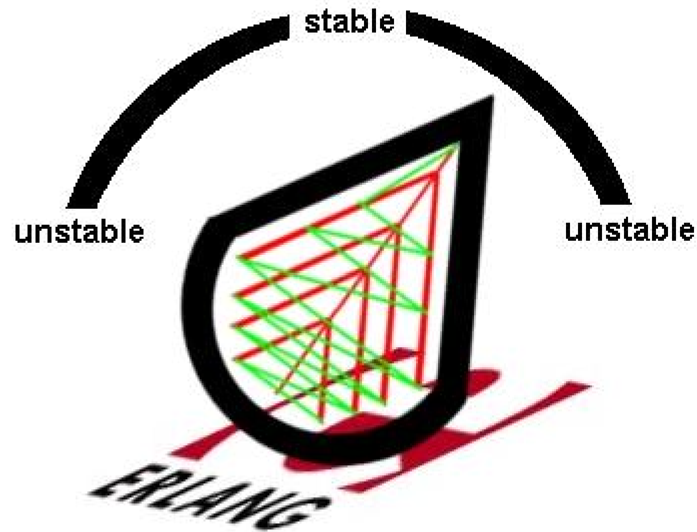
CARL VON OSSIETZKY universität OLDENBURG



SiSSDA

Simulator for **S**elf-**S**tabilizing **D**istributed **A**lgorithms
to Determine Fault-Tolerance Measures

Manual

Carl von Ossietzky Universität Oldenburg

This Manual is part of the Diploma-Thesis

Simulation of Distributed Self-Stabilizing Algorithms
to Determine Fault-Tolerance Measures

Group System Software and Distributed Systems

Contents

1 **Foreword** 1

2 **Setup** 3

2.1 Windows . . . 3

2.2 Linux . . . 4

2.3 Development Environment . . . 5

3 **Getting Started Quickly** 7

4 **Getting Started Slowly** 9

4.1 Using Erlang . . . 9

4.1.1 Standard Literature . . . 9

4.1.2 Required Commands . . . 10

4.2 Configuration . . . 11

4.2.1 Accuracy . . . 11

4.2.2 Fault Environment . . . 14

4.2.3 Execution Semantics . . . 16

4.2.4 Fault Injector . . . 16

4.2.5 Fault Probability Correction . . . 16

4.2.6 Arbitrary Initial Values . . . 17

4.2.7 Logging of Events . . . 17

4.2.8 Scheduler Behavior . . . 17

4.2.9 Server . . . 18

4.2.10 Version . . . 18

4.2.11 Verbose . . . 18

4.2.12 Topologies . . . 18

4.3 Execution Flow 18
4.3.1 Initialization 21
4.3.2 Execution 26
4.3.3 Post Processing 28

5 How to extend the Simulator **31**
5.1 Adding new Topologies 31
5.2 Adding new Algorithms 33
5.2.1 Spanning Tree Algorithms 33
5.2.2 Value Obtaining Client Algorithm 35
5.2.3 Observer Specific Functions 35
5.3 Adding new Fault-Environments 36
5.4 Tweaks 37

Bibliography **39**

1. Foreword

This Manual is intended for users and developers that use the Simulator for Self-Stabilizing Distributed Algorithms (SiSSDA). Steps required to run the simulator are discussed in chapter 2. To keep up with Erlang-style introduction, this manual features a quick-start guide as well as a slow-start guide.

- If you like to run the simulator and get a first impression, take a look in chapter 3.
- For users that like to take a deeper look into and behind the simulator, chapter 4 is most suitable as important adjustments are explained.
- For developers that are already acquainted with the simulator chapter 5 will be interesting for adjusting and extending the simulator to individual requirements. The simulator is built to cope with appropriate topologies as well as new algorithms and even dynamic fault-environments that meet individual needs.

If you like to get used to Erlang and discover the advantages of distributed-computing, please refer to section 4.1, where a list of standard literature is given and the most important features are briefly explained.

2. Setup

To run the simulator, one or more computers are needed that run on Windows NT (including NT 4.0, 5.0 2000, 5.1 XP, 6.0 Vista) or Linux. For testing the simulator, a heterogeneous network was used containing PCs running Windows 2000, Debian 3.0 Sarge and Ubuntu 6.06 Dapper Drake at the same time running Erlang R11B1.

Please mind, that for Ubuntu 6.10 Edgy Eft there are known issues when trying to compile Erlang R11B1. Different solutions to this problem are discussed in the mailing list[1].

Another issue using Erlang is, that for cryptographic features an installed version of OpenSSL is required[2]. As those features are not required by the simulator since the random-number generator included is used instead of the one provided by OpenSSL, this will not be mentioned in the following guide.

Please mind, that, although the simulator was implemented to run in a network, the single nodes are only simulated by threads such that a graph can also be simulated on one single PC. There is no notable improvement using multiple computers with execution semantics set to 1 (please also refer to section 4.2.3) since server, client and fault-injector execute successively. Nevertheless, future algorithms that require higher execution semantics and more processor ressources and maybe even a continuous time model featured by a new scheduler might encounter a massive impact on the time required for execution leading to favoring a distributed execution of client ressources.

2.1 Windows

Obtain the actual version of Erlang from http://www.erlang.org/download.html by clicking in the second row in the column labeled *Windows binary (incl. documentation)* on `yes`.
After downloading double-click the file to install and follow the on-screen instructions.

[1] http://www.erlang.org/pipermail/erlang-questions/2006-November/023941.html
[2] http://www.openssl.org/

For secure communication within a LAN-environment Erlang requires a cookie to be set up, which contains exactly the same phrase on each computer. The cookie has to be set up as a file labeled `.erlang.cookie` in the path the variable `$HOME` points to. Usually this is `C:\Documents and Settings\`*`user name`*, where *user name* is a placeholder for your current user name.

For Windows, a special executable is available in `C:\Program Files\erl*\bin` (* indicating the version) labeled `werl.exe`. This file has the feature of tab-completion which is missing in the DOS-shell version.

2.2 Linux

If you are using a packet-manager such as apt, you might want to use it to get Erlang. If you are using apt,

1. run `apt-get update` to get current status
2. run `apt-cache search erlang` to check whether you have the necessary repositories.
3. if this does not show you the package `erlang-base`
   - open `/etc/apt/sources.list` with a text-editor
   - append the lines
     `deb http://neutronic.mine.nu/ unstable/`
     `deb-src http://neutronic.mine.nu/ unstable/`
     to the list
   - save the file and close the editor and
   - restart at top of list.
4. The important packages are
   - erlang-mode - a plug-in for emacs,
   - erlang-base - the runtime environment,
   - erlang-base-hipe - HiPE (High Performance Erlang)[3], a highly optimized version that is not officially supported by the simulator and is not a required, but recommended package for optimizing the simulator and
   - erlang-src - source files for the system (*.erl, *.hrl).
5. Install accordingly with
   `apt-get install` *packetname*
   Note that it might be reasonable to build dependencies first with
   `apt-get build-dep erlang erlang-base`

To compile erlang manually, please ensure that you have the packages

- openssl

[3] http://www.it.uu.se/research/group/hipe/

- unixodbc-dev
- make
- gcc

properly installed. Obtain the current version of Erlang from http://www.erlang.org/download.html, extract the source code with `tar -xzvf otp_src_R*B-*.tar.gz` (where * indicates the version) and execute

- `sudo ./configure`
- `sudo make`
- `sudo make install`

Now you should be able to run Erlang from command-line by simply entering `erl`. To compile Erlang files (*.erl), please run `erlc *.erl` from command line.

2.3 Development Environment

Since Ericsson does not deliver an IDE as Sun does with Beans for Java[4], one reasonable way to implement with Erlang is using Eclipse[5] enhanced by the ErlIDE plug-in[6]. Eclipse is available for Linux and Windows. For information on installing and running Eclipse and Erlang, please refer to the corresponding websites since changes regarding the use or installation are possible.

The plug-ins Texlipse[7] and Subclipse[8] have been used for development, too.

- *Eclipse* is a free IDE developed by IBM as successor to Visual Age. It is designed as an integrated development environment (IDE) for Java and intends to use the Concurrent Versions System (CVS) as repository. For this project Erlang has been chosen as programming language for its advantages in distributed concurrent computing, and Subversion (SVN) was chosen as repository for its advantages in version control in contrast to CVS.
- *ErlIDE* is a plug-in for Eclipse to gain functionality of Erlang within the IDE. With ErlIDE, syntax-highlighting and a console are available. One important feature, the debugger, is missing yet but will be released soon according to ErlIDE developer Vlad Dumitrescu.
- *SubClipse* is a plug-in for Eclipse enhancing the IDE by giving the functionality to handle subversion repositories from within the IDE making external tools such as RapidSVN (Linux) or TortoiseSVN (Windows) obsolete. The final version of this thesis is available from the SVN repository

  https://moradin.svs.informatik.uni-oldenburg.de/svn/phoenix/

  If it is not reachable anymore, please feel free to contact me at nils.muellner@gmail.com

[4]http://www.netbeans.org/
[5]http://www.eclipse.org
[6]http://erlide.sourceforge.net/
[7]http://texlipse.sourceforge.net/
[8]http://subclipse.tigris.org/

- *Texlipse* is a plug-in to handle Latex-files with Eclipse. Also the tool Texnic-Center [9] has been used as sidekick to develop this thesis. As LaTeX -frame the wiss-doc package by Roland Bless has been used[10].

