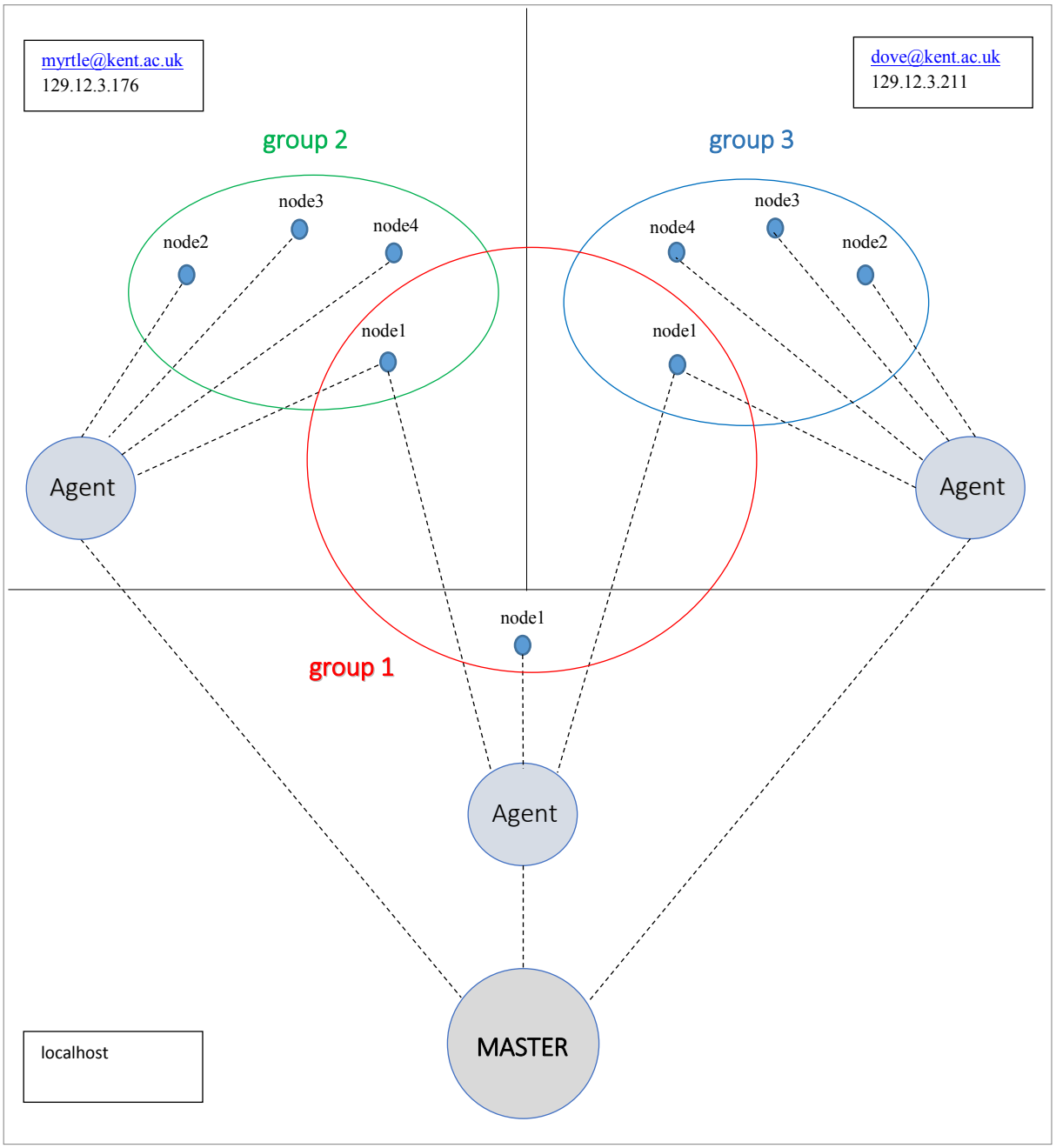



**Figure 6: SD-ORBIT on nine nodes**

Only `inter_node` tracing is enabled in this example, therefore only inter_node and inter_group messages are included in the resulting tracing files.

