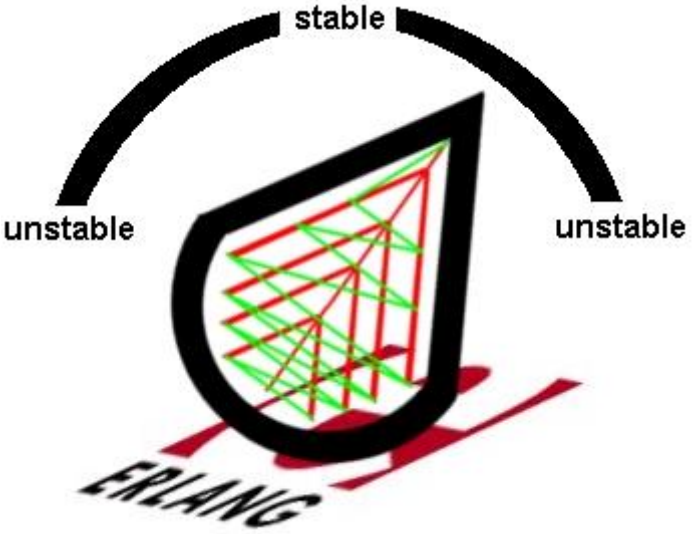
Simulation of Self-Stabilizing Distributed Algorithms to Determine Fault Tolerance Measures



Faculty II
Department of Computer Science

Group System Software and Distributed Systems

Carl von Ossietzky Universität Oldenburg

by
Nils Müllner

Supervisor: Prof. Dr.-Ing. Oliver Theel
Co-Supervisor: Abhishek Dhama, MSc CS&CE

Date of Submission: April 10 2007
Rev 1.02 May 03 2007

Abstract

Fault tolerance measures such as reliability and availability are employed to select the most suitable fault tolerant system to be deployed under a given environment. Although such measures have been defined for masking fault tolerant systems, until recently they were not defined for non-masking fault tolerant systems. In [War06], a procedure has been outlined to determine the reliability, instantaneous availability and limiting availability for self-stabilizing distributed algorithms[Dol00] using Markov-chains. The procedure utilizes a method similar to predicate abstraction to reduce the state space of a self-stabilizing system and derives the Markov-chain representing the abstracted system. However, assumption about fault-propagation and approximations introduced by the abstraction technique hinder the accuracy of the measures obtained. Simulation of the distributed algorithms can be used to determine more accurate transition probabilities and hence can be used to fine tune the Markov-chains representing the abstraction technique. Such a simulator can also facilitate the study of the variation of reliability and availability due to fault profile of the environment.

This work is concerned with the development of a simulator which can simulate a self-stabilizing distributed algorithm's behavior under transient faults. It also provides a mechanism for fault injection along with facility to vary the error probability distribution. The simulator is written in the purely functional concurrent language *Erlang* and provides the possibility to record measures which can be fed to external tools for further analysis. Erlang was chosen for its abilities in distributed concurrent computing. The simulation results are used to verify the metrics obtained from the analysis procedure described in [War06]. This work provides insights as to how to improve the fault tolerance metrics of a given self-stabilizing algorithm based on the results from the simulations. Also the fine tuning of the analysis procedure based on feedback from the simulator is available. A complete user manual of the simulator is enclosed with this thesis.

I hereby certify that the work presented in this thesis is my own and that work performed by others is appropriately cited.

Ich versichere hiermit, diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfsmittel bedient zu haben.

Signature of the author:

Oldenburg, April 09 2007

Contents

1. Introduction

1.1 Motivation

Distributed systems are gaining more and more importance because increase of computing power is achieved easier by using multiple components instead of rising one components quality. Not only for large-scale projects[1] but also for desktop computers where recent development indicates an increase of the number of cores featured per chip [WR03] the gain of resources is coupled to the increase of components.

Since the number of elements is proportional to the possibility of the occurrence of faults and distributed systems consist of more elements than single-core systems, the possibility for the occurrence of faults rises. To cope with faults, fault-tolerance is a required feature that is provided by self-stabilization.

In a recent research-project at the University of Oldenburg an approach has been developed to analyze fault-tolerance-measures in self-stabilizing distributed non-masking fault-tolerant systems utilizing a method similar to predicate abstraction to reduce the state space of a self-stabilizing system and to derive the Markov-chain representing the transitions [War06]. As the trade-off reduces the accuracy of the results, it is reasonable to simulate the behavior of self-stabilizing distributed systems.

1.2 Problem Specification

The approach discussed in the paper [War06] introduces an abstraction technique that makes assumptions about fault-propagation and approximations. These reduce the accuracy of the values obtained as described in the paper. To acquire fault-tolerance measures with a certain degree of accuracy it is reasonable to simulate an appropriate environment and observe its behavior.

In this thesis, a *Simulator for Self-Stabilizing Distributed Algorithms* (SiSSDA) is developed and four algorithms are simulated to be evaluated in significant scenarios. SiSSDA is implemented in the purely functional concurrent language *Erlang*

[1] http://distributedcomputing.info/

and can be executed in both distributed and local environments to determine fault-tolerance measures. Erlang was chosen because it was developed for distributed applications featuring platform independence, high reliability (nine nines [Arm, P.29]), it is runtime-efficient and facilitates rapid prototyping since it is code-efficient, too.

SiSSDA offers not only flexibility provided by important parameters like dynamic fault-environments. It is also built for expansion and is ready to cope with future algorithms, topologies and environments.

To acquire proper results using the simulator, a manual is enclosed where implementation details are discussed. For users that are already familiar with the programming language used (Erlang) as well as the distributed algorithms featured and fault-tolerance measures and self-stabilization, a quick-start guide will contain all information needed to use the simulator.

A full version containing all the information and background covered not only in this thesis but the simulator is also provided in the manual. Since the simulator is highly flexible, a large number of parameters are available to achieve the detailed execution of case studies.

To compare the results derived with the analytical method described in [War06], significant scenarios have been chosen to be simulated as case studies. A scenario consists of the algorithm and the topology chosen as well as fine-tuning of the parameters provided by the configuration as described in section 4.2 in the manual.

1.3 Thesis Outline

First, the system model is introduced in section 2.1. After applying self-stabilization in section 2.3 the case studies used for the simulator are presented in section 2.4 using the system model.

To present the analytical background, fault-tolerance measures like availability and reliability are discussed in section 3.1. After introducing discrete-time Markov-chains in section 3.2, an example shows the utilization of a system that is reduced to a spanning tree using a self-stabilizing algorithm. Following, the appropriate Markov-chain is derived to to calculate fault-tolerance measures.

Markov-chains have been used in [War06] as an important tool and are required for the analytical approach.

Chapter 4 presents the simulator and gives background knowledge. Nevertheless, the important information on how to use the simulator and its functioning is provided in the enclosed manual.

The results acquired with the simulator gathered in several hundred executions are presented in chapter 5. Although complex scenarios were chosen that were not feasible for the analytical approach, the assertions given by the results can be compared.

Looking at the approach, the simulator and the results discussed in this thesis, chapter 6 concludes the outcome and delivers a short outlook on possible further ways for further research.

2. Self-Stabilizing Distributed Algorithms

This chapter starts with an example for a self-stabilizing distributed system that was introduced by Shlomi Dolev, one of the leading scientists in the field of self-stabilization [Dol00]. After the example, the self-stabilizing system model is introduced.

Using this background, the case studies required for the simulation are presented featuring the four self-stabilizing algorithms *Depth-First Search*, *Breadth-First Search*, *Leader Election* and *Mutual Exclusion* as well as significant topologies.

Example

As a simple but yet demonstrative example for self-stabilizing distributed systems Dolev instances a scenario, in which an orchestra tries to play on a windy day outdoors without a conductor. Due to the wind each member can only hear her direct neighbors. Dolev presents various versions to lead to one self-stabilizing solution for the orchestra. Each player has a score according to her instrument for the same piece of music. After finishing with the score, players will start over again. Faults may occur since the wind may change the pages individually which players would not notice and by this the musicians might loose synchrony.