[9]http://www.toolscenter.org
[10]http://tm.uka.de/~bless/wissdoc.tar.gz

3. Getting Started Quickly

To run the simulator follow these steps:

- Ensure that Erlang is set up as described in chapter 2.
- Copy the source files (CD-ROM/SiSSDA/*.*) into a directory of your choice.
- Adjust the hostnames in `global_config.hrl` to the hostname of your PC. These are given by the variables

```
1 -define(fault_injector,{fault_injector,fault_injector@HOSTNAME}).
2 -define(server,{server,server@HOSTNAME}).
```

  Simply exchange the HOSTNAME to the hostname the desired service is executed on.

- Start a console and change path according to where you copied the simulator-files, e.g.
- Compile the sources with `erlc *.erl`
- Start the server from within the directory where the source-code is available:
  - `erl -sname server`
  - `server:start().`
- Follow the on-screen instructions:
  - Choose an algorithm (for example for *Depth-First Search* enter 2.) and
  - choose a topology (for example, for *serial8* enter 1.).
- When asked, start sufficient number of clients each in its own shell from within the directory where the source-code is available:
  - `erl -sname client#` where each client has a unique number
  - `client:start().`

- After the environment is set up properly, the simulator only requires the fault-injector to join to begin with the simulation. The fault-injector is started with

  - `erl -sname fault_injector`
  - `fault_injector:start().`

  from within the directory where the source-code is available

- When the simulation is finished, each node shuts itself down automatically.

For further information the next chapter is recommended.

4. Getting Started Slowly

To reasonably use the simulator, many parameters have to be taken into account. Not only the algorithms are available in more than one distinct version, but also the behavior of the scheduler and the desired accuracy for example are important factors that must not be disregarded.

In the following section Erlang will be introduced. To minimize redundancy of information with the thesis, references will be given at corresponding places.

4.1 Using Erlang

Although Erlang is a potent language, it is quite easy to learn. Most standard literature is available online. Since a complete introduction is unfeasible for this thesis, only basic characteristic features are mentioned.

4.1.1 Standard Literature

For a complete guide please refer to the following literature:

- The present standard book is *Concurrent Programming in Erlang* [AVWW96] by Joe Armstrong, Robert Virding, Claes Wikström and Mike Williams from 1996. Although this book is quite old, most of the important features are still discussed in a sufficiently actual manner. The first chapter was published free of charge and contains an introduction to the most important features[1].
- The coming up standard book is *Programming Erlang* [Arm07] by Joe Armstrong. As this book will be released in July 2007, is could not be used for this thesis. Yet, it will be the designated successor to *Concurrent Programming in Erlang.*
- Another good source to deepen knowledge in Erlang is the PhD thesis by Joe Armstrong [Arm03], which is available at http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf.

[1] http://www.erlang.org/download/erlang-book-part1.pdf

- The site http://wiki.trapexit.org/ delivers many examples like the cookbook series released by O'Reilly.
- http://www.planeterlang.org/ is a portal where professionals from the Erlang community exchange knowledge and make it public in blogs. Many current projects and their difficulties are discussed in this portal. In contrast to the mailing list, *planeterlang* is a platform only for professionals.
- The standard beginners guide is available at http://www.erlang.org/doc/doc-5.5.3/doc/getting_started/part_frame.html. This tutorial is intended for very beginners and can be accomplished within one day. A similar introduction is available at http://www.erlang.org/course/course.html, featuring also a historic review of Erlang.
- The site http://www.erlang-projects.org/ was established by Mickaël Rémond, the author of the french book *Erlang programmation* [RA03]. It contributes to popular projects developed in Erlang like the Jabber-Server *ejabbered*.

4.1.2 Required Commands

For using the simulator, only few features are mandatory to know.

1. Every input is ended with a dot.
2. The decimal separator is a dot. If asked for a decimal with $0 < INPUT < 1$, 0.1. is a possible input. The first dot is the separator and the second dot finishes the input.
3. Variables start with a capital letter.
4. Macros are referred to with a leading question mark.
5. A tuple is a set of attributes with a distinct number of elements, framed by curly braces. For example $A = \{a, b, c\}$. is a tuple.
6. A list is a set of attributes with a variable number of elements, framed by brackets. For example $B = [a, b, c]$. is a list. To get the first element of a list, a pipe operator is used. For example `[First|Rest] = B.` returns *First* as *a* and *Rest* as $[b, c]$.
7. Message passing is executed with an exclamation mark in the format `{server, server@hostname} ! Message.`
   The first part is a tuple, containing the process name and the according machine the referenced process is executed on. The exclamation mark is equivalent to `erlang:send(Dest, Msg)-> Msg` and `Message` is the message that is passed. Commonly `Message` is a tuple itself since usually multiple parameters are transfered (using lists for message passing is considered to be bad style). Unlike Java, a tokenizer is obsolete. The pattern matching guards are also able to look into tuples such that the first parameter in a passed tuple is often used to indicate the tuples purpose [AVWW96, section 1.3].

8. Erlang has no variables. Variables are instantiated constants. That means, a process has to be re-instantiated before a variable descriptor can be assigned again.
9. The logical *and* respectively && is expressed as a comma , in Erlang.
10. The logical *or* respectively || is expressed as a semicolon ; in Erlang.
11. The logical *not* respectively ! is expressed as / in Erlang such that *not equal* in Erlang is expressed as / =.
12. *mod* respectively *modulo* is expressed as *rem* (remainder) in Erlang.
13. The $\geq$ and $\leq$ operators are expressed such that the spike of the arrow points to the equal sign $a \geq b \mathrel{\hat{=}} a >= b \mathrel{\hat{=}} b =< a$.

For using the simulator, only the first two points have to be taken into account.

4.2 Configuration

The whole configuration of the simulator is done in the file `global_config.hrl`. The `.hrl` file extension means *Erlang header*. There are no functions given in header files. The following parameters are available to adjust:

name	possible values	default value
accuracy	$1000000 >= X >= 0.000001$	1
accuracy_field	$length([0, 0, ..., 0]) >= X >= length([0, 0])$	$[0, 0, 0, 0, 0]$
accuracy_min_run_length	$X \in \mathbb{N}, X >= 1$	1000
env	default	default
exec_sem	$X \in \mathbb{N}, X >= 1$	1
fault_injector	{fault_injector, fault_injector@hostname}	to be adjusted
faulty_fault	$X \in \mathbb{R}, 1.0 >= X >= 0.0$	0
init_arbitrary	true, false	true
log_file	desired log-file name	logged_events.log
logging	enable, disable	disable
scheduler_behav	random, ordered	random
server	{server, server@hostname}	to be adjusted
version	any string with length of 3	`"1.0"`
verbose	true, false, topology, client, sched	false
tops	list of topologies	please refer to sec. 5.1

4.2.1 Accuracy

The accuracy of a simulation is defined by the parameters

- `accuracy`,
- `accuracy_field` and

- `accuracy_min_run_length`.

The desired accuracy is reached, if the maximal difference of the values measured as availability in the last defined steps is smaller than a certain value and a minimal number of steps has been executed:

$$\begin{array}{l} if \\ \quad (max(accuracy_field) - min(accuracy_field)) * 1000000 =< accuracy, \\ \quad Steps_{executed} >= accuracy_min_run_length \rightarrow \\ \qquad accuracy = true. \end{array}$$

With $Length = length(accuracy_field)$ being the length of the list, the latest $Length$ measured values for availability are recorded. This means, being in step N, the list $accuracy_field$ contains the calculated availabilities for the steps N, $N-1$, ..., $N-(Length-1)$.

If the maximal difference between the values in $accuracy_field$ is less or equal the macro $accuracy$ multiplied with 1000000, and at least $accuracy_min_run_length$ number of steps were executed, adequate stability is reached.

For example, after the execution of N steps with $N \in \mathbb{N}, N >= Length$, the list $accuracy_field$ contains $Length$ values. If the simulator was started with initial arbitrary values (please refer to subsection 4.2.6) and the predefined legal set of states was not reached until step N, the list $accuracy_field$ will only consist of zeros. The first criteria for reaching the desired degree of stability is, that at least every field in $accuracy_field$ was initialized by the simulator such that at least $Length$ steps were executed.

Since the predefined set of legal states can only be reached in the absence of further transient faults, it is reasonable to introduce a helping variable that forces the simulator to execute a minimal number of steps that eventually exceeds the size of $Length$. This variable is labeled $accuracy_min_run_length$ with $accuracy_min_run_length \in \mathbb{N}, accuracy_min_run_length >= Length$.

If within $accuracy_min_run_length$ number of steps stability was not reached and the simulator was started with arbitrary values, obviously the corresponding availability will be 0. To maximize accuracy, the length of $accuracy_field$ might be increased while the value for $accuracy$ is decreased. Figuratively, the following graph shows the impact of the accuracy parameters mentioned.