Here are some statistics reporting sent inter-node messages during a run of this example:

| Node | Inter-node | Inter-group | IG/IN |
|---|---|---|---|
| node1@localhost | 2271 | 1133 | 49.9% |
| | | | |
| node1@myrtle | 1877 | 431 | 23.0% |
| node2@myrtle | 2667 | 1775 | 66.6% |
| node3@myrtle | 2556 | 1683 | 65.8% |
| node4@myrtle | 2481 | 1596 | 64.3% |
| | | | |
| node1@dove | 1874 | 388 | 20.7% |
| node2@dove | 2583 | 1680 | 65.0% |
| node3@dove | 2505 | 1606 | 64.1% |
| node4@dove | 2345 | 1405 | 59.9% |

| | |
|---|---|
| **Total number of inter-node messages** | **21159** |
| Max inter-node messages node sender<br>number of sent messages | node2@myrtle<br>2667 |
| | |
| **Total number of inter-group messages** | **11697** |
| Max inter-group messages group sender<br>number of sent messages | group2<br>5485 |
| **inter-group messages / inter-node messages** | **55.3%** |

The following table summarizes the message flow through "bridge nodes", which are the nodes belonging to more than one group and through which all communication toward and from external groups is supposed to pass.
The incoming messages reported in the table are those sent to the bridge node by any other node that is member of at least one of its groups. For instance, the incoming messages considered for `node1@myrtle` are those sent by `node1@localhost` and `node1@dove` (members of group1) and by `node2@myrtle`, `node2@myrtle`, `node3@myrtle` and `node4@myrtle` (members of group2).
The outgoing messages are instead those sent by the bridge node to any other node that is member of at least one of its groups. For instance, the outgoing messages considered for `node1@myrtle` are those sent to `node1@localhost` and `node1@dove` (members of group1) and to `node2@myrtle`, `node2@myrtle`, `node3@myrtle` and `node4@myrtle` (members of group2).
Inter-node messages not reported in the table are sent or received by external nodes, bypassing the bridging role of the bridge nodes.

| Bridge Flow Tab | **TOTAL INCOMING** | **TOTAL OUTGOING** |
|---|---|---|
| **node1@myrtle** | **2540** | **1446** |
| **node1@dove** | **2752** | **1486** |

Global statistic data for this example can be found in Appendix A2.

# 5 Devo

Devo is an online visualisation tool for SD-Erlang programs, described in Deliverable 5.2 and available online from https://github.com/RefactoringTools/devo. Devo underwent some fairly major changes since D5.2 was delivered: in order

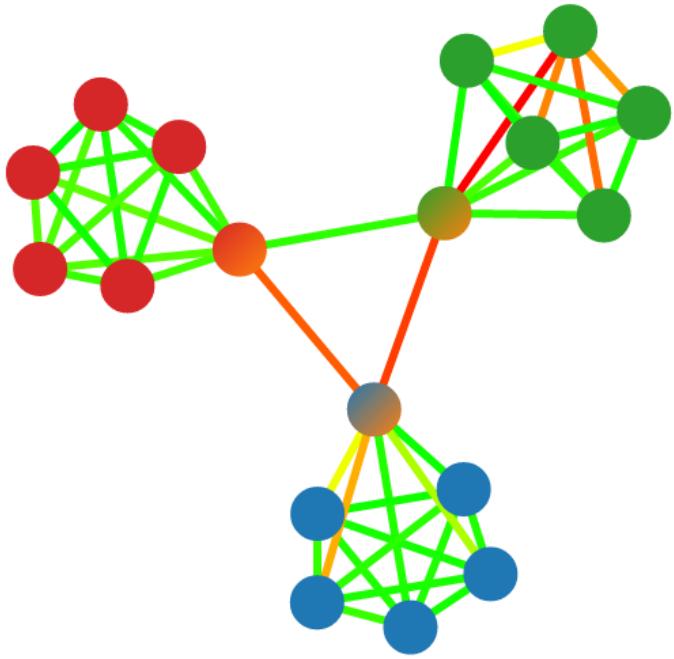
- to make the devo visualisation scalable to situations with more than a handful of s_groups, and
- to make devo easier to deploy and to integrate with other tools

the visualisations were changed significantly and a large amount of refactoring took place in both the server and client-side code.