First Scenario If one musician hears that one of her neighbors is playing a different part of the score, she can decide to synchronize with this status or not. The problems are:

- If rule is to change to the part of the score one of the neighbors is playing, there is a decision-problem if both neighbors are at different parts of the score. Even majority-consensus methods might fail since it is possible that one musician and her direct neighbors all play different parts of the score.
- If the eyed musician and one of her neighbors are playing different parts of the score, both could change according to each other, being in an asynchronous state again. This would lead to stuttering behavior.

So further measures are required to achieve self-stabilization.

Second Scenario If asynchrony is detected, it is reasonable to restart at a predesignated part of the score, for example the beginning of the whole score. It might also be reasonable later on to define certain sections in which different behavior is aspired. In this scenario one possible behavior might be to restart the least common movement.

- The problem is that only direct neighbors will restart at a certain state if they detect local asynchrony. The next neighbors will realize the restart too late, thus restarting again with their neighbors which will lead to a global stuttering situation again.

Third Scenario One self-stabilizing strategy is, that every player has to join her neighbor who plays the earliest part, if the page is earlier than her own page.

- It is always determinable if each player is in synchrony with her neighbors and if she has to change according to the neighbor's status.
- There is not the possibility to hang in a stuttering loop in the absence of further faults since the earliest stage is always the valid state and a flip-flop always requires two valid states.
- As there is always one player with the least page-number, all members will converge to this player's state until synchrony is reached if no further transient faults occur. If further transient faults occur, synchrony will still be achieved setting the least page-number.

If one period in which no transient fault happens is long enough, the players will be in synchrony again and if no further transient faults happen all players will reach the end of the score simultaneously.

This example presents a solution, how distributed systems, in which transient faults might occur, use self-stabilization to regain an operational status. The orchestra is a self-stabilizing distributed system. It's efforts to regain synchrony in presence of transient faults can follow different strategies. The outcome of these strategies, measured as mean time to loose synchrony and mean time to regain synchrony, can be used for comparison.

2.1 The System-Model

According to [Dol00], the system model has to be abstract, such that one model can represent different settings like

- communication networks,
- multiprocessor computers or
- multitasking single computers.

2.1.1 Communication

The system is defined as a set of n processors which are connected by m communication channels with $n, m \in \mathbb{N}$. The processors are referred to by their index so the ith processor is denoted as P_i and the communication channels are referred to as message queues with the first identifier being the sending processor and the second identifier referring to the receiving processor such that $q_{i,j}$ references the communication channel from process P_i to P_j. There are two common ways for communication.

- Shared Memory: Processors share a common set of memory with read- and write-restrictions to propagate their current status.
- Message Passing: A message is sent via a common channel.

These techniques works also using uni- and bidirectional links. While in the shared memory, model processes use shared communication registers. One processor writes to a resource with exclusive right to write while another process may only read from that resource. Analogously for the message passing model, P_i can send a message to P_j, while P_i is unreachable to P_j.

As elaborated in [Dol00], a configuration

$$c = (s_1, s_2, \cdots, s_n, q_{1,2}, q_{1,3}, \cdots, q_{i,j}, q_{n,n-1}) \tag{2.1}$$

is defined as the set of all states of all processors, where s_i is the state of processor P_i, and the message queues $q_{i,j}$ with $i, j \in \mathbb{N}, i \neq j$. A queue consists of messages sent by P_i to P_j that were not received yet. To model the shared memory model in an appropriate way, the registers use a first-in-first-out (FIFO) strategy.

2.1.2 Temporal Model

The time-model used is important since the speed of the processors in a distributed system may vary and time required for message passing is not static. Hence, this nondeterminism is likely to *result in totally different state transitions of processors from identical initial states* [Dol00].

The model used in this thesis features discrete steps in which only one single processor may execute one step that consists of one computation step and one communication operation that may be either sending (writing) or receiving (reading). The model using these atomic steps is called *interleaving model*.

Each communication step may lead to a change of the state of the processor that executed a computation step. Note that although this thesis only covers atomic computation steps in an arbitrary interleaving fashion, SiSSDA is also able to cope with concurrent execution of computation steps as discussed in section 2.3.2.

A computation step will be referred to as *step* in the following. A step, denoted as a_i, is defined as transition between two configurations c_i and c_{i+1} with $c_i \xrightarrow{a_i} c_{i+1}$. In the interleaving model a sequence of configurations and transitions is called execution

$$E = (c_1, a_1, c_2, a_2, \cdots) \tag{2.2}$$

2.1.3 Properties

An execution E is

- *live*, if each process is executed infinitely often when the choice over the set of transitions is executed infinitely often (that means, there is no *starvation* as there is no *deadlock*),
- *fair*, if, for every transition, if it is enabled infinitely often, it is taken infinitely often and
- *safe*, if there is no violation of safety conditions [Lyn96].

Examples

- Liveness: if the attribute of liveness is violated, there is at least one process that cannot execute. A possible reason might be, that the process is waiting for a resource but never gets a grant to use it because another process blocks it. The consequence is called starvation.
- Fairness: In a fair execution no processor is favored and processors are assigned equally for execution.
- Safety: in an unsafe environment deadlocks are possible. For a deadlock, several preconditions are required:
  - Mutual Exclusion: A process has exclusive right to a resource.
  - Hold and Wait: While waiting for further required resources a process waits keeping its exclusive rights for resources gathered so far. This is also called *busy waiting*.
  - No Preemption: Once granted, resources must not be withdrawn from a process by a higher instance.
  - Circular Wait: If a chain exists consisting of processes and resources in which one process holds a resource another process requires and vice versa, a deadlock exists.

One popular example that depicts these attributes of distributed systems is the *dining philosopher's problem*

It was first introduced by Tony Hoare after Edsger Dijkstra used a similar question on a synchronization problem where five computers compete for the access to five tape-drives earlier in 1971 [Dij02].

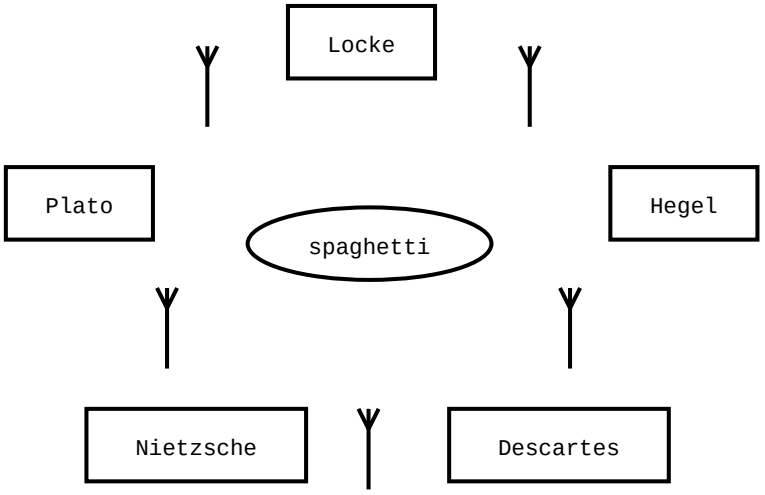


Figure 2.1:
Five dining philosophers sharing five forks while two adjacent forks are required to get access to the common resource spaghetti.

In this model five philosophers meet for lunch. Philosophers are only capable of eating or thinking. Since these philosophers are not communicative, they do not each other. The philosophers sit down at a round table. Five forks are on the table as shown in figure 2.1. A philosopher requires two forks at a time to eat. While a philosopher is thinking, he puts the forks aside, one to his left and one to his right.

If a philosopher is busy eating, his direct neighbors have to wait until he has finished and their particular next neighbors are not eating as well. When a philosopher has finished his meal, he starts thinking until he gets hungry again.

- One major issue to depict is that a philosopher may take a fork if possible when he gets hungry. In that case, *deadlocks* are possible since a situation might occur where all philosophers get hungry at the same time, leading to a situation where each of them holds one fork. In that case, nobody can eat but everybody is waiting for their neighbors to finish. This is called a violation of the attribute of *safety*.
- Another problem of concurrency this metaphor depicts is *starvation*, which is a tellingly pun in this case. If there is one philosopher with two adjacent neighbors from which at least one is eating all the time, this philosopher will starve over time. This is a violation of the attribute of *liveness*.

The model is not-preemptive, since no one can force a philosopher to put a fork back if he has one or two in his hands. The model is fair, since every philosopher has the same chance to eventually get two forks if he gets hungry.