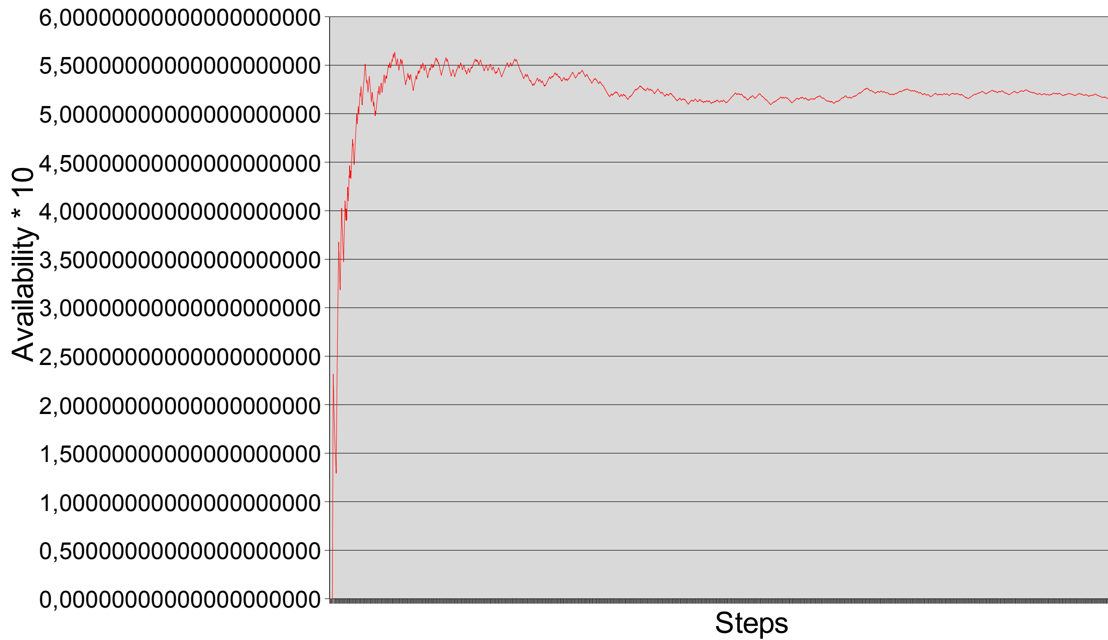


Figure 4.1:
This graph shows the measured availability for the first 20.000 steps with {DFS, SERIAL8, GFP 0.02}.

This graph was the result of a run that was used to measure availability. It shows the availability for each of the first 20.000 steps, using

- an *accuracy* of 1,
- an *accuracy_field* with the length of 5 and
- an *accuracy_min_run_length* of 1000.

Obviously, the first results show a value of 0, since several steps are required to converge to the predefined legal set of states. Since the *accuracy_field* is filled with zero values again in these first steps, the maximal difference between these values is zero, too. In step 74 the system has stabilized, so the value for availability for step 74 is $1/74 = 0.\overline{135}$. This means, that the maximal difference for the values of availability of the steps 70, 71, 72, 73 and 74 is $0.\overline{135}$. The accuracy of 1 must not be greater than this maximal difference multiplied with 1000000 to satisfy the first stability criteria. Since $0,\overline{135} \cdot 1000000 = 1351351.\overline{351}$ and $1351351.\overline{351} > 1$, the first criterion for stability is not met.

This procedure is repeated in step 75 with the values of availability of the steps 71, 72, 73, 74 and 75 accordingly until the maximal difference is smaller than 1.

The complete execution of the simulation of the configuration described above consisted of 565399 steps. The minimal bounding box defined by *accuracy* (vertical edges) and *accuracy_field* (horizontal edges) would not be visible in the graph 4.1 since it is too small!

The first occurrence of a legal state is depending on the algorithm, the topology and especially the fault-possibilities. For example, in a sufficiently large topology with

relative large values for fault-possibilities, chances are low for the system to reach the legal set of states. To measure these almost infinitely small values, an accordingly large number of steps has to be simulated. This can be achieved by setting the value for *accuracy_min_run_length* to an appropriate value. The tests performed in this thesis indicate, that a value of 1000 is adequate and that the values measured become unreliable for a measured availability below 1%. This Assumption is also supported by the number of steps executed. For low values of fault-possibilities, few steps are executed. The number of steps executed to meet the stability criteria rises until a certain point that is characteristic for the combination of algorithm and topology. After this point, the number of required steps decreases according to a rise of fault-possibility values. If too few steps are executed, results become unreliable, leading to a resulting availability of 0 if the predefined set of stable states was not reached within *accuracy_min_run_length* number of steps.

4.2.2 Fault Environment

Although only a static fault-environment was required, the possibility of changing parameters during runtime according to runtime environments that are defined by the values for the parameters

- *Algorithm*,
- the complete configuration labeled *Client_Data* and
- the execution semantics *Exec_Sem*

is provided. While changing the algorithm during runtime is strictly not advised since data-structures of configurations are not verified to be compatible, it might be reasonable to adjust the configuration and even the execution semantics to simulate environmental influences such that the simulator is not only influenced by internal parameters. While a static environment where both node and edge fault-possibilities do not change during execution was intended to use for the thesis, the possibility to extend the simulator with further environments was implemented to cope with possible future demands. The default environment implies a graph, where fault possibilities do not change during execution as shown for a configuration-example in the following graph:

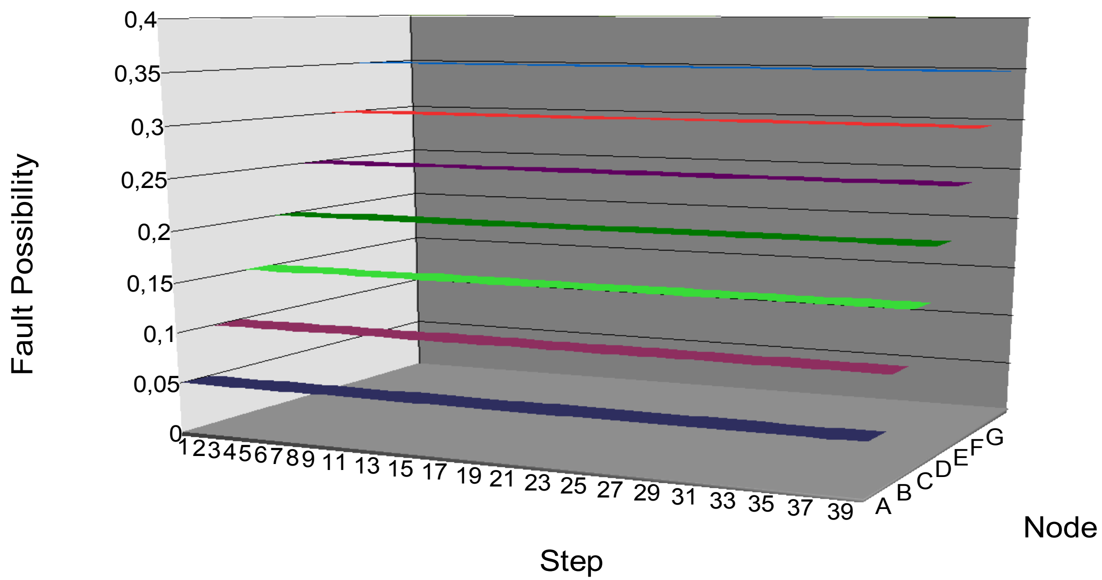


Figure 4.2:
This graph exemplifies the default fault-environment for eight nodes showing the first 40 steps.

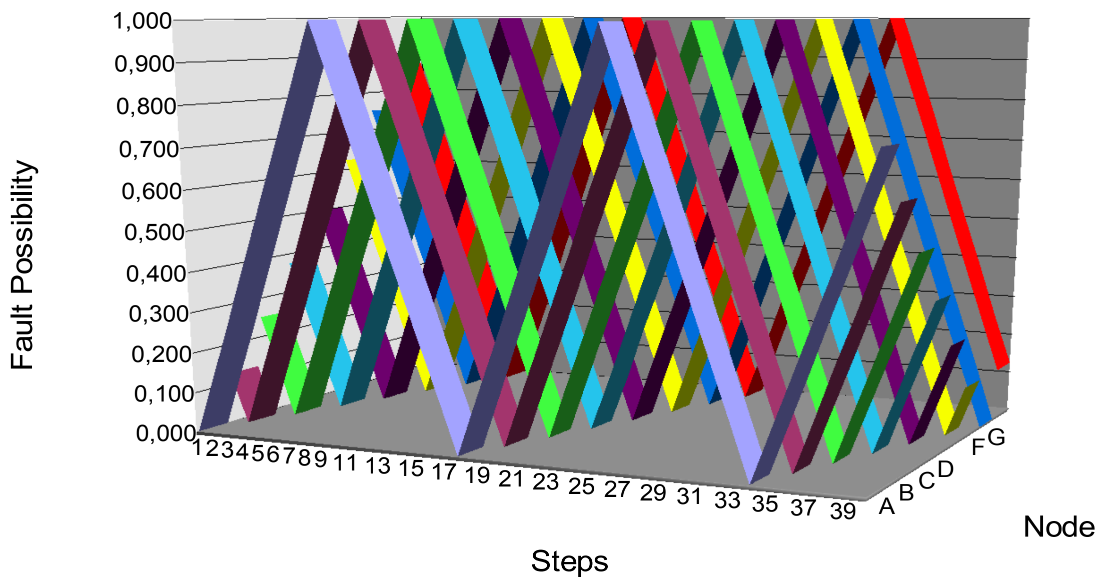


Figure 4.3:
This graph exemplifies a dynamic behavior for eight nodes showing the first 40 steps. It was not implemented since it is only an example for the possibilities for the fault-environment feature.

In contrast, a dynamic behavior can be related on either algorithmic behavior or random behavior or a combined strategy based on the input. The following graph even features a canonical behavior such that a distinct wave is repeated after a

certain time. This behavior models delayed impact in the different nodes due to a phase shift of one calculation step. Possible scenarios for this example shown in figure 4.3 are likely to be time triggered events like seasons or day- and night-time that influence the nodes on different time steps due to spatial intervals between the nodes.

Although this graph features an environmental influence that is independent of the current or past configuration, environments featuring burn-in and burn-out-phases or other environments depending on the current and / or past configuration are feasible.

Parameters that can not be presented in an illustrious fashion are the alteration of the algorithm and the execution semantic accordingly although they can be changed alike.

4.2.3 Execution Semantics

Being also referred to as *Hamming Distance* in the thesis (section 2.3.2) with a theoretical background, this option was statically set to a value of 1 for all tests featured in this thesis. Although the implementation was not part of the task, it has been implemented for future purposes.

The parameter indicates the maximal number of nodes that are allowed to execute parallel in the same execution step. This leads to the problem of interlocking dependabilities, solved with time-out events that lead to a local state that is not part of the predefined legal set of states.