## 5.1 Visualization Changes

The visualization options available to a user have been simplified to two options “Low” and “High” level visualizations. The low level visualization shows process migrations and the run queue lengths of a single Erlang node. This option was available previously but has been renamed.

The high level visualization has undergone more extensive changes. The shape of the s_groups is displayed using D3’s force-directed graph.



The s_groups to which a node belongs are indicated by the colour(s) of the graph node representing the Erlang node, and the edges connect nodes within the same group. When a single node is in more than one group the node’s colour is split between the two appropriate colours. When the nodes communicate with each other the edges change colour to indicate the level of communication between those two nodes relative to amount other nodes are communicating with each other.

In the previous version, groups were indicated iconically by circles enclosing the nodes, but while this novel visualisation was appropriate for relatively small numbers of s_groups it did not scale to larger numbers of nodes.

## 5.2 Back-end refactoring

Before the new visualizations were completed Devo's back-end was refactored to make it simple to make changes to the system in the future. All dependencies to PHP and Java have been removed. Java was used to help draw the previous high level visualization and PHP facilitated the communication between Javascript and Java. With the update to the high level visualization both PHP and Java became unnecessary, and the visualisation now relies on putting together Erlang and standard web technology, including JavaScript.

Finally we have begun removing hard coded data to allow for Devo to be integrated into other projects.

## 5.3 Future work to Devo

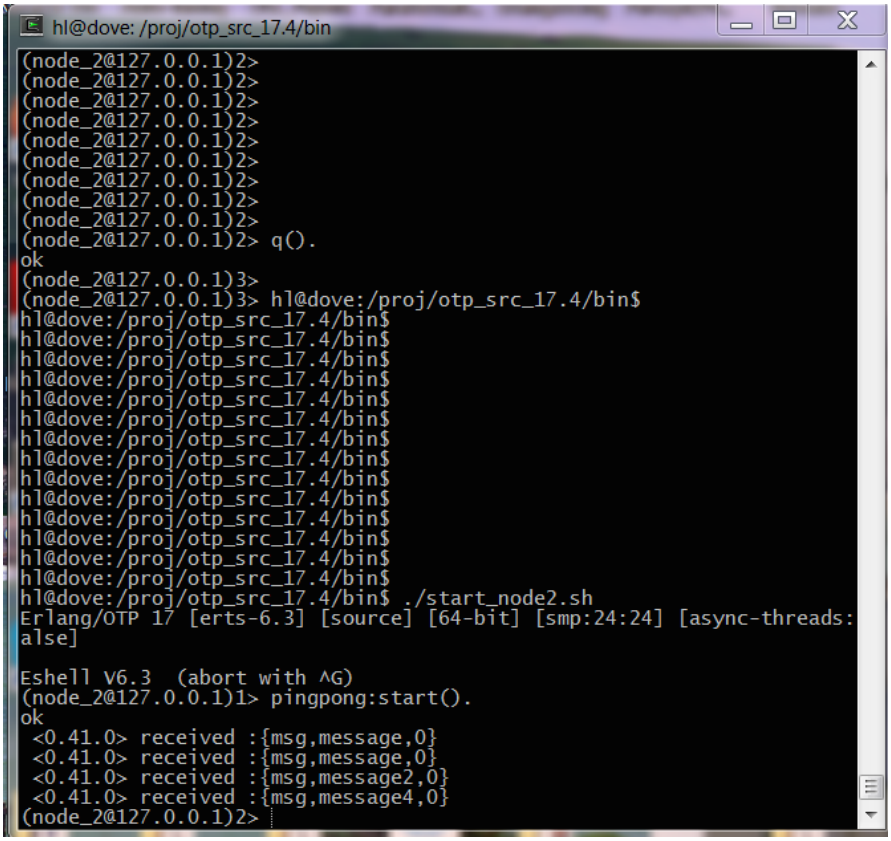
For now future development to Devo will focus on backend developments to increase Devo's robustness and integrability. Increased testing of integrating Devo with other Erlang projects will help refine and expand Devo's integration procedures.