The implementation of both schedulers implemented in the simulator are

- fair,
- live and

- safe

and the results are measured with a number of steps that is sufficiently large (see also section 4.2.1 in the manual).

2.1.4 Graphs

A distributed system can be abstracted as graph when the processors can be mapped to nodes and the communication channels or shared memory respectively can be mapped to edges. To keep up with identifiers introduced so far, vertices, processors and nodes will be referred to as P_i and edges, shared memory, message passing and communication channels will be referred to as message-queue $q_{i,j}$ as defined in section 2.1.

2.1.1 *A graph G can be defined as a pair (V, E), where V is a set of vertices, and E is a set of edges between the vertices $E = \{(u, v)|u, v \in V\}$. If the graph is undirected, the adjacency relation defined by the edges is symmetric, or $E = \{\{u, v\}|u, v \in V\}$ (sets of vertices rather than ordered pairs). If the graph does not allow self-loops, adjacency is irreflexive.[BT]*

One reasonable way to define graphs are *adjacency matrices*. An adjacency matrix of a finite graph of n vertices is a $n \times n$ matrix. The entry $a_{i,j}$, which is the entry is the i^{th} row and j^{th} column of the adjacency matrix, refers to the number of edges from vertex i to vertex j. The diagonal from a_{00} to a_{nn} of the matrix refers to the number of edges that are self-pointing to each vertex.

For the thesis, a finite simple graph is used where the adjacency matrix is a $(0, 1)$-matrix.

The topologies featured in the thesis are also presented as adjacency matrices and can be used as example (section 2.4.5).

Obviously, matrices can be represented with less information while using strict bidirectional topologies since the values can be mirrored at the diagonal from $[1, 1]$ to $[n, n]$ as shown in the following picture:

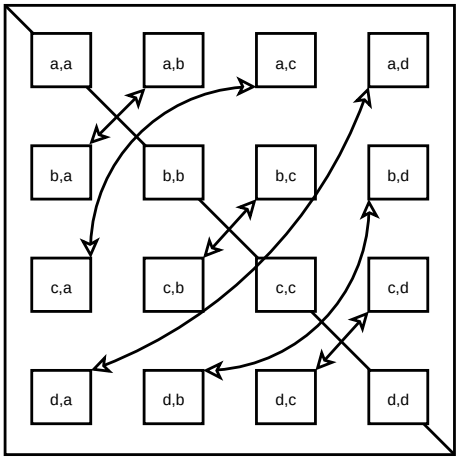


Figure 2.2:
This picture shows the redundancy of information in a matrix that features strictly bidirectional edges.

The algorithms require different kinds of graphs, though. While *Depth-First Search*, *Breadth-First Search* and *Leader Election* require the graph to be in the format of a tree, which is a graph in which any two vertices are connected by exactly one path, the graphs used for *Mutual Exclusion* for example are limited to the format of a ring, which also is a legitimate tree.

The graphs used in the thesis for simulation are described in section 2.4.5.

2.2 Distributed Algorithms

A distributed algorithm is defined to be a collection of local algorithms from the same copy one for each processor. Every processor independently executes its local algorithm and cooperates with the other processors to achieve a certain objective. [WCWH03, Ch.5]

The vertices described in the previous section are abstract for nodes that are a part of a distributed environment as mentioned in section 2.1. The single components are communicating and exchanging data while processing the data propagated may lead to a change of the local status of a node executing a step according to the algorithm executed.

2.2.1 Example

The following example shows, how a system is used to derive an appropriate spanning tree.

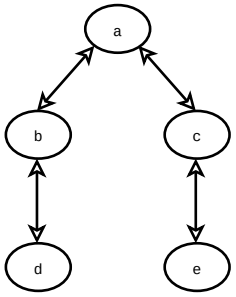


Figure 2.3: This graph is used in the following example.

The graph shown in Figure 2.3 models a distributed system. The appropriate adjacency matrix is defined as:

$$M = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

In a distributed environment like this many algorithms are feasible to be executed. The algorithms used for the thesis are discussed in section 2.4 in detail.

The algorithm executed defines the spanning tree that is derived using the given tree. One feasible algorithm for the tree shown in Figure 2.3 is *Depth-First Search* as presented in section 2.4.2. The appropriate spanning tree this distributed algorithm builds is shown in Figure 2.4.

2.2.1 *A spanning tree is a connected, acyclic subgraph containing all the vertices of a graph.[BT]*

By contrast to the original graph, the original edges are replaced by new edges depending on the algorithm executed. The *minimum spanning tree* for the given example, using the system shown in Figure 2.3 and the Depth-First Search Algorithm presented in 2.4.2, is shown in the following figure:

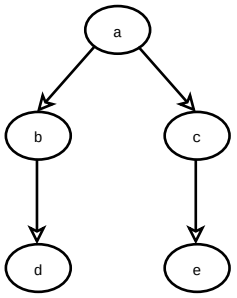


Figure 2.4: This is a minimum spanning tree of the previous example.

In a minimum spanning tree, only edges required are used, so that every node has at least one connection to the rest of the graph. Using a graph with weighted edges, the minimum spanning tree prefers the edges with the smallest values.

Spanning trees can not only be defined by means of minimal size. It is also possible to build spanning trees according to algorithms as elaborated in 2.4. In such a spanning tree, nodes update their status by communicating with their neighbors if they are granted to execute a step. Nodes as well as edges of the spanning tree have certain possibilities to fail. Even though there are different types of failure like

- communication is not possible if recipient is unreachable,
- communication is not possible when dispatcher is unreachable or
- communication delivers wrong values since values committed are corrupt.

Every failure is considered critical for *self-stabilizing distributed fault-tolerant systems*. Systems can be classified as *masking fault tolerant* and *non-masking fault-tolerant* systems.

2.2.1 *Fault Tolerance: This is the ability of a system to deliver desired level of functionality in the presence of faults, i.e., instead of preventing faults from occurring, one tries to tolerate their effects. To achieve this, the system should be able to detect and/or correct errors in the system.[Jhu]*

While masking fault-tolerance conceals transient faults from the user, non-masking fault-tolerance can be used to determine fault-tolerance measures.

In contrast to masking fault-tolerance, this thesis' emphasis is on non-masking fault-tolerance and therefore consequences by failures are entirely taken into account.

Until now, we have an abstract system model, that can also be defined by its adjacency matrix, a spanning tree that relies on algorithms that are described later, and faults corrupting the system. In the next section the aspect of self-stabilization discussed, leading toward the scenarios observed in the thesis.

2.3 Self-Stabilization

Self-stabilization was first introduced by Edsger Dijkstra (1930-2002), a dutch computer scientist who received the A. M. Turing-Award. The intention is to design a distributed system that can be started in an arbitrary state and still converge to a desired behavior [Dol00].

These systems are designed to work in an environment where transient faults can happen. A *transient fault* corrupts the state of the system by corrupting message channels or shared memory while the behavior of the system remains unchanged[1]. The term transient implies that faults occur for a limited time and do not persist. While the fault ceases to influence the state of a processor, the system may stabilize autonomously and regain a legitimate state.

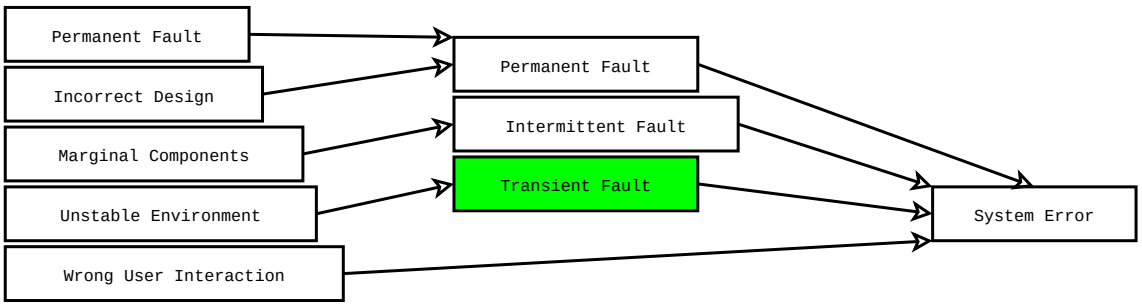


Figure 2.5:
The fault model in reference to [Tz] shows types of faults. By this, transient faults can be distinguished from other types of faults that are disregarded in the thesis.