Assuming that a local network environment is used to run the simulator, the timeout has been set to 0.005 seconds (line 113 in client.erl, `after 5 ->`), thus leading to a tremendous increase for the global execution time the more interlocking dependabilities have to be attended.

Since this feature was not part of the task, values greater 1 have not been tested thoroughly and a correct execution can not be guaranteed.

4.2.4 Fault Injector

The value set for `fault_injector` consists of one tuple containing two parts. The first one is the name of the process that grants the desired services. The second one is the node where the antecedent process is available on. The interesting part is the name after the @ sign, where the term HOSTNAME has to be replaced by the current hostname. If you do not know your hostname, simply start an Erlang console and enter
`inet:gethostname().`
This will return a tuple `{ok, "HOSTNAME"}` showing your hostname which you have to insert as described above.

4.2.5 Fault Probability Correction

Earlier versions contained the possibility that despite the presence of a transient fault a node eventually gets the correct value. Since this behavior is not common for real systems, this value was outsourced to the global variable $faulty_fault$ which is

applied to node- and edge-faults before execution. The term $faulty_fault$ derives from the impression, that a false fault delivers the right result.

For example, if the original global fault possibility (GFP) equals 0.1 and $faulty_fault$ is set to 0.1, this leads to a resulting GFP of 9% since every tenth fault will deliver the correct result.

For the simulated tests the variable $faulty_fault$ was set to zero, assuming that a 64-bit register has $2^{64} = 18446744073709551616$ possible values and that only one of them leads to the correct result. This means for a real system the possibility to achieve a legal state in presence of a fault has a possibility of $\frac{1}{18446744073709551616} = 5,42101086242752217003726400434970 * 10^{-20}$ which is sufficiently small to be disregarded.

4.2.6 Arbitrary Initial Values

If the value *init_arbitrary* is set to *true*, the simulator will use arbitrary values for the initial configuration of the nodes status. Otherwise, if set to $false$, nodes will start with a status that is in the predefined set of legal states.

It is reasonable to set this value to *true* if a stable state is unlikely to be reached and the accuracy described in subsection 4.2.1 should not depend on the value *accuracy_min_run_length* but only on the other two variables defining the accuracy. On the other hand it is reasonable to set this value to $false$, if the GFP is set sufficiently low and the accuracy should not be determined by the value of *accuracy_min_run_length*.

Another reason to start with the system in a legitimate state is the measurement of reliability as discussed in section 3.2.1 in the enclosed thesis.

4.2.7 Logging of Events

The variable log_file simply contains the desired name of the log-file. The variable *logging* can be set to *enable* oder *disable*. This variable is set to *disable* by default since recording of the configuration leads to a tremendous rise of execution time. This should only be set to enable if the configuration of each step is required for further analysis. Please also refer to sections 5.4 and 4.3.3, where possibilities of post processing are discussed and keep in mind that for the logging of one million steps an available hard disk space of about 500MB is required. Successive execution of configurations will not overwrite previous results since new results are always appended to the file. For this reason the file has to be deleted before hard disk space runs out. Alternatively, the term *append* in line 58 in the file `server.erl` may be deleted, changing from an appending to an overwriting behavior.

4.2.8 Scheduler Behavior

Although not explicitly required, two schedulers were implemented. If *scheduler_behav* is set to *random*, the scheduler uses an aging strategy that rises the possibility of being chosen for execution according to waiting time. This strategy is common for real systems. The random scheduler satisfies the terms of liveness, fairness and safety as indicated by the almost equal number of steps each node executes.

For special scenarios, as conceivable especially for the mutual exclusion algorithm, a second scheduler has been implemented, choosing the nodes ordered by their id, starting with node a, continuing with node b and so on until the node with the highest id is reached, continuing with node a afterwards again.

4.2.9 Server

Analogously to subsection 4.2.4, this variable has to be set to point to the computer where the server application is available.

4.2.10 Version

The variable $version$ was used to differ between different branches in development. It indicates the version number of the simulator, which, due to the fact that the development finished, is `"1.0"`.

4.2.11 Verbose

The verbose option leaves many possibilities to trace the execution of steps during runtime.

true	All possible output is printed.
false	No output is printed.
topology	During initialization, the initial working configuration will be printed (phase *topology* in server.erl).
client	Each client will print each step of it's configuration, regardless if the node executed a step or was idle (phase 4 in client.erl).
sched	The fault injector will print a list of elected nodes each step (phase 1 in fault_injector.erl).

Please mind, that I/O-operations are quite costly in contrast to the rest of the simulation and moreover the current value measured as availability for every 1000th step is printed in any event (please also refer to subsection 4.3.1).

4.2.12 Topologies

The given topologies are explained thoroughly in the thesis in section 2.4.5. The adding of further topologies is explained in section 5.2 in this manual.

4.3 Execution Flow

The execution flow basically contains three parts:

- initialization to gather and prepare data,
- execution to use the data collected in the previous phase to aggregate results and
- post processing to use and interpret the information delivered by the simulator.

The first step is to compile the source code. This can be done by typing `erlc *.erl` from the console:

```
phoenix@mystra:~$ cd run/mystra/
phoenix@mystra:~/run/mystra$ erlc *.erl
./client_algorithm_mutex.erl:70: Warning: the guard for this clause evaluates to 'false'
./client.erl:100: Warning: the guard for this clause evaluates to 'false'
./client.erl:133: Warning: the guard for this clause evaluates to 'false'
./fault_injector.erl:76: Warning: this clause cannot match because a previous clause at line 74 always matches
./fault_injector.erl:97: Warning: this clause cannot match because a previous clause at line 93 always matches
./fault_injector.erl:102: Warning: the guard for this clause evaluates to 'false '
./matrix_init_bfs.erl:109: Warning: the guard for this clause evaluates to 'false'
./matrix_init_dfs.erl:106: Warning: the guard for this clause evaluates to 'false'
./matrix_init.erl:54: Warning: the guard for this clause evaluates to 'false'
./matrix_init_le.erl:108: Warning: the guard for this clause evaluates to 'false '
./matrix_init_mutex.erl:104: Warning: the guard for this clause evaluates to 'false'
./server.erl:94: Warning: the guard for this clause evaluates to 'false'
./server.erl:240: Warning: the guard for this clause evaluates to 'false'
./server.erl:260: Warning: the guard for this clause evaluates to 'false'
phoenix@mystra:~/run/mystra$
```

Figure 4.4:
The compiler is intended to throw several warnings that may be disregarded.

These warnings may be ignored as they are thrown due to the predefinition of the verbose variable.

Alternatively files can be compiled with `c(`*filename*`)` singularly from within the Erlang shell. Note, that the extension `.erl` is not used and header files (`.hrl`) must not be compiled.

Compiling the sources generates `.beam` files. Beam stands for *Björn's (respectively Bogdan's) Erlang Abstract Machine*. These files are executable from the Erlang shell. To start an Erlang shell, usually the command `erl` is executed from a Linux console or the file `werl.exe` is executed under Windows NT. Since the simulator was implemented to cope with real distributed systems, the Erlang network interface is required. For that reason, a designated server node is initialized by starting Erlang on both, Linux and Windows, with additional parameters:

```
phoenix@mystra:~/run/mystra$ erl -sname server
```

There has been a recent discussion about the use of the `-sname` and `-name` parameters on the mailing list. The simulator was built using strictly the `-sname` parameter for communication in a local area network, although it should also be able to run in a wide area network using the `-name` parameter. For further information please refer to [AB, section 3.4] and [Hof, slide 13].

To start the simulator, the first part to be executed is `server.erl`. This is achieved by entering

```
server:start().
```

from the previously started server node. The system can now be reached as process `server` on the node `server@hostname` with the command

`{server, server@hostname} !` *message.*

The simulator consists of 20 files. 18 of them are used for the simulator. They are relying on each other according to the following figure:

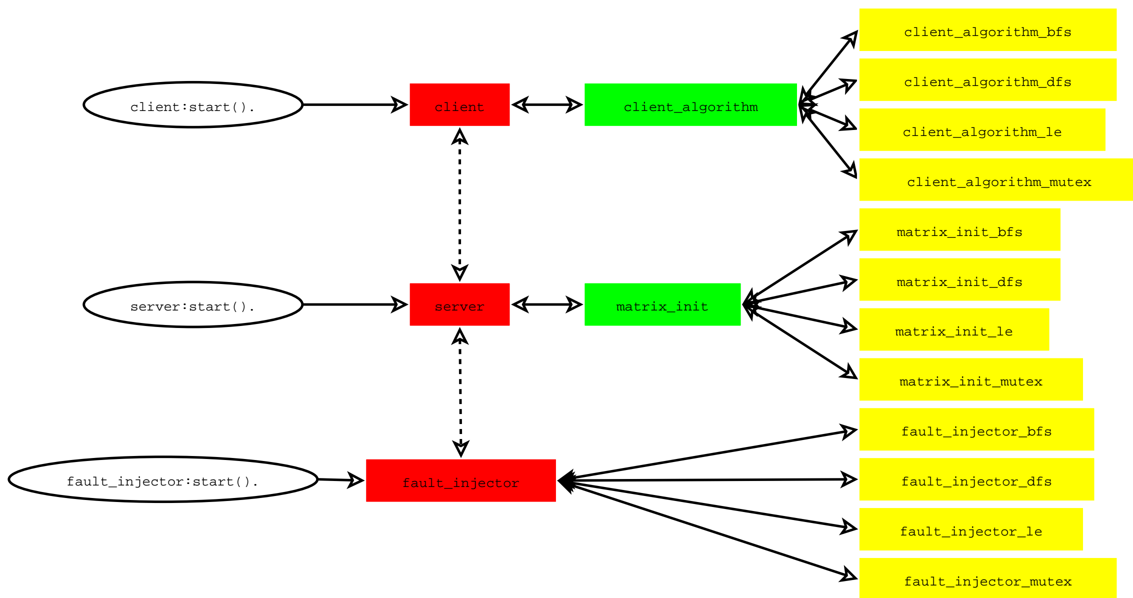


Figure 4.5:
This figure shows how the single components are used. The dotted lines indicate communication, straight lines indicate hierarchical usage. Boxes are classes, ovals are start-routines. Red classes are main classes, green classes are interfaces and yellow classes are subclasses.