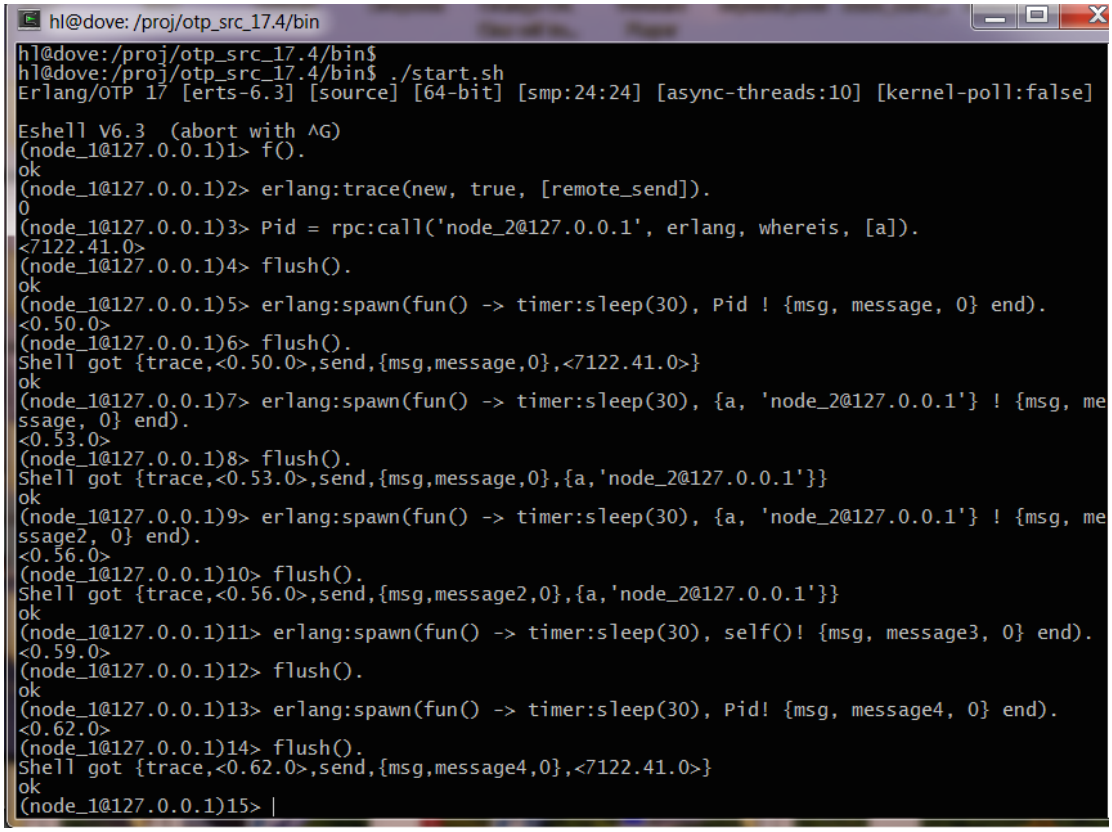
Another major downside that the developers are aware of is that Devo can only profile nodes that a user has told it about. It would be ideal for Devo to automatically detect and profile all of the s_groups and nodes within an Erlang system.

Devo's source repository can be found here: https://github.com/RefactoringTools/devo. If you have any questions or comments about Devo please email Stephen Adams at sa597@kent.ac.uk.

# 6 OTP Patch to augment tracing

Augmenting work reported earlier in the project (WP1, Section 7.2) extensions to the Erlang/OTP built-in trace facility have been submitted as patches to Erlang/OTP. These allow trace messages to be generated selectively – and so not to overwhelm tracing systems – and this has been done to support tracing of only those messages that are to processes on remote nodes, thus supporting monitoring of the distributed aspects of a system.