Due to occurrence of transient faults the system will reach an arbitrary state that might not be part of the specified set of legal states so it gets corrupted. The system is self-stabilizing because it is designed to converge to a desired behavior from any arbitrary state in a finite amount of time. That means, it regains a state that is part of the specified legal set of states within a finite amount of time.

Following the remarks by Marco Schneider, a distributed self-stabilizing system S is *the union of the local states of its components* [Sch93]. In contrast to the definition of the configuration c by Dolev [Dol00] as wrapped in section 2.1.1, the definition of a system by Schneider does not explicitly take message queues into account. As the algorithms featured in the simulator do not require message queues since the execution semantics are limited to one and by this messages are processed instantaneously, the more abstract definition of a system S by Schneider is sufficient for this thesis.

2.3.1 Definition

2.3.1 *We define self-stabilization for a system S with respect to a predicate P, over its set of global states, where P is intended to identify its correct execution. S is self-stabilizing with respect to predicate P if it satisfies the following two properties:*

[1] http://pierre.ici.ro/ici/revista/sic1998_4/art01.html

1. *Closure: P is closed under the execution of S. That is, once P is established in S, it cannot be falsified.*

2. *Convergence: Starting from an arbitrary global state, S is guaranteed to reach a global state satisfying P within a finite number of state transitions [Sch93].*

Convergence is also referred to as *reachability* in contemporary literature [CD94, P.21]. This means, that the state space for a system S is divided into legitimate states that satisfy P and illegitimate states that do not satisfy P. P depends on the algorithm employed. The predicate P used in this thesis is quite simple. If every node is in its legal configuration according to the predefined spanning tree, the system S satisfies P. If one or more nodes are in an illegitimate state, neither the global predicate P is satisfied.

As this thesis does not cover dynamic topologies (although the simulator is ready to cope with), further definitions of self-stabilization regarding dynamic patterns as described in [Sch93] are not taken into account.

2.3.2 Execution-Semantics

The discussion of *execution-semantics* is mandatory as it plays an important role for further research and and deeper understanding for the simulator. For the practical issues please refer to the manual section 4.3.2.

The execution semantics, or *overlapping semantics* [AFG93, Ch. 2.2], alleges the number of nodes that are at most allowed to execute per step. While featuring *maximal parallelism* allows all feasible processes to execute when ready, *serialized semantics* as used for the simulation, allow execution to one process per step as defined in [AFG93, Ch. 2.1].

If we define a graph g according to section 2.1.4 consisting of

- vertices P_i and
- edges $e_{i,j}$

and if the legal state of each node $s(n_i)$ is specified as $s_i = 1$ and the illegal state as $s_i = 0$ where i is referring to the appropriate node, for each step the state of the system is represented as system state $S_k = \{s_0, \ldots, s_{n-1}\}$, with k being the number of the step and n being the number of nodes.

There are 2^n possible configurations since each node can be either in the legal set of states or in the illegal set of states and there are n nodes. Each state represents one possible configuration, starting with every node being in the illegal set of states $S = \{0, 0, \cdots, 0\}$ until the binary counting leads to a configuration in which every node is in the legal set of states $S = \{1, 1, \cdots, 1\}$.

Furthermore, the possibility to reach system state S_j from a system state S_i which now depends on the behavior of the nodes, the scheduler's behavior and the waiting-time of each node, can be calculated. As the simulator features only an arbitrary interleaving behavior in which at most one node may execute in one time-step, only

one bit representing the nodes status in the configuration can flip if the executing node changes its status.

With the set of all possible configurations $\Gamma = S_0, \ldots, S_{2^n-1}$ (according to [Kok05]) and therefore all appropriate transition possibilities $\Pi = [\pi_{i,j}]$ that are feasible for possible steps a as defined in section 2.1.2, only system states are within range, that differ at most in one bit from the originating configuration. While regarding Markov-chains the term *Hamming-distance* is commonly used to refer to the maximum range within the Markov-chain per step as discussed later in section 3.2, the term *execution semantics* is used referring to the system model.

Although atomic steps are simulated and by this, concurrent execution of tasks is prevented, further analysis might require higher execution semantics to deal with concurrency. The following figure exemplifies the impact of execution semantics. As the 2.1 example already demonstrated, using execution semantics greater one brings several issues that have to be regarded.

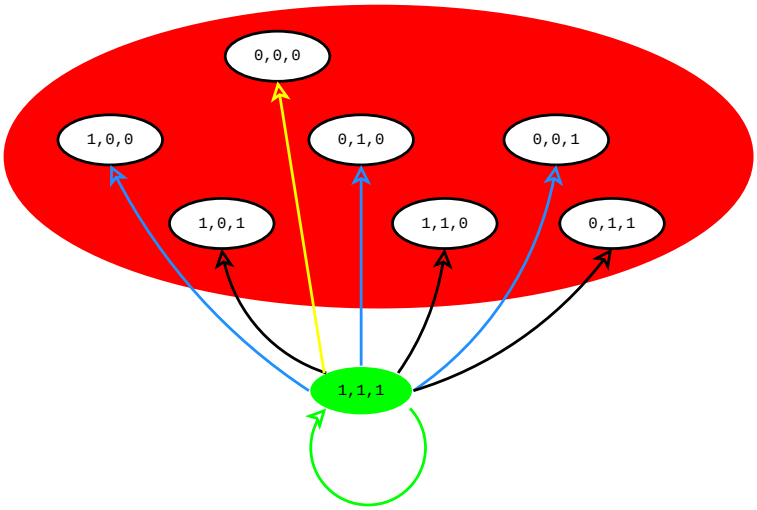


Figure 2.6: Serialized semantics in a graph featuring three processes.

From the state $S = \{1, 1, 1\}$ only four other states are reachable within a hamming distance of one (black arrows), while with a hamming distance of two, seven states are reachable (blue arrows and black arrows). Since the system S consists of three processes P_1, P_2 and P_3, the maximal reasonable value for execution semantics is three. Within execution semantics of the size of the system any configuration is within range from any other configuration.

Defining all possible configurations presents the state space. By the exact definition of the state space it is possible, to distinguish the illegal states from the legal set of states. As already indicated in the previous figure, a system is in the predefined set of legal states if each process P_i satisfies the predicate P as defined in section 2.3.1.

2.3.3 Using Self-Stabilization

Self-stabilization brings many advantages. Not only that systems can converge to a legitimate state under the occurrence of transient faults, but they can also start

in arbitrary states and since omit an initialization phase. The simulator supports both features, since the flag *init_arbitrary* implements both behaviors (please refer to section 4.2.6 of the enclosed manual). To measure the self-stabilization property, so called *fault-tolerance measures*, it is reasonable to introduce the terms

- availability,
- reliability and
- limiting availability.

Furthermore, Markov-chains are introduced to model the theoretical background of the simulator. It is reasonable, to first define the case studies and present several scenarios, consisting of topologies and algorithms, in section 2.4, to have a scenario ready to exemplify fault-tolerance measures and Markov-chains in chapter 3.

2.4 Case Studies

To compare fault-tolerance measures that are introduced in section 3.1, different scenarios are presented. On the one hand, different topologies are compared in section 2.4.5. Each topology is represented by an 8×8 matrix.

On the other hand, each topology is thoroughly simulated using different strategies to define the appropriate spanning tree according to the algorithm used. As a constraint for the denotation of the nodes, there are only two restrictions:

- the identifiers consist of one atom (as defined in chapter 5.1 in the manual) and
- the root node is identified as a.

For the topologies implemented as case studies, descriptors were chosen to be the first eight letters from the alphabet. However, it might be reasonable to choose different descriptors for future topologies.

The spanning trees defined by the algorithms are built as example with topology COMPLEX8 that is described in subsection 2.4.5. While the nodes in the graph are connected with black arrows, red arrows indicate the traversing sequence by the spanning tree.

Please note that there is a difference in the spanning tree algorithm and the root- and client-node value assignment sub-protocols as discussed in [CDK99]. As for the topologies, only the spanning tree algorithms are important, the value assignment is presented just for the DFS algorithm to discuss aspects of self-stabilization.

2.4.1 Self Stabilizing Breadth First Search Spanning Tree Algorithm

The *Self Stabilizing Breadth-First Search Spanning Tree Algorithm* algorithm (BFS) builds a spanning tree in which the nodes are visited according to their minimal distance to the root node a (please also refer to [SS92] and [AB98]).