The remaining two files are `k.erl` and `fault_env.erl`. The class `k` can be used to prematurely shutdown the server, fault-injector and client applications by entering `k:k().` as discussed in section 5.4. Looking at the source code, the class `k.erl` consists only of the function `k` which uses the global variable `server` from the header file and sends a token labeled `kill` to the server which ensuing initiates the shutdown sequence.
The class `fault_env.erl` features only one static fault environment as described in sections 4.2.2 and 5.3.

After starting the server, the simulator starts gathering and generating information required for initialization:

```
phoenix@mystra:~/run/mystra$ erl -sname server
Erlang (BEAM) emulator version 5.4.12 [source] [threads:0]

Eshell V5.4.12  (abort with ^G)
(server@mystra)1> server:start().
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%                                                                     %%%%%
%%%%%                                SiSSDA                               %%%%%
%%%%%                                                                     %%%%%
%%%%%                                 v "1.0"                             %%%%%
%%%%%                                                                     %%%%%
%%%%%                                                                     %%%%%
%%%%%                  Welcome to the Simulator for                       %%%%%
%%%%%             Self-Stabilizing Distributed Algorithms                 %%%%%
%%%%%                                                                     %%%%%
%%%%%                                SERVER                               %%%%%
%%%%%                                                                     %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% INITIALIZATION-PHASE 1: CHOOSE ALGORITHM                            %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
true
%%%%% The following algorithms are available:                             %%%%%
%%%%%    [1]bfs
%%%%%    [2]dfs
%%%%%    [3]le
%%%%%    [4]mutex
%%%%% Please enter the appropriate number [n.]>1.
```

Figure 4.6: The server application initially offers a choice between algorithms.

Now that the server has been started, initialization begins.

4.3.1 Initialization

The initialization consists of five sub-phases which are successively traversed in `server.erl`.

- The first phase is labeled *alg* (`Phase == alg ->`; server.erl line 62). In this phase, all possible algorithms are taken from the global variable `tops` (please also refer to section 5.2) and then listed in alphabetical order as shown above. To choose one of these algorithms, the appropriate number indexing each algorithm has to be entered followed by a dot as mentioned in subsection 4.1.2. A wrong input will repeat the phase until a valid algorithm is chosen.

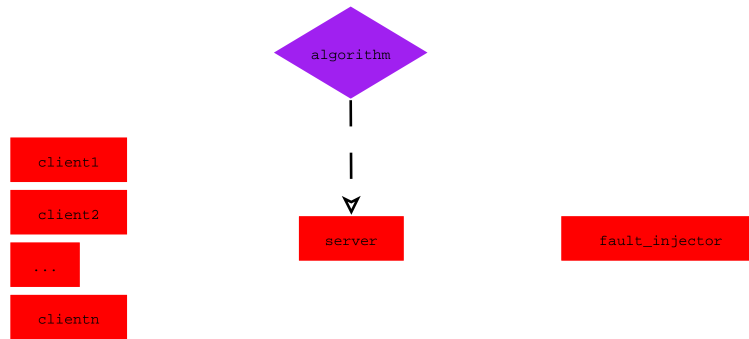


Figure 4.7: Initialization Step 1: Entering Algorithm

- Now that an algorithm has been chosen to be simulated, an appropriate topology is required. This phase is labeled *topology* in `server.erl` (`Phase == topology ->`; line 88). The simulator offers only topologies that are defined as feasible for the chosen algorithm as discussed in section 5.1.

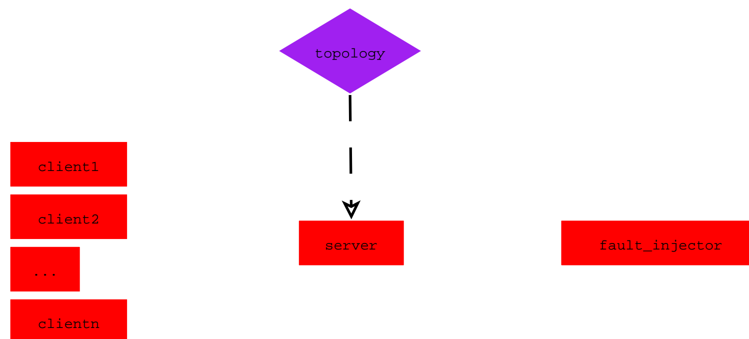


Figure 4.8: Initialization Step 2: Entering Topology

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% INITIALIZATION-PHASE 2: CHOOSE TOPOLOGY                              %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% The following topologies are available:                              %%%%%
%%%%%    [1]COMPLEX8
%%%%%    [2]PARALLEL8
%%%%%    [3]SERIAL8
%%%%%    [4]RING8
%%%%% Please enter the appropriate number [n.]>
```

Figure 4.9:
Subsequently the server offers a selection of appropriate topologies according to the previously chosen algorithm.

- If, for example, *Mutual Exclusion* was chosen as algorithm, the only topology offered is *ring8*. As in the previous step, the choice can be referenced by the according number.

- In the next phase, fault possibilities can be entered. This operation is carried out in phase *augmenting* (`Phase == augmenting ->`, line 118). For the user's

convenience, a batch processing routine was implemented to assign global fault possibilities for nodes and edges. The routine accepts `y.`, `yes.`, `n.` and `no.` as tokens. If answered in the affirmative, the user may first enter the global node fault possibility (GlobalNFP). Otherwise local node fault possibilities (LocalNFP) are required. The format for NFPs may be either integer or decimal smaller or equal one and greater or equal zero. For example, 0. is a valid argument as well as 0.9999999999. is a valid argument. Erlang has an accuracy for decimals of up to 20 digits past the dot. Please note that the last dot is always interpreted as finish of the command.
Entering invalid values returns to the very beginning of this step. This option has proven reasonable since accidentally entered wrong values can be corrected entering intentional invalid types in the sequent step.

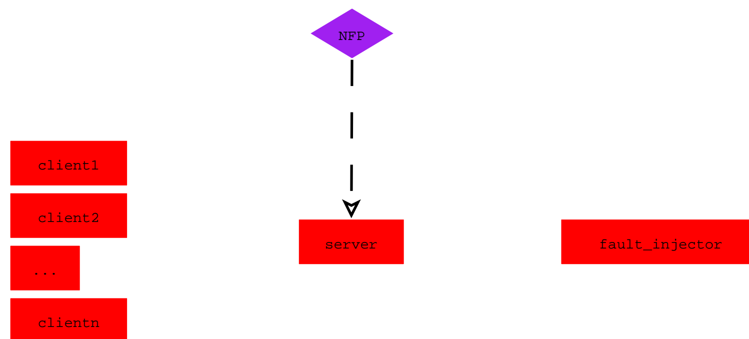


Figure 4.10: Initialization Step 3a: Entering NFP

- Analogously edge fault possibilities can be entered. If the type-check fails, the whole phase starts over again with requesting NFPs again.

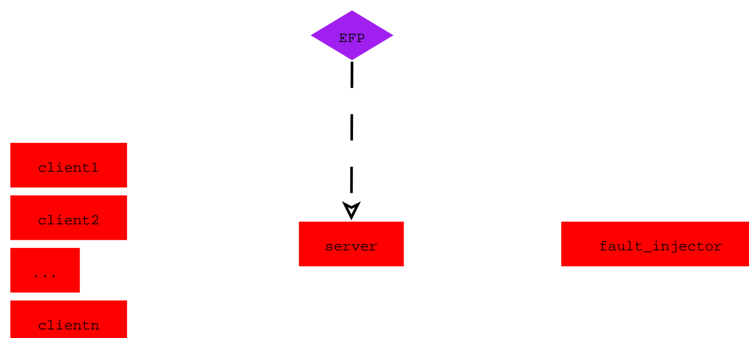


Figure 4.11: Initialization Step 3b: Entering EFP

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Do you want to use GLOBAL NFP? [y. / n.]:>y.
%%%%% Please enter GLOBAL NFP [0.000000. =< r =< 1.000000.]:>0.15.
%%%%% ACCEPTED                                                             %%%%%
%%%%% Do you want to use GLOBAL EFP? [y. / n.]:>y.
%%%%% Please enter GLOBAL EFP [0.000000. =< r =< 1.000000.]:>0.15.
%%%%% ACCEPTED                                                             %%%%%
%%%%% Topology successfully initialized.                                   %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% PLEASE CONNECT 8 CLIENTS...
```

Figure 4.12:
The third phase requires information about node fault probabilities (nfp) and edge fault possibilities (efp).

- The last two steps require a sufficient number of clients and one fault-injector to connect. This phase follows directly to the previous phase *augmenting* and is labeled *connect_clients* (`Phase == connect_clients ->`; line 130).
  In contrast to the server and the fault-injector that are precisely defined in the header to guarantee their reachability, client nodes are implemented without the need of a static address since future algorithms might demand dynamic addresses. Analogously to the server, clients are initialized as shown in the following figure. After each client has connected to the server he shows a message with the unique id the server allocated and the server shows a message with the process id (pid) that connected as client as well as the appropriate id.

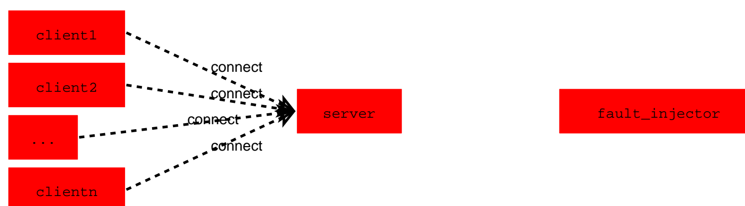


Figure 4.13: Initialization Step 4: Connecting Clients

After a sufficient number of clients has connected, the server asks the user to start the fault-injector.

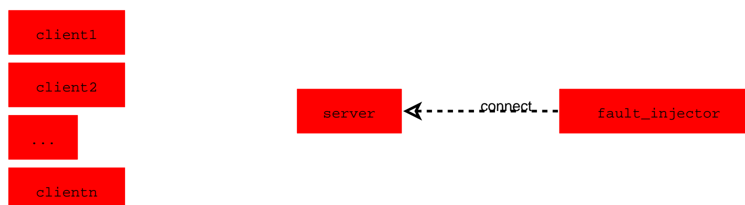


Figure 4.14: Initialization Step 5: Connecting Fault-Injector

If a client fails to connect to the server or the server application is not ready yet, no id will be given to the client. In that case it is reasonable to shut down the client and first start the server and ensure proper communication.

```
(client4@talona)17> client:start().
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%                                                                     %%%%%
%%%%%                                SiSSDA                               %%%%%
%%%%%                                                                     %%%%%
%%%%%                                 v "1.0"                             %%%%%
%%%%%                                                                     %%%%%
%%%%%                                                                     %%%%%
%%%%%                  Welcome to the Simulator for                       %%%%%
%%%%%             Self-Stabilizing Distributed Algorithms                 %%%%%
%%%%%                                                                     %%%%%
%%%%%                              CLIENT                                 %%%%%
%%%%%                                                                     %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
true
%%%%% Connected successfully. This node has role of node a
(client4@talona)18>
```

Figure 4.15:
After the data-structure has been completely initialized, clients connect with their process id (pid) and get the role of a node. In this figure, the client gets the role of node a.

```
%%%%% Client a is now identified as process <4753.39.0>
%%%%% Please connect 7 more clients.
%%%%% Client b is now identified as process <4754.39.0>
%%%%% Please connect 6 more clients.
%%%%% Client c is now identified as process <4755.39.0>
%%%%% Please connect 5 more clients.
%%%%% Client d is now identified as process <4756.39.0>
%%%%% Please connect 4 more clients.
%%%%% Client e is now identified as process <4757.39.0>
%%%%% Please connect 3 more clients.
%%%%% Client f is now identified as process <4758.39.0>
%%%%% Please connect 2 more clients.
%%%%% Client g is now identified as process <4759.39.0>
%%%%% Please connect 1 more clients.
%%%%% Client h is now identified as process <4760.39.0>
%%%%% SUFFICIENT NUMBER OF CLIENTS CONNECTED                              %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% PLEASE START THE FAULT-INJECTOR NOW                                 %%%%%
```

Figure 4.16:
Each time a client has connected, an id is allocated to the process and the remaining number of clients is given.

- Analogously to server and clients, the fault-injector is started as shown below in the last phase of the initialization (`Phase == fault_injector ->` line 158).

After starting the fault injector, the simulator will automatically start with the execution. To observe the status, every 1000th step the current availability is printed in the server-window. Additional output is available according to the setting of the verbose variable (please refer to subsection 4.2.11).

```
phoenix@talona:~/workspace/erlang/demoX02$ erl -sname fault_injector
Erlang (BEAM) emulator version 5.4.12 [source] [threads:0]

Eshell V5.4.12  (abort with ^G)
(fault_injector@talona)1> fault_injector:start().
```

Figure 4.17:
Starting the fault-injector is the final step required to initialize the simulator. After starting the fault-injector, processing of simulation starts immediately.

```
%%%%% FAULT-INJECTOR CONNECTED SUCCESSFULLY                                %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
%%%%%      SYSTEM IS NOW RUNNING. PLEASE WAIT...
%%%%%
%%%%%
%%%%% STEP: 1000     LEGAL: 0.239960
```

Figure 4.18: After starting the fault-injector, simulation begins.

4.3.2 Execution

The execution consists of six global phases. The following figures show the processing for one step traversing through each of the six phases.

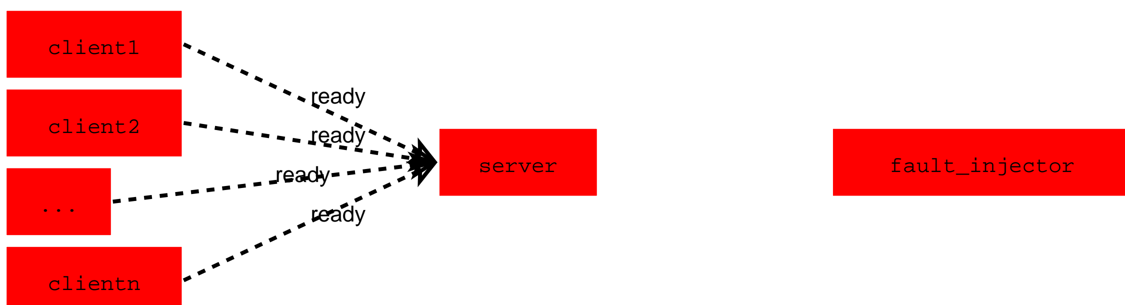


Figure 4.19: In step0 clients propagate that they are ready to execute one step.

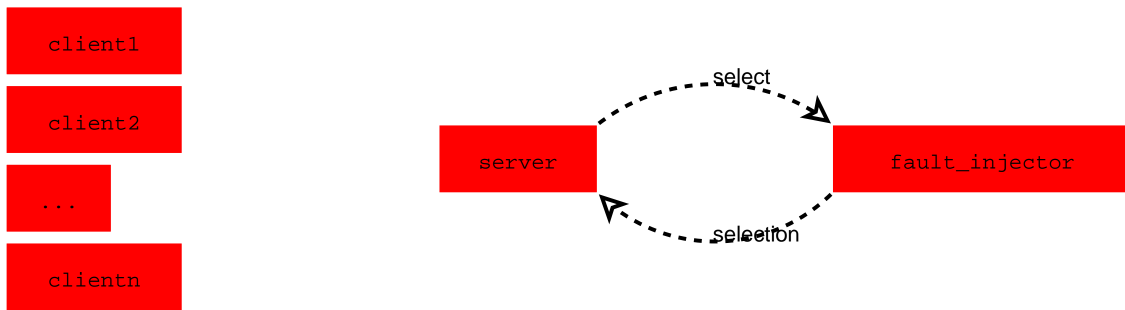


Figure 4.20:
In step1 the server passes the current configuration to the fault-injector. Based on the transferred data, the fault-injector chooses an appropriate number of clients.

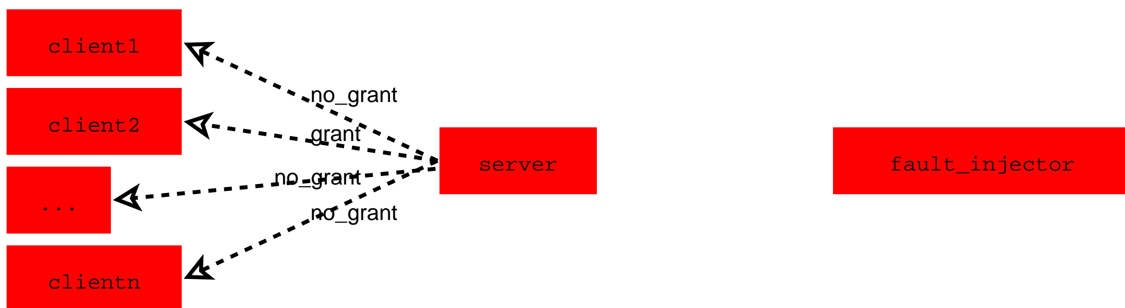


Figure 4.21:
Each client gets a token from the server according to the previous step. If the token reads idle, the client listens to other clients requests. Otherwise the token is a list and reads grant such that results can be acquired.

Figure 4.22:
According to the passed tokens, each client listens or executes one step.

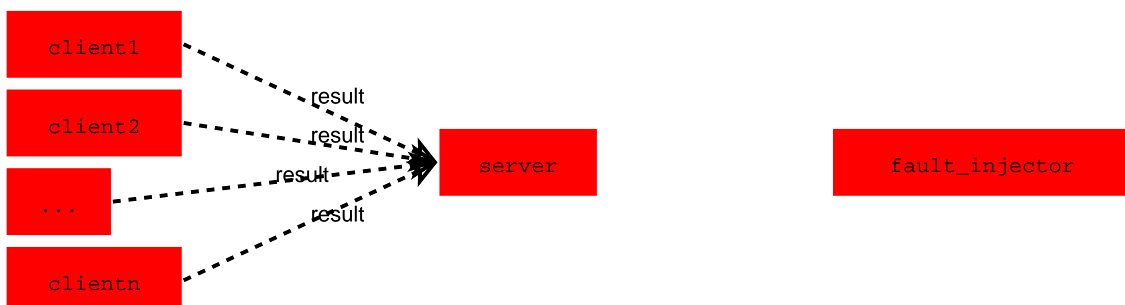


Figure 4.23: After executing one step, clients propagate their results to the server.

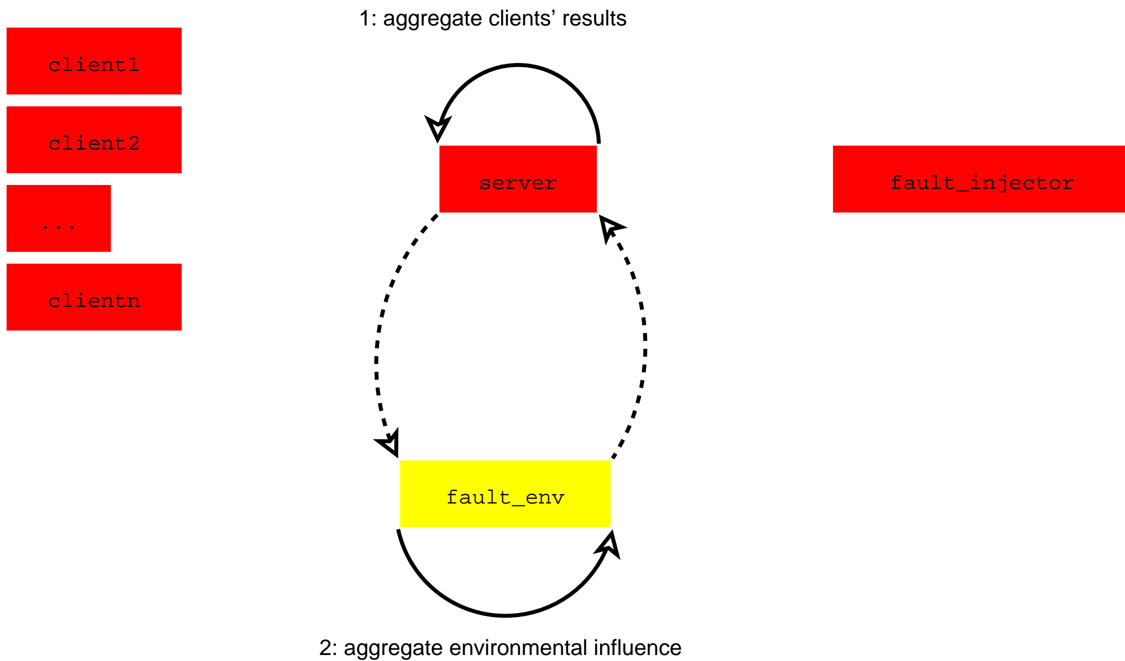


Figure 4.24:
The user may interfere with results gathered by using the fault-environment. First, the server aggregates the developed results and passes them to the fault-environment. Afterwards, the fault-environment may influence the results and passes them back to the server. By default the results are not influenced in this step.

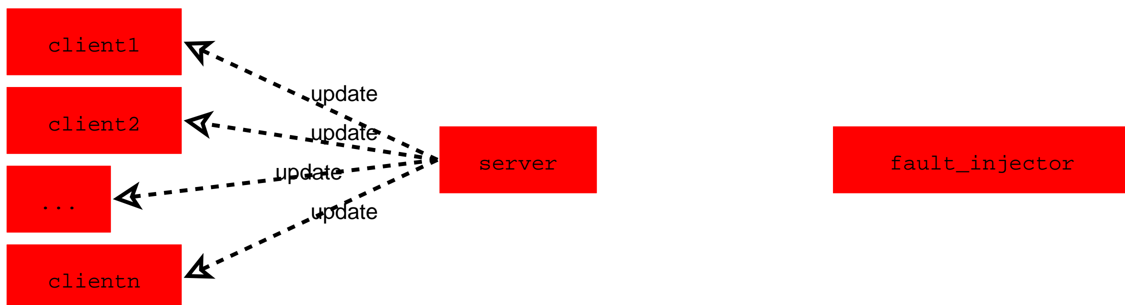


Figure 4.25:
Finally, the current status is reported to client nodes This step might become necessary if algorithms require the clients to operate based on current data and the topology is not static.

4.3.3 Post Processing

Post processing suffers from various restrictions. First, Erlang writes only as binary. Every term exported to a file has to be reformatted to a binary first

```
Bytes = term_to_binary(New_Server_Data),
file:write(File, Bytes);
```

before being flushed to the log-file. Furthermore, several possibilities to continue with the developed results are possible like using Matlab or GNUPlot. These have

different formats. The initial idea was to let MatLab calculate the stability criteria discussed in 4.2.1. During development it was considered to be more reasonable to calculate these in Erlang to save execution time. Another reason was, that network interfaces of MatLab are insufficient and there is no known way to connect MatLab to Erlang during the execution of the simulator.

The simulator can be used in different ways.

- The appropriate value measured for *availability* is printed directly to the console. Simulating several scenarios as done for this thesis can be used to generate appropriate graphs disposing the feature of recording the simulation steps.
- Using the logging function grants the user to post process the simulation of one scenario. The results logged are accessible from the recorded file (see section 4.2.7). The file can be processed using standard Linux commands like

  ```
  strings logged_events.log | less > output.log
  ```

  to stream the data into a new file satisfying appropriate format requirements.
- Furthermore, it is easy to stream any kind of parameter the simulator uses into a file. As shown in Figure 4.1 for example, recording the availability for each step shows the leveling off results measured. For this graph, not the whole data structure but only the the variable

  ```
  Elem = lists:last(New_Acc_Field),
  ```

  has been recorded, transformed as described above after simulation and finally copied into OpenOffice Calc, a free spread sheet analysis tool, for post processing.

5. How to extend the Simulator

As intended for the scenarios featured in this thesis, the simulator uses a limited range of attributes. The accuracy is limited to a certain point to demonstrate the different scenarios for comparison reasons, the fault-environment is set to a static behavior and the output is quite restrictive.

Yet, the simulator can be used for a lot more purposes. For example, the speed with which fault-tolerance measures swing into a certain region is characteristic for distinct scenarios. The composition of a spanning tree without further simulation might be interesting. Even client behavior can be observed live during runtime since input-/output-operations need a lot of time compared to the simulation.

For these reasons the files *global_config.hrl* and *fault_env.erl* have been implemented.

5.1 Adding new Topologies

Adding of new topologies is quite simple. All topologies are defined in the file *global_conf.hrl* within the variable *tops*. It is recommended to use a unique name for each topology as an identifier. The topologies do not need to be ordered since they are ordered automatically by their name.

Each topology is a list itself, consisting of

1. an identifier of the topology,
2. a list containing the algorithms that are eligible for the corresponding topology and
3. the tuples representing the connections between the processors.

The tuples are intended to consist of a parent node and the according child node. For example $\{b, a\}$ is a typical tuple which denotes node a to be able to send information to node b.

Another requirement is that the root node has to be labeled a. The nomenclature of the nodes is not static, so nodes are permitted to get significant names that satisfy

the definition of an atom in Erlang as introduced in [AB, ch.2.3]. The label of the topology (the first part) as well as the algorithms are atoms.

To append new algorithms,

1. append a comma after the last topology,
2. after the comma open a new list,
3. enter the three necessary items to the list:
   (a) identifier (atom),
   (b) eligible algorithms (list of atoms) and
   (c) as many tuples as required (tuples containing two atoms).
4. close the list.

```
-define(tops,
    [
        [serial8,
            [dfs, bfs, le],
            {a,b},{b,a},
            {b,c},{c,b},
            {c,d},{d,c},
            {d,e},{e,d},
            {e,f},{f,e},
            {f,g},{g,f},
            {g,h},{h,g}],
        [parallel8,
            [dfs, bfs, le],
            {a,b},{a,c},{a,d},{a,e},{a,f},{a,g},{a,h},
            {b,a},{c,a},{d,a},{e,a},{f,a},{g,a},{h,a}],
        [complex8,
            [dfs, bfs, le],
            {a,d},{a,e},
            {b,c},{b,e},
            {c,b},{c,d},{c,e},{c,g},{c,h},
            {d,a},{d,c},{d,e},{d,g},
            {e,a},{e,b},{e,c},{e,d},{e,g},{e,h},
            {f,g},{f,h},
            {g,c},{g,d},{g,e},{g,f},
            {h,c},{h,e},{h,f}],
        [ring8,
            [dfs, bfs, le, mutex],
            {b,a},{c,b},{d,c},{e,d},{f,e},{g,f},{h,g},{a,h}]
    ]).
```

Figure 5.1: List of the topologies included

For example, a new topology might look like the following:

```
-define(tops,
    [ ...
        [RING8,
            [dfs, bfs, le, mutex],
            {b,a},{c,b},{d,c},{e,d},{f,e},{g,f},{h,g},{a,h}],

        [
        designator,
            [alg1, alg2],
            {e_i,j},...,{e_k,l}
        ]
    ]).
```

Figure 5.2: Appending new topologies

5.2 Adding new Algorithms

To expand the simulator with new algorithms a little more effort is required. Basically the simulator requires three parts:

1. one module for the calculation of the spanning tree that is used by the server application which should fulfill the naming convention *matrix_init_ALG.erl*,
2. one module for the client behavior that is used by the client application; this module should be labeled *client_algorithm_ALG.erl* and
3. one module for the calculation of the stability criteria that is used by the fault-injector. This file should be labeled *fault_injector_ALG.erl*.

Figure 4.5 allows an informative insight on the structure. To append a new algorithm, the following steps have to be taken:

5.2.1 Spanning Tree Algorithms

Open the file *matrix_init.erl*. The interesting part is shown below:

```
if
    Alg == bfs ->
        Client_Data = matrix_init_bfs:client_data(Data),
        Server_Data = matrix_init_bfs:server_data(Client_Data, []),
        Number_of_Clients = length(Client_Data);
    Alg == dfs ->
        Client_Data = matrix_init_dfs:client_data(Data),
        Server_Data = matrix_init_dfs:server_data(Client_Data, []),
        Number_of_Clients = length(Client_Data);
    Alg == le ->
        Client_Data = matrix_init_le:client_data(Data),
        Server_Data = matrix_init_le:server_data(Client_Data, []),
        Number_of_Clients = length(Client_Data);
    Alg == mutex ->
        Client_Data = matrix_init_mutex:client_data(Data),
        Server_Data = matrix_init_mutex:server_data(Client_Data, []),
        Number_of_Clients = length(Client_Data)
end,
```

Figure 5.3: Appending new algorithms

Analogously a section for the new algorithm has to be appended to this interface class. Note that the algorithms identifiers are unique.

The next step is to create the file *matrix_init_ALG.erl* which generates the required data out of the user input. The method gets a list containing all tuples as input like *[{b,a},{c,b}]*. The output has to be a list of seven-tuples containing all required information. The algorithms featured in this thesis all traverse the following steps:

- Create one entry for each processor and enter all the children in the appropriate parents list.
- Enter each processors possibility for a fault, the so called node faults (NF).
- Enter each relations possibility for a fault, the so called edge faults (EF).
- Enter each elements legitimate set of states (LS) according to the spanning tree algorithm.

INPUT:[$\{b, a\}, \cdots$]		
Step0	{b,a}	-> {a,_,_,[{b,_}],_,_,_}
Step1	{a,_,_,[{b,_}],_,_,_}	-> {a,NF,_,[{b,_}],_,_,_}
Step2	{a,NF,_,[{b,_}],_,_,_}	-> {a,NF,_,[{b,EF}],_,_,_}
Step3	{a,NF,_,[{b,EF}],_,_,_}	-> {a,NF,_,[{b,EF}],_,LS,_}
OUTPUT:[$\{a, NF, _, [b, EF], _, LS, _\}, \cdots$]		

This was the most difficult part of creating new algorithms. Yet, several values are missing, indicated by the blanks in the above table.

The format of the seven-tuple data structure reads:

{
name of the node,
node fault,
internal process id (PID),
list of children and respective fault probability,
current (initial) state,
set of legitimate states,
number of steps executed (initially 0)
}

Since the data is generated before the clients connect, the client addresses will be added automatically later. The current state will be generated in the final phase of the initialization. The last missing value will be set to zero in a later process. The variable *steps* allows the proof, that the scheduler is fair and the clients are granted almost the same amount of steps.

5.2.2 Value Obtaining Client Algorithm

The procedure is almost the same.

1. Fill in an appropriate entry in the interface module labeled *client_algorithm.erl*, then
2. create a file labeled *client_algorithm_ALG.erl*.

The client usually gets a list generated by the fault-injector with nodes that he can legally reach. If a node executes a step and experiences a node-fault, a fault-value is automatically assigned withholding the node the possibility to acquire a value that is part of the legitimate set of states.

Communication hazards also deliver a faulty-value. Yet, it is possible for certain algorithms to acquire a correct value if redundancy is available allocated by multiple nodes that can deliver a correct value.

Obviously the only task is to ask all neighbors that are on the list delivered by the fault-injector and aggregate the results.

5.2.3 Observer Specific Functions

The fault-injector has many tasks:

1. It elects nodes to commit an execution step.
2. It generates lists for traversal incorporating both node- and edge-faults.
3. Also aggregation of new states to propagates current topologies and data to be recorded is done by the fault-injector.

The implementation of an interface was disclaimed since the function consist just of one line per algorithm. The corresponding function is labeled *gen_fault(Algorithm, Data) ->* and the line to be appended reads like:

if Algorithm == ALG ->
    Fault = fault_injector_ALG:gen_fault(Data)

The file *fault_injector_ALG.erl* has to be created accordingly. The function gets the current data structure as input and delivers a falsified value that has an adequate format according to the algorithm employed. It is checked automatically whether the value equals a value from the legitimate set of states. If so, a new wrong value is generated such that a fault is always guaranteed to be a fault.

For example, *fault_injector_mutex.erl* generates integer values while *fault_injector_bfs.erl* generates lists containing a sequence of randomly chosen nodes. For this reason, observer specific functions can be recycled if possible.

Since Erlang has dynamic type assertion it is even possible to generate different types of faults within one algorithm.

5.3 Adding new Fault-Environments

The adding of new fault environments is still in experimental untested status. The reasons for a dynamic fault-environment are discussed in section 4.2.2.

Fault environments are inserted into *fault_env.erl*. Selected by the global variable *env* set in the configuration file *global_conf.hrl* the data structure can be manipulated each step affected by the variables:

1. *Algorithm*,
2. *Client_Data*,
3. *Server_Data* and
4. *Execution_Semantics*.

Even the algorithm can be changed dynamically. Projected on a meta plane, an algorithm deployed in the fault environment might choose a self-stabilizing algorithm based on the current input to follow different strategies in different sections of the execution.

The client-data is the common data-structure to be manipulated. External influences can be applied in a post-processing manner to modify not only the current configuration but also the whole topology or priority of nodes for the scheduler-election procedure.

For the parameters to be recorded an adaption of the changes applied to the former data structure is reasonable, although parameters recorded can also be generated or omitted due to individual requirements.

Finally, the maximal number of nodes that may execute at most concurrently per step can be changed dynamically, too. Although it is neither advised to use execution semantics higher than 1 nor dynamic execution semantics are reasonable, the parameter is intended for the aggregation of new data structures.

The feature of dynamic behavior of environmental influence relying on the current state is reasonable for the simulation of future scenarios. Although this feature was not used for this thesis it was implemented for the sake of expandability.

5.4 Tweaks

In this chapter several tweaks are introduced to further customize the simulator.

Emergency Brake

If the simulator is running and it has to be stopped for a reason, one way to exit the simulation is by pressing *Ctrl + c* which throws the simulator back to the console. A more gentle way is to use the *k*-function from any Erlang-shell. Just by entering

k:k().

a kill token is sent to the server which initiates an immediate shutdown sequence.

For example, if unintended values have been entered in the initialization phase, the kill-command can be used to initiate the shutdown sequence which will only exit the current simulation while the processes remain in the Erlang environment and are not thrown back to the systems console.

Negative Steps

The number of steps an algorithm has waited for execution, which is in medial equal to the number of nodes available, can be set to a negative value. Although this feature is not used, it might be reasonable to assign waiting-times a process is not ready for execution. This functionality is already implemented, tested and ready for use.

Starting in Medias Res

Batch processing is reasonable to execute a series of case studies without supervision so it is reasonable to provide routines that allow the simulator to omit initialization and start with data provided. The corresponding methods are:

- *server:start(Phase, Algorithm, Client_Data, Server_Data, GFP).*
- *client:start(Name).* and
- *fault_injector:start(phase_1).*

Although these functions were not required and not used for the evaluation of results they are helpful to acquire the results of several scenarios in little time.

Bibliography

[AB] Ericsson AB. *Getting Started With Erlang*. Ericsson. http://www.erlang.org/doc/doc-5.5.3/doc/getting_started/part_frame.html, last visited: 2006-09-25.

[Arm03] J. Armstrong. Making reliable distributed systems in the presence of software errors, 2003.

[Arm07] J. Armstrong. *Programming Erlang*. 2007.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, March 2000.

[Frä] Martin Fränzle. *Folienskript Eingebettetet Systeme 1, Teil 1h*. Universität Oldenburg.

[Hof] Petra Hofstedt. *Nebenläufige Programmierung in Erlang*. TU Berlin. uebb.cs.tu-berlin.de/projekt01/slides.I.ps, last visited: 2006-09-25.

[Mar] Andrey Markov. *Markov chain*. Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete. http://en.wikipedia.org/wiki/Markov_chain, last visited: 2006-09-25.

[Mar06] Peter Marwedel. *Embedded System Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[MP] Nick McKeown and Balaji Prabhakar. *Discrete-Time Markov Chains, Handout*. Stanford University. http://www.stanford.edu/class/ee384x/Handouts/rev3_v4.pdf, last visited: 2006-09-25.

[RA03] Mickaël Rémond and Joe Armstrong. *Erlang programmation*. Eyrolles, May 2003.

[Tri82] Kishar Shridharbhai Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1982.

[VS] Ute Vogel and Michael Sonnenschein. *Folienskript Modellbildung und Simulation ökologischer Systeme*. Universität Oldenburg. available in the StudIP e-learning platform: https://elearning.uni-oldenburg.de/.

[Wik] Wikimedia Foundation. *Markov chain.* http://en.wikipedia.org/wiki/Markov_chain, last visited: 2006-09-25.

List of Figures

4.1 DFS SERIAL8 GFP = 0.2 . . . 13
4.2 Static Fault Environment . . . 15
4.3 Dynamic Fault Environment . . . 15
4.4 Compiler Warnings . . . 19
4.5 SiSSDA Class Diagram . . . 20
4.6 Server Application Step1 . . . 21
4.7 Initialization Step1 . . . 22
4.8 Initialization Step2 . . . 22
4.9 Server Application Step2 . . . 22
4.10 Initialization Step2.5a . . . 23
4.11 Initialization Step2.5b . . . 23
4.12 Server Application Step 3 . . . 24
4.13 Initialization Step3 . . . 24
4.14 Initialization Step4 . . . 24
4.15 Client Application Step1 . . . 25
4.16 Server Application Step4 . . . 25
4.17 Fault Injector Step1 . . . 26
4.18 Server Application Step4 . . . 26
4.19 Execution step0 . . . 26
4.20 Execution step1 . . . 27
4.21 Execution step2 . . . 27
4.22 Execution step3 . . . 27
4.23 Execution step4 . . . 27
4.24 Execution step5 . . . 28
4.25 Execution step6 . . . 28

5.1 List of the topologies included 32
5.2 Appending new topologies 33
5.3 Appending new algorithms 34