# 7 Conclusion

In this deliverable we have presented three additions to the Erlang ecosystem specifically designed to support monitoring and debugging of the "Scalable Distributed" (SD) aspects of SD Erlang, namely: *SD-Mon*, which supports off and online monitoring of SD-Erlang systems by means of a shadow network of monitoring nodes; a substantially re-engineered version of *Devo*, designed to give a more scalable visualisation of distributed Erlang systems, and also a *patch to Erlang/OTP* to support more efficient monitoring of inter-node messages in Erlang distributed systems.

## *Change Log*

| Version | Date | Comments |
|---|---|---|
| 0.1 | 28/02/2015 | First version submitted to internal reviewers |

# References

1. SD-Mon: https://github.com/RefactoringTools/SD-Mon
2. Devo: https://github.com/RefactoringTools/devo
3. SD Orbit: https://github.com/release-project/benchmarks
4. Percept2: https://github.com/RefactoringTools/percept2

# Appendix

# A. SD-Mon

## A1. Directory Structure

The SD-Mon directory structure is represented in the schema below.

```
SD-Mon
├── bin
├── config
├── doc
├── ebin
├── logs
├── src
├── test
└── traces
```

| | |
|---|---|
| ── bin | |
| ├── sdmon_start | script to start the tool |
| ├── sdmon_stop | script to stop the tool |
| ├── to_master | script to attach to the master node shell |
| └── to_node | script to attach to any other Erlang node |
| ── config | |
| ├── group.config | the group configuration file |
| └── trace.config | the trace configuration file |
| ── doc/ | tool documentation |
| ── ebin/ | Erlang compiled code + sdmon.app file |
| ── logs/ | contains log files for master and agents |
| ── src | Erlang source code |
| ├── run_env.erl | test environment start and generation of configuration files |
| ├── sdmon_app.erl | sdmon application file |
| ├── sdmon.erl | agent code |
| ├── sdmon_db.erl | erun-time visualization db code |
| ├── sdmon_master.erl | master code |
| ├── sdmon_sup.erl | application supervisor |
| ├── sdmon_trace.erl | tracing code |
| └── sdmon_worker.erl | test code |
| ── test | |
| ├── bin | |
| │ ├── gen_env | script to generate configuration files |
| │ ├── run_env | script to run the test environment |
| │ └── ... | Other utility test scripts |
| ├── config | |
| │ └── test.config | test configuration file, normally a soft link to the proper one |
| ├── ebin/ | Erlang compiled code |
| ├── sderlang/ | test dependency |
| ├── sd-orbit/ | test dependency |
| └── src/ | Erlang source code |
| ── traces | trace files (see section **Error! Reference source not found.**). |

# A2. Global Statistics for Example 2

| Node Tab (NxN) | From Node | To Node | Tot. size | inter-node |
|---|---|---|---|---|
| | node1@127.0.1.1 | node1@129.12.3.176 | 39170 | 539 |
| | | node1@129.12.3.211 | 39311 | 599 |
| | | node2@129.12.3.176 | 32198 | 206 |
| | | node2@129.12.3.211 | 31472 | 173 |
| | | node3@129.12.3.176 | 31043 | 152 |
| | | node3@129.12.3.211 | 31580 | 178 |
| | | node4@129.12.3.176 | 32384 | 215 |
| | | node4@129.12.3.211 | 32039 | 200 |
| | | to_node1@127.0.1.1 | 1167 | 9 |
| | node1@129.12.3.176 | node1@127.0.1.1 | 17905 | 607 |
| | | node1@129.12.3.211 | 8667 | 456 |
| | | node2@129.12.3.176 | 3378 | 159 |
| | | node2@129.12.3.211 | 3258 | 155 |
| | | node3@129.12.3.176 | 2199 | 104 |
| | | node3@129.12.3.211 | 2718 | 129 |
| | | node4@129.12.3.176 | 2535 | 120 |
| | | node4@129.12.3.211 | 2535 | 120 |
| | | sdmon_group1@127.0.1.1 | 13574 | 27 |
| | node1@129.12.3.211 | node1@127.0.1.1 | 19181 | 694 |
| | | node1@129.12.3.176 | 8211 | 389 |
| | | node2@129.12.3.176 | 2424 | 114 |
| | | node2@129.12.3.211 | 2799 | 133 |
| | | node3@129.12.3.176 | 2631 | 125 |
| | | node3@129.12.3.211 | 3054 | 144 |
| | | node4@129.12.3.176 | 2778 | 132 |
| | | node4@129.12.3.211 | 2664 | 126 |
| | | sdmon_group1@127.0.1.1 | 7054 | 17 |
| | node2@129.12.3.176 | node1@127.0.1.1 | 17696 | 667 |
| | | node1@129.12.3.176 | 11670 | 555 |
| | | node1@129.12.3.211 | 10986 | 577 |
| | | node2@129.12.3.211 | 3870 | 183 |
| | | node3@129.12.3.176 | 3102 | 146 |
| | | node3@129.12.3.211 | 3843 | 180 |
| | | node4@129.12.3.176 | 4032 | 191 |
| | | node4@129.12.3.211 | 3540 | 168 |
| | node3@129.12.3.176 | node1@127.0.1.1 | 16301 | 584 |
| | | node1@129.12.3.176 | 11013 | 523 |
| | | node1@129.12.3.211 | 10482 | 553 |
| | | node2@129.12.3.176 | 3447 | 164 |
| | | node2@129.12.3.211 | 3402 | 161 |
| | | node3@129.12.3.211 | 3765 | 178 |
| | | node4@129.12.3.176 | 3903 | 186 |
| | | node4@129.12.3.211 | 4344 | 207 |
| | node4@129.12.3.176 | node1@127.0.1.1 | 15637 | 544 |
| | | node1@129.12.3.176 | 11232 | 534 |
| | | node1@129.12.3.211 | 10551 | 553 |
| | | node2@129.12.3.176 | 3636 | 172 |
| | | node2@129.12.3.211 | 2880 | 136 |
| | | node3@129.12.3.176 | 3783 | 179 |
| | | node3@129.12.3.211 | 3795 | 179 |
| | | node4@129.12.3.211 | 3879 | 184 |
| | node2@129.12.3.211 | node1@127.0.1.1 | 16344 | 597 |
| | | node1@129.12.3.176 | 11253 | 536 |
| | | node1@129.12.3.211 | 10716 | 563 |
| | | node2@129.12.3.176 | 3711 | 176 |
| | | node3@129.12.3.176 | 3939 | 186 |
| | | node3@129.12.3.211 | 3441 | 163 |
| | | node4@129.12.3.176 | 3897 | 185 |
| | | node4@129.12.3.211 | 3747 | 177 |
| | node3@129.12.3.211 | node1@127.0.1.1 | 15743 | 555 |
| | | node1@129.12.3.176 | 11121 | 529 |
| | | node1@129.12.3.211 | 10329 | 542 |
| | | node2@129.12.3.176 | 3870 | 183 |
| | | node2@129.12.3.211 | 4089 | 193 |
| | | node3@129.12.3.176 | 3774 | 177 |
| | | node4@129.12.3.176 | 3408 | 162 |
| | | node4@129.12.3.211 | 3447 | 164 |
| | node4@129.12.3.211 | node1@127.0.1.1 | 13202 | 342 |
| | | node1@129.12.3.176 | 11193 | 532 |
| | | node1@129.12.3.211 | 11283 | 592 |
| | | node2@129.12.3.176 | 3813 | 181 |
| | | node2@129.12.3.211 | 3744 | 177 |
| | | node3@129.12.3.176 | 3681 | 173 |
| | | node3@129.12.3.211 | 3636 | 171 |
| | | node4@129.12.3.176 | 3714 | 177 |

**Sent Tab**

| Node | inter-node | inter-group | IG/IN |
|---|---|---|---|
| node1@127.0.1.1 | 2271 | 1133 | 49.9% |
| | | | |
| node1@129.12.3.176 | 1877 | 431 | 23.0% |
| node2@129.12.3.176 | 2667 | 1775 | 66.6% |
| node3@129.12.3.176 | 2556 | 1683 | 65.8% |
| node4@129.12.3.176 | 2481 | 1596 | 64.3% |
| | | | |
| node1@129.12.3.211 | 1874 | 388 | 20.7% |
| node2@129.12.3.211 | 2583 | 1680 | 65.0% |
| node3@129.12.3.211 | 2505 | 1606 | 64.1% |
| node4@129.12.3.211 | 2345 | 1405 | 59.9% |

**Flow Tab**

| Node | Incoming | Outgoing |
|---|---|---|
| node1@127.0.1.1 | 4590 | 2271 |
| | | |
| node1@129.12.3.176 | 4137 | 1877 |
| node2@129.12.3.176 | 1355 | 2667 |
| node3@129.12.3.176 | 1242 | 2556 |
| node4@129.12.3.176 | 1368 | 2481 |
| | | |
| node1@129.12.3.211 | 4435 | 1874 |
| node2@129.12.3.211 | 1311 | 2583 |
| node3@129.12.3.211 | 1322 | 2505 |
| node4@129.12.3.211 | 1346 | 2345 |

# B. Devo README file

To compile the code you need rebar in your PATH.

Type the following command:

```
$ make
```

Then:

```
$ cd tests
$ erlc *.erl
```

## B1. Introduction to Devo

Devo has two visualisation modes, low and high level. The low level visualisation shows the length of each run queue on each of your processor's cores and the migration of processes between cores.

The high level visualisation is dependent on the SD-Erlang S_group feature. This visualisation shows each of the nodes in your system, colours them according to which s_group they belong to, and displays the amount of message sending that is currently occurring within each s_group.

## B2. The Devo interface

You can start the devo server from the root devo directory by running

```
./start.sh
```

In a web-browser if you navigate to `'localhost:8080'` the Devo home page should be displayed. There should be two radio buttons at the top that allow you to select which type of visualisation you want. Below that is where you must list the nodes that you want Devo to visualize. You can manually enter the node name into the text field or if your nodes follow a numeric pattern (e.g. `'node1@127.0.0.1'`, `'node2@127.0.0.1'`, ...) Devo can generate these names for you. Finally there is the start/stop button that toggles the visualisation on or off.

## B3. Running Examples

Start a devo server as described above.

Open a new terminal window and go to the 'devo/tests' directory and run:

```
./start.sh node1@127.0.0.1
```

An Erlang repl should be running in this terminal now. Go back to the devo web page and add `'node1'` to the list of Devo's nodes, and start the visualisation. You should see the graphic representing your processor's cores but nothing will be running. Go back to the terminal with the node1 repl and enter

```
orbit:run_on_one_node().
```

now if you look at your web-browser you should see the visualisation.

## B4. High Level Visualisation

You must have SD Erlang installed to run the high level visualisation

First, if you just ran the low level example kill the 'node1' repl. In a terminal that is in the 'devo/tests' directory run

```
./fiveNodeLocalStart.sh
```

You should now see an Erlang repl for `'node1@127.0.0.1'`. The difference is that this time there are four additional Erlang nodes running in the background.

Go to the devo webpage and generate five Erlang nodes, the basename is "`node`", the start index is 1, the end index is 5, and the domain is "`@127.0.0.1`". When you start the visualisation this time the five nodes should appear as colored circles. To run this example go back to the `'node1'` repl and enter

```
high_level_test:run().
```

this will take you through a set of s_group operations and distributed computing examples which will be visualised by Devo.

## B5. Integrating Devo into your Project

Using Devo to visualize your own project should be fairly simple.

The first step is including Devo's custom implementation of the DBG module in your project's path. After running the make command you can copy `devo_dbg.beam` from the ebin folder to your project's path.

If your project has some initial s_group configuration without calling new_s_group then this configuration should be placed in an `s_group.config` file. For example config file syntax see:

```
%DEVOROOT%/tests/s_group.config
```

Now you are ready to startup devo and your own project. After both of these services are running navigate to the devo homepage and you should be ready to start profiling.