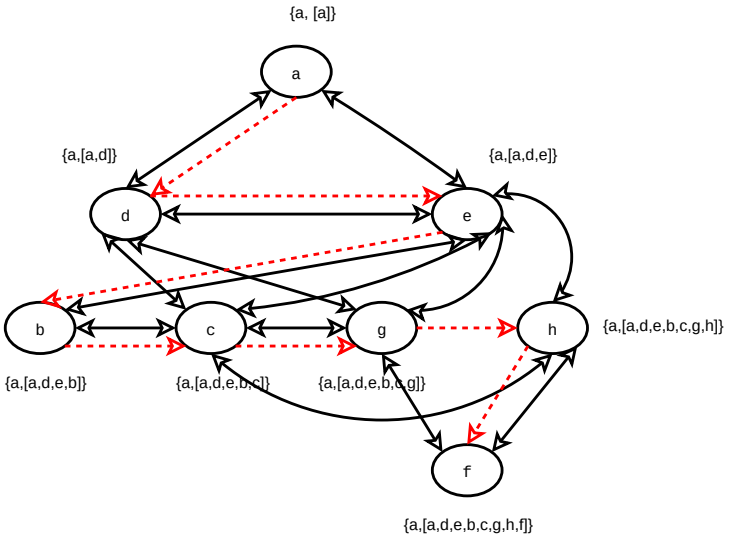


Figure 2.7:
This illustration shows the operation method of the Breadth-First Search Algorithm

As input, a graph G and the name of the root node, that is predefined as a, are given to the algorithm.

```
Root: do forever
          for m := 1 to δ do r_im :=< 0, 0 >
      od
Other: do forever
          for m := 1 to δ do lr_mi :=read(r_mi)
          FirstFound := false
          dist := 1 + min(lr_mi.dis | 1 ≤ m ≤ δ)
          for m := 1 to δ
          do
              if not FirstFound and lr_mi.dis = dist − 1 then
                  write r_im :=< 1, dist >
                  FirstFound := true
              else
                  write r_im :=< 0, dist >
          od
      od
```

Figure 2.8: BFS Spanning Tree Algorithm

The BFS spanning tree algorithm in reference to [Dol00, P.13] allocates the minimal distance to each node. Traversing through each level, nodes are ordered by their identifier, such that, according to the given topology, the algorithm performs the following steps:

1. The algorithm is executed and the value obtaining sub-algorithm first initializes root node a.

2. Node a has two children, d and e, which are visited according to their ID (first d, then e).

3. d and e have four children [b,c,g,h] (equivalent to the grandchildren of node a) which are visited according to their ID. Being a child of a set of nodes means, that at least one node of the specified set of nodes is a parent (e.g. b is a child of e and not of d). Yet having multiple parents is of course possible (e.g. c is a child of both, d and e). Note that having multiple parents grants no privileges such as benefiting in priority.

4. Since f is the last unvisited node and f has the greatest minimal distance to node a, f is the last node visited.

It is interesting, that two successive nodes do not have to be directly connected. Having the same minimal distance to a common ancestor is a sufficient criteria to be visited consecutively such as nodes g and h in the example.

BFS has a time complexity of $O(b^m)$ and a space complexity of $O(b^m)$.

Self-stabilization requires each node to hold the whole topology ready. For example, node h updates the local status by reports from nodes c, e and f although the direct predecessor defined by the spanning tree is g. Since there is no connection between g and h, BFS algorithm requires all nodes to maintain the whole topology which is transmitted with every update operation such that each node can generate its new status even if its predecessor is not directly connected.

To prove the correctness of self stabilization in reference to [SS92, P.4], three steps have to be considered:

1. The legitimate state of a graph represents the BFS spanning tree. The proof follows the BFS spanning tree algorithm:
   - Root node has a level of 0.
   - Since root propagates its level, all children set their level to 1.
   - Repeating this argument for all nodes except those initialized delivers the BFS spanning tree.
2. If the system is in a legitimate state, no node will change its value.
3. In any illegitimate state at least one node will update so some action is always guaranteed while the system is not in a legal state.

Lemma 2.4.1 *In any illegitimate state there exists at least one privileged node, i.e. in any illegitimate state some action is always guaranteed. [SS92, P.5]*

As continued in [SS92, P.8], the system is guaranteed to reach the predefined legal set of states within a finite number of steps in the absence of further faults. For the complete derivation please refer to [SS92].

2.4.2 Self Stabilizing Depth First Search Spanning Tree Algorithm

The *Self Stabilizing Depth-First Search Spanning Tree Algorithm* algorithm (DFS) also builds a spanning tree. Starting in the root-node, always the child with the smallest identifier is visited first. When a node has no children, backtracking is used and siblings are visited.

The following example shows the proceeding of DFS on the previously mentioned topology COMPLEX8:

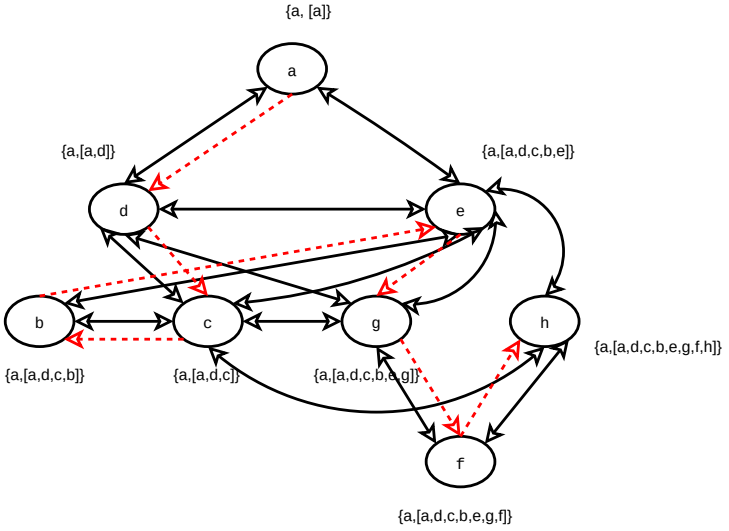


Figure 2.9:
This illustration shows the operation method of the Depth-First Search Algorithm

There are two important papers that concentrate on the DFS algorithm:

- The paper *Self-Stabilizing Depth-First Search* introduces the self-stabilizing depth first search spanning tree algorithm. The DFS algorithm is proven to be self-stabilizing by induction [P.4], yet it only covers the spanning tree algorithm.

- The paper *Self-Stabilizing Distributed Constraint Satisfaction* on the other hand differs between the spanning tree algorithm and the *Value Assignment Sub-Protocol* [CDK99, P15]. The authors Zeev Collin, Rina Dechter and Shmuel Katz first introduce a spanning tree algorithm similar to the one discussed in [CD94] and give evidence for self-stabilization by proving convergence with induction [CD94, P.13].
  Following they present the value assignment sub-protocol [CD94, P.20] and prove self-stabilization by verifying reachability and closure (please also refer to 2.3.1).

With reference to [CD94, P.4], self-stabilizing Depth-First Search is given as

root P_1:
    do forever
        $path_1 := \perp$
    od
non-root P_i:
    do forever
        for $j := 1$ to δ do $read_path_j := read(path_j)$od
        $writepath_i := \min\{|\ read\_path_j \circ a_j(i)\ |_N, suchthat 1 \leq j \leq \delta\}$
    od

Figure 2.10: DFS Spanning Tree Algorithm

Lemma 2.4.1 *The graph traversal mechanism is self-stabilizing with respect to the set of legally controlled operations.[CDK99, Sec.3.4.3]*

Self-stabilization can be verified by induction. In contrast to BFS, the proof does not list the generations that are traversed, but the distinct branches. The example presented in figure 2.9 indicates that for topology COMPLEX8 only one branch is traversed. Spanning trees that have a maximum branching factor of 1 obviously do not require backtracking.

According to topology COMPLEX8 the algorithm performs the following steps:

1. The algorithm is executed and the value obtaining sub-algorithm first initializes root node a.

2. Since the current node a has two adjacent nodes d and e, the one with the lesser id is chosen such that node d is the next node.

3. Following, the smallest unvisited neighbor is chosen, which of course is c within the set of adjacent nodes [a,c,e,g].

4. The smallest unvisited neighbor of node c is node b, which is followed by

5. e. Note that although e is directly connected to node a, DFS first searches all the nodes mentioned above before node e is visited.

6. The smallest neighbor of e is g.

7. Although f has the greatest minimal distance of 3 to the root node a, it is visited after node e due to a smaller id than node

8. h, which is the last node visited.

It is important that the algorithm memorizes the nodes already visited. Otherwise loops are possible that lead to a deadlock as demonstrated in the following figure.

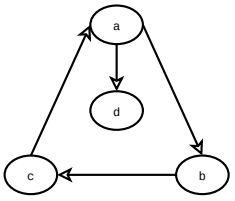


Figure 2.11: DFS requires memory to prevent deadlocks

The algorithm has a time complexity of $O(b^m)$ and a space complexity of $O(b \cdot m)$ with b being the maximal branching factor and m being the maximal path length. Note that $O(b^m) = O(|V| + |E|)$. Even though DFS is nether optimal nor complete since infinite paths and cycles are possible, this algorithm is eligible to determine fault-tolerance measures. Popular derivatives of DFS are

- *iterative deepening DFS*, which is a combination of DFS and BFS and is complete, and
- *depth-limited search*, which is complete within limitation even for cyclic and infinite graphs[2].

Obviously, backtracking is not needed in this topology in contrast to topology *PARALLEL8* presented in section 2.4.5, where node a has to be revisited each time before a new node can be visited. The appropriate sequence defining the spanning tree for topology PARALLEL8 is

$$a\text{->}b\text{->}a\text{->}c\text{->}a\text{->}d\text{->}a\text{->}e\text{->}a\text{->}f\text{->}a\text{->}g\text{->}a\text{->}h.$$

If we set a to be part of an infinite branch containing only nodes with IDs smaller than c (such as a_1, a_2, ...), obviously the rest of the tree will will never be reached. This means, that DFS is not complete for topologies with infinite branches.

2.4.3 Self Stabilizing Leader Election Spanning Tree Algorithm

The *Self Stabilizing Leader Election Spanning Tree Algorithm* algorithm is important for many protocols used in distributed systems. For example, in Blue Tooth networks a leader has to be elected first before communication is possible. To maintain communication even in fault-hazardous environments, dynamic patterns are required to cope with the occurrence of transient faults.

[2] www.cis.upenn.edu/~matuszek/cit594-2002/Slides/tree-searching.ppt

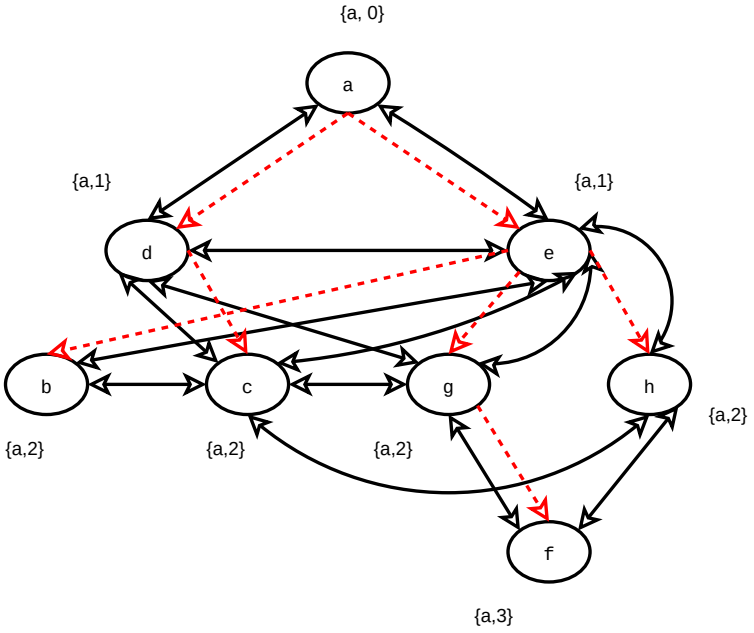


Figure 2.12:
This illustration shows the operation method of the Leader Election Algorithm

P_1:
    $path_1 := \perp$;
P_i(i $\neq$ 1):
    For all edges e incident on v do ->
        receive leader, path,
        Set with smallest leader-id,
        tuple with shortest path from set with smallest leader-id

Figure 2.13: Leader Election Spanning Tree Algorithm

The system is in the predefined set of legal states, if every node accepts the root node as leader and the minimal distance to the root node is provided in the write register. If a node gets corrupted or its predecessor is not reachable, it will become leader itself propagating its new role until it updates again. If this corrupted node executes a step again, it might stabilize if the node is not compromised and it eventually gets a link to root node by its predecessor.

A node will always accept the proposed leader with the lowest id as new leader and if more than one neighbor propagates the same leader, it will accept the shortest path to the leader, incremented by one. For example, root node will always propagate the tuple $\{a, 0\}$ since its id is a and the shortest path to itself has the length 0. If the root node gets corrupted, though, it will propagate a wrong id.

For certain scenarios referring to stability measures it might also be interesting to define sub-trees to be in stable state if the whole sub-tree agrees on one leader that is root node of the sub-tree while the root node of the sub-tree does not have to accept the root node of the tree as leader. For comparison of the algorithms, stability is achieved if every single node is in the predefined set of legal states.

The interesting comparison is between leader election and the next algorithm, mutual exclusion. For a serialized execution semantic as used in this thesis, both algorithms fulfill the same task as discussed in the following section.

Since the legitimate state of each node is built from the length of the shortest path to the root node as also featured for BFS, the proof for self-stabilization can be adopted.

2.4.4 Self Stabilizing Mutual Exclusion Spanning Tree Algorithm

The *Self Stabilizing Mutual Exclusion Spanning Tree Algorithm* algorithm (MutEx) was first introduced by Edsger Dijkstra in 1962 [Ala03]. As outlined in the dining philosophers example 2.1, nodes ordered in a ring compete for a common resource. Each step, one node is chosen by the scheduler. If the predecessor of the node has a value not equal its own, it will copy the neighbors value and execute one step. Otherwise it will not do anything. The predecessor of the root-node is the last node in the ring. If root-node is granted to take a step, it compares its own value with the predecessors value. In contrast to all other nodes, root will only execute the step, if the own value is equal to the predecessors value. The new value is the neighbors value incremented by one modulo the number of nodes in the ring plus one.

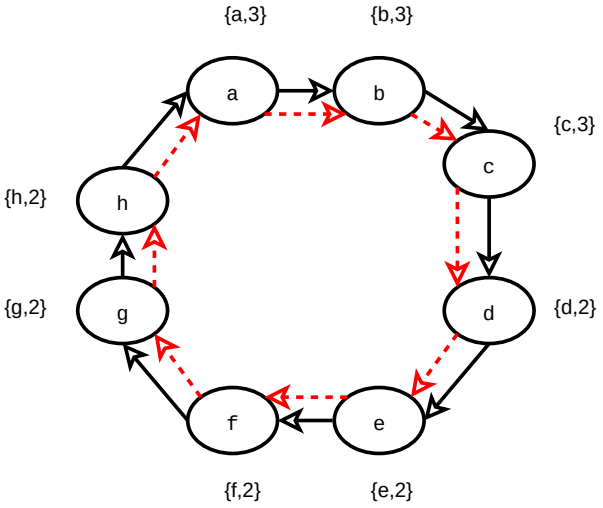


Figure 2.14:
This illustration shows the operation method of the Mutual Exclusion Algorithm

P_1:
    do forever
        if $x_1 = x_n$ then
            $x_1 := (x_1 + 1) mod (n + 1)$
$P_i (i \neq 1)$:
    do forever
        if $x_i \neq x_{i-1}$ then
            $x_i := x_{i-1}$

Figure 2.15: Mutual Exclusion Algorithm in reference to [Dol00, P.17].

The system is in the predefined set of legal states, if only one is privileged. One node is privileged, if

- all nodes have a value smaller than the number of nodes connected in the graph and
  - all nodes have the same value or
  - nodes $P_1 = a$ to P_i have a value x with $i, x \in \mathbb{N}; i, x \leq \sum_{i=0}^{n} P_i$ and nodes P_{i+n} to P_n have a value of $x - 1$ with $i, x \in \mathbb{N}; i, x \leq \sum_{i=0}^{n} P_{i+1}$ or
  - nodes $P_1 = a$ to P_i have a value 0 with $i \in \mathbb{N}; i \leq \sum_{i=0}^{n} P_i$ and nodes P_{i+n} to P_n have a value of $\sum_{i=0}^{n} P_i$.

Comparing leader-election algorithm with mutual-exclusion algorithm, both feature almost the same strategy for a ring-topology and serialized execution semantics. While leader-election is in the legal set of states when exactly one leader is elected, mutual exclusion is in the legitimate set of states when exactly one processor is eligible to execute one step.

As the example given by Dijkstra features a concurrent execution of processes, the simulator only executes processes consecutively. By this, deadlocks cannot influence the systems behavior. The only results measured are affected by fault-propagation and self-stabilization.

The leader election protocol stabilizes within $O(\Delta\mathcal{D})$ where

- Δ is the maximal degree of a node and
- $\mathcal{D}$ denotes the diameter of the graph

as presented in [DIM97]. Since the time required to reach an legitimate configuration is predictable, mutual exclusion is obviously self-stabilizing.

2.4.5 Topologies

The algorithms described above were tested on four different topologies. Each topology consists of eight nodes that are connected in different ways. To compare the impact of different orderings, extreme settings have been chosen.

SERIAL8

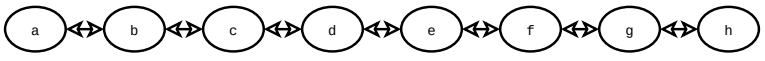


Figure 2.16: Topology SERIAL8

The topology *SERIAL8* features eight nodes that are connected in a serial fashion. Each node can only communicate with it's direct neighbors. The appropriate adjacency matrix is defined as

$$M_{SERIAL8} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Since nodes are connected like the musicians in the orchestra example, this leads to the expectation that faults have the possibility to propagate through all nodes. Besides, each node, except for the first and the last node, depends on two neighbors. The first and the last node only depend only themselves and their particular neighbor.

PARALLEL8

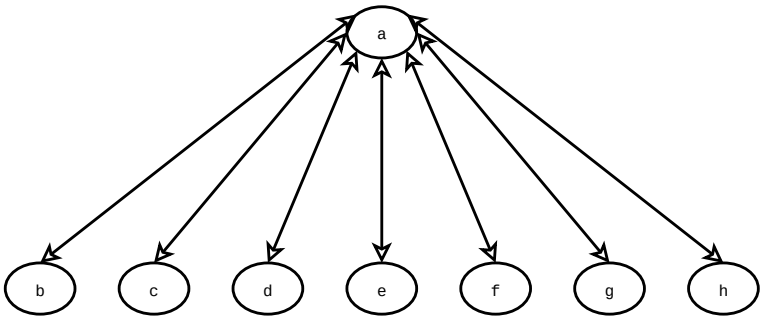


Figure 2.17: Topology PARALLEL8

The topology *PARALLEL8* on the other hand features maximal parallelism so both extremes are covered. In this topology seven nodes are connected with one root node while they can not communicate with each other. Compared to the orchestra example, each musician would only be able to get in synchrony by looking at the conductor on a foggy and windy day while all the musicians are deaf.

The topology is defined by the appropriate adjacency matrix

$$M_{PARALLEL8} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Each node relies on the root node. On the one hand, faults cannot propagate through all nodes since the root node does not rely on communication. On the other hand, all nodes rely on one root node neutralizing the redundancy provided by a second neighbor.

COMPLEX8

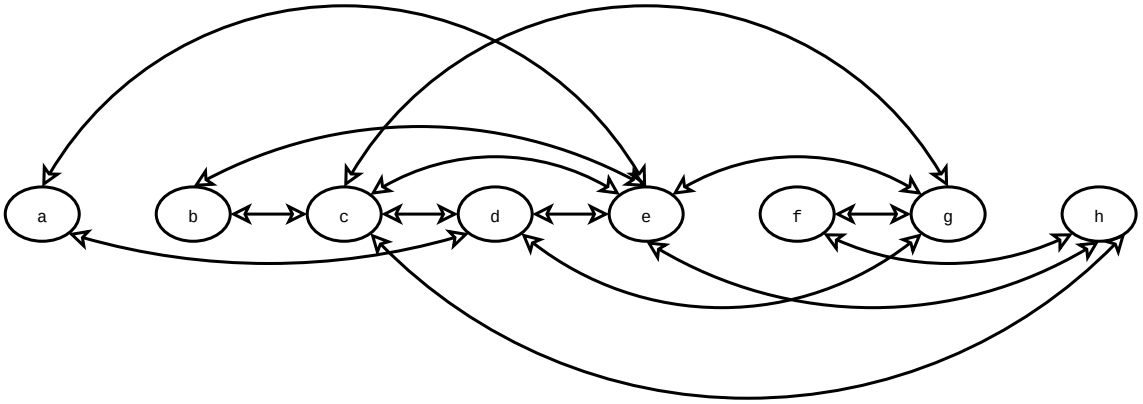


Figure 2.18: Topology COMPLEX8

With *COMPLEX8* a complex topology is introduced to simulate an environment that features a high complexity. This topology also features bidirectional links and can be expressed as a matrix

$$M_{COMPLEX8} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

With the topology COMPLEX8 a complex setting is simulated. Comparing the number of edges, SERIAL8 and PARALLEL8 only feature seven edges each. COMPLEX8 on the other hand features 13 edges. Although faults may propagate better in this environment, redundancy is likely to compensate or even reduce effects of fault-propagation.

Regarding the algorithms, this is the topology that has different spanning trees for BFS and DFS. While the topologies presented so far build identical spanning trees for both BFS and DFS algorithms to observe differences in fault-tolerance measures influenced only by the algorithms and not by the spanning tree, COMPLEX8 intends to demonstrate the influence of the spanning tree.

RING8

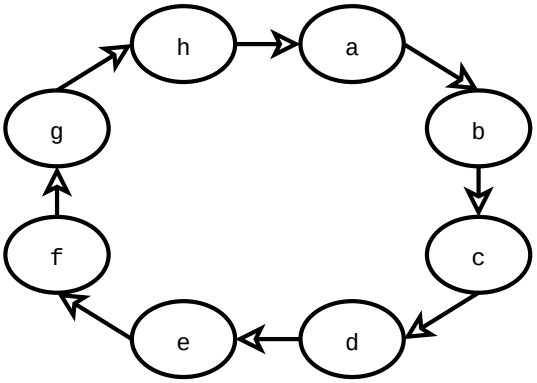


Figure 2.19: Topology RING8

With *RING8* a simple topology is introduced to simulate an environment that is also feasible for the *Mutual Exclusion* algorithm.

$$M_{RING8} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

This setting is also famous as *token-ring* or *round-robin*. Faults are likely to propagate and redundancy it not available since each vertex relies on its predecessor.

This topology is required by the MutEx algorithm since it requires a token-ring topology. As a ring suffices the criteria to be also a legal tree, all other algorithms featured are feasible for this topology, too.

Complexity Issues

Obviously, predictions about fault-propagation and redundancy can be made referring to the given adjacency matrix. Since this is part of the theoretical background elaborated in [War06], the relation between the complexity of a given matrix and the impact on fault-propagation and redundancy is only briefly discussed.

The linear complexity of a matrix can be calculated with

$$f_i = \sum_{i=0}^{n} a_{i,j} x_j \tag{2.3}$$

where $M = a_{i,j}$ is the associated adjacency matrix and x_i is the indeterminate for every P_i in S as described in [DLN06, Sec. 2.3].

The complexity is proportional to both fault-propagation as well as redundancy. Again, the definition of the system model forces the effect on the results. The model

used in the simulator assumes that faults are distinguishable from correct values. Furthermore, the parameter *faulty_fault* discussed in the manual in section 4.2.5 rectifies the assumption of nodes preferring false values in presence of true values. For this reason, the influence of fault-propagation is supposed to be lower than the advantages of redundancy.

The complexities of the featured topologies are

Topology	Complexity
SERIAL8	14
PARALLEL8	14
COMPLEX8	28
RING8	8

3. Fault-Tolerance Measures and Markov-Chains

After discussing the aspects of self-stabilization, fault-tolerance measures have to be introduced to measure the quality of different self-stabilizing algorithms under various environments. Subsequently Markov-chains are introduced to facilitate a suitable theoretical background for the system model.

3.1 Fault-Tolerance Measures

Since stability is required to grant a high degree of *reliability* and *availability* of services in distributed systems, the concept of *self-stabilization* is required

- to keep costs for maintenance low due to the ability to regain a legal operational state autonomously,
- to keep required time for maintenance low (referred to as *Mean Time to Repair*, see also 3.1.1) to maximize the time the system is operational and
- to grant a certain level of fault-tolerance measures such as availability and reliability.

As discussed in the abstract of the thesis, this simulator has been implemented to determine non-masking fault-tolerance measures. By collecting data about the fault-tolerance measures of distinct scenarios that are simulated, the quality of self-stabilization can be determined.

3.1.1 Availability

As this thesis focuses on the discrete-time model, availability can easily be calculated with

$$A = \frac{E[uptime]}{(E[uptime] + E[downtime])} \tag{3.1}$$

where uptime is the number of steps the system was in a specified set of legal states and downtime means the number of steps the system was not in the specified set of legal states. To give a more concrete equation, it is reasonable to introduce abbreviations that are usually used in this field. The preceding equation can also be expressed as

$$A = \frac{MTTF}{MTBF} \tag{3.2}$$

The abbreviations are discussed in the following table:

MTTF	$= avg < TTF_i >$	Mean Time to Failure	is the medial time for which an element is operable. This can be estimated by field studies for each element. This is also known as „medial uptime“.
MTTR	$= avg < TTR_i >$	Mean Time to Repair	is the medial time a faulty element needs for stabilization. This is also known as „ medial downtime“.
MTBF	$= avg < TBF_i >$	Mean Time Between Failure	is the medial for one cycle, starting operative, operating till a fault occurs, continuing with repair measures until operating state is regained.

Obviously, availability is the ratio of the value of the observed uptime of a system to the aggregate of the observed values of up- and downtime. Therefore

$$MTBF = MTTR + MTTF.$$

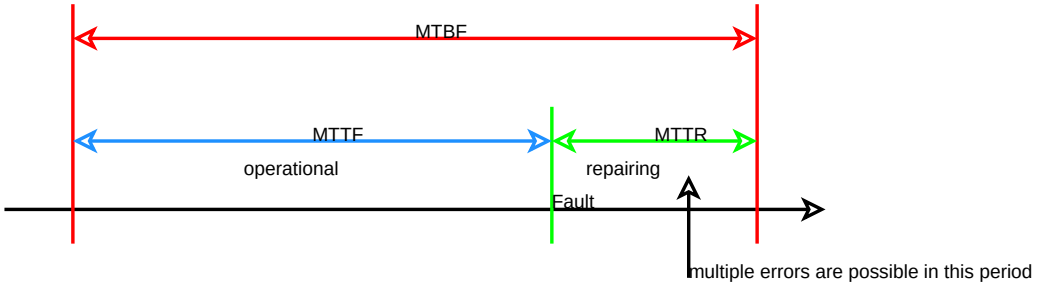


Figure 3.1: This graph shows the connection between MTTF, MTTR and MTBF

Another analogous definition is given by the IEEE:

3.1.1 *Availability is the degree to which a system or component is operational and accessible when required for use[oEE90].*

In our case the terms *operational and accessible* are referred to *a specified legal set of states*. If in a sequence of executions the configuration of the system is in a legal state for each step, the equations result is 1 since

$$A = \frac{1}{(1+0)} = 1 \tag{3.3}$$

If a sequence of executions results the configuration of the system to be only in illegal state, the equation's result is 0 since

$$A = \frac{0}{(0+1)} = 0 \tag{3.4}$$

Availability Classifications

Availability can be further classified:

- instantaneous (or point) availability,
- average up-time availability (or mean availability),
- steady state availability,
- inherent availability,
- achieved availability and
- operational availability.

According to [Rela, Ch.5], the types of availability are defined as follows:

- *Instantaneous availability* is the probability, that the system is in the legal set of states at a random time t. In contrast to reliability, instantaneous availability also regards information about the systems maintainability represented by the term $m(u)$:
  $A(t) = R(t) + \int_0^t R(t-u)m(u)du$
- *Average uptime availability* is the proportion of time a system was in the predefined set of legal states:
  $\overline{A}(t) = \frac{1}{t} + \int_0^t A(u)du$
- *Steady state availability* or limiting availability is the limit of the instantaneous availability as it converges to infinity:
  $A(\infty) = \lim_{t\to\infty} A(t)$
- *Inherent state availability* is the type that is featured discussed above and used for comparison in section 5.1.
  $A_I = \frac{MTTF}{MTBF}$
- *Achieved state availability* uses also preventive maintenance downtime which is not used in the thesis. With $\overline{M}$ as mean maintenance downtime it is computed by
  $A_A = \frac{MTBM}{MTBM+\overline{M}}$
- *Operational state availability* is the availability measured regarding all sources of downtime.
  $A_O = \frac{Uptime}{OperatingCycle}$

For further information, please refer directly to the discussion of the observed measures in section 5.4. For this thesis, the results presented are limited to inherent state availability A_i. The other subclasses are implicitly covered, like limiting availability with regards to accuracy (see also Ch. 4.2.1 in the manual), or can be disregarded like instantaneous availability, where the maintainability factor is not covered by the simulator.

3.1.2 Reliability

The systemic definition for *reliability* is:

3.1.1 *Reliability is the ability of a system or component to perform its required functions under stated conditions for a specified period of time [oEE90].*

In other words: it is desired to calculate the probability of a system to operate for a certain amount of time without failure [Relb]. As the *time to failure* and the *time to repair* (explained in subsection 3.1.1) may vary from turn to turn, it is reasonable to calculate with the particular mean times.

To visualize reliability for discrete systems, a graph is drawn showing the amount of time the system runs without failure on the x-axis and the according probability on the y-axis. For many systems this results in a Gauss error distribution curve.

Let

- $\lambda = \frac{1}{MTTF}$ be the failure-rate,
- $\mu = \frac{1}{MTTR}$ be the repair-rate and
- $\nu = \frac{1}{MTBF}$ be the medial fault frequency,

and by this follows accordingly with equation 3.2

$$\nu = \frac{\lambda \cdot \mu}{\lambda + \mu} \qquad [\text{Gei}, P.9] \tag{3.5}$$

While availability is depending on the simulated time, reliability is driven by the number of failures. Reliability can be calculated with the following equation:

$$R = e^{-(\lambda \cdot t)} = e^{-(\frac{t}{Q})} = e^{-N} \tag{3.6}$$

where t is the execution time, $Q = \frac{1}{\lambda} = MTTF$ and N being the number of failures during the mission.

Dynamic fault-environments as described in section 4.2.2 in the manual might demand the use of the *Weibull distribution*

$$R = e^{-(\frac{t}{h})^b} \tag{3.7}$$

where h is the characteristic *age-to-failure-rate* and b is the *Weibull shape factor* as discussed in [ADI03]. Since the simulator uses only static fault-environments in the scenarios executed, although dynamic behavior is also supported, the former equation is used.

3.2 Markov-chains

In 1907, the Russian mathematician Andrey Markov began with his research in the calculus of probabilities. Nowadays, Markov-chains have a wide field of application. They are used in environmental sciences, in mathematics like stochastic and miscellaneous fields of computer science [Frä, VS].

To introduce Markov-chains, Markov-processes are discussed first which lead to discrete-time Markov-chains. As discrete-time Markov-chains are sufficient for the simulator, continuous-time Markov-chains can be disregarded. Following, the probability mass function is presented to introduce the probability distribution.

The execution E as defined in section 2.1.2 can obviously be represented as a Markov-chain. Markov-chains define a set of states $S = \{s_1, s_2, \ldots, s_n\}$. The process starts in one of these states as the simulator starts in an arbitrary state. For each state i there is a probability p_{ij} to reach state j. The process moves from the initial state successively from one state to another. Each move is called a step, even if $i = j$, which means that the process reaches the same state after executing a step [GS97].

The state-space is defined in reference to section 2.3.2 as set of all reachable states. For the given system-model every state that is in the set of possible states is reachable.

A graph G consists of $n \in \mathbb{N}$ vertices, referred to as processors P_i. Each processor either satisfