SD-Mon Tool Description

Content

Contents

[2 Introduction 1](#_Toc410732036)

[3 Overall Architecture 2](#_Toc410732037)

[3.1 Master 3](#_Toc410732038)

[3.2 Agents 3](#_Toc410732039)

[3.3 Robustness 4](#_Toc410732040)

[3.4 Configuration and Trace 5](#_Toc410732041)

[3.5 Post-Mortem analysis 7](#_Toc410732042)

[3.6 How to run SD-Mon 9](#_Toc410732043)

[3.6.1 EXAMPLE 1: SD-ORBIT on single-host 9](#_Toc410732044)

[3.6.2 EXAMPLE 2: SD-ORBIT on multi-host 11](#_Toc410732045)

[4 Future Plans 13](#_Toc410732046)

[5 Appendix 14](#_Toc410732047)

[5.1 Directory Structure 14](#_Toc410732048)

[5.2 Global Statistics for Example 2 15](#_Toc410732049)

Chance Log

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Comment** |
| 1.0 | 03/02/2015 | First version for D5.4 delivery |

# Introduction

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

This purpose is accomplished by means a *shadow* *network* of agents, mapped on 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.

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.

# 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 nodes connection across the network.

Agent

Agent

Agent

Agent

Agent

**MASTER**

**s\_group 1**

**s\_group 2**

**global group**

**free nodes**

Figure 1: SD-Mon shadow network

## Master

The Master is gen\_server started and supervised by a supervisor process at application start-up.

The shadow network is deployed according to configuration. In order to limit the 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 in nodes to be monitored and traces to be run on them. Each network change in s\_group is cached by agents and notified to the master, which is the only one having a global network view. It takes care to reshape 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 quitted, 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 (originals are saved) in order to allows the tool to consider 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 $HOME/SD-Mon/traces/ directory and decoded on the fly: a text file is produced for each controlled node. A summary of statistics is also created for each agent and at global system level.

System events are logged by the master under $HOME/SD-Mon/logs.

## 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 to get aware of s\_group operations (create or delete an s\_group, add or remove nodes to a s\_group) happening 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.

Agent

Agent

**s\_group 1**

**s\_group 2**

Agent

new

Agent

new

Agent

**s\_group 1**

**free nodes**

Figure 2: Delete s\_group 2

## 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 grants a fast recovery from network failures.

once   
a second

Master

Agent

nodedown

start direct monitor

ping

ping

pong

nodeup

monitor

last token

Token not matching

get configuration

configuration

Restart tracing on all nodes

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.

## Configuration and Trace

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

* group.config
* trace.config

The first one (also used by s\_group processes) is a kernel environment and defines 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 $HOME/SD-Mon/config/.

They can be compiled 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 HOME/SD-Mon/test/config/

This file consists of a set of Erlang terms defining the system to be observed. It is composed of 4 sections:

1. The first section just lists the hosts where the system is running.
2. The second section is an integer stating how many Erlang nodes (virtual machines) should be started for each host.
3. In this part groups are listed
4. The last part describes traces to be applied on each group

Currently a few tracing options are supported but more can be added in future.

* *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.  
N.B. 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).

In section 4 it is also possible to define Erlang commands to be executed on the monitored system through a MFA tuple.

## 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

1. 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

1. 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

1. 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 5.2.

## How to run SD-Mon

Some prerequisites apply to be able to run the tool:

* SD-Mon must be installed in the home directory of the local host and on all non-local hosts.
* 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).

SD-Mon is started by executing from the base directory ($HOME/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).

### 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 4.

ORBIT parameters are:

* Generators: bench:g124/1
* Size of space: 100000
* Processors: 8

Agent

Agent

**MASTER**

**group 1**

**group 2**

node3

node1

node2

node4

node5

Figure 4: SD-ORBIT on five nodes

To execute this test open a terminal and type:

export PATH=$HOME/SD-Mon/bin/:$HOME/SD-Mon/test/bin/:$PATH

cd SHOME/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=$HOME/SD-Mon/bin/:$HOME/SD-Mon/test/bin/:$PATH

cd $HOME/SD-Mon

to\_nodes node1

sdmon\_test:run\_orbit\_on\_five\_nodes().

back on the first terminal:

application:stop(sdmon).

find tracing and statistics in: $HOME/SD-Mon/traces.

### 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 5.

Agent

**MASTER**

**group 2**

**group 3**

node3

node1

node2

node4

node1

node1

Agent

node4

Agent

**group 1**

node3

node2

[myrtle@kent.ac.uk](mailto:myrtle@kent.ac.uk)  
129.12.3.176

[dove@kent.ac.uk](mailto:dove@kent.ac.uk)  
129.12.3.211

localhost

Figure 5: SD-ORBIT on nine nodes

To execute this demo, edit the file $HOME/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 type:

export PATH=$HOME/SD-Mon/bin/:$HOME/SD-Mon/test/bin/:$PATH

cd SHOME/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=$HOME/SD-Mon/bin/:$HOME/SD-Mon/test/bin/:$PATH

cd $HOME/SD-Mon

to\_nodes node1

sdmon\_test:run\_orbit\_on\_nine\_nodes().

back on the first terminal:

application:stop(sdmon).

find tracing and statistics in: $HOME/SD-Mon/traces.

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  number of sent messages | node2@myrtle  2667 |
|  |  |
| **Total number of inter-group messages** | **11697** |
| Max inter-group messages group sender  number of sent messages | group2  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 5.2.

# Future Plans

|  |  |  |
| --- | --- | --- |
| Rationale | Priority | Effort |
| 1. Inter-group messages info on WEB (cowboy) | ? | \*\*\* |
| 1. TRACES: add more options (system profiles, …) | \* | \* |
| 1. SCALABILITY and Nonfunctional requirements 2. Submaster level 3. Efficient data handling (ets or key-val as dict (faster on read) or gb\_trees) | \*\*\*\*  \*\*\*  \*\*\* | \*\*\*\*  \*\*\*  \*\* |
| 1. Wombat Integration | ? | \*\*\* |

# Appendix

## 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\_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 3.5).

## Global Statistics for Example 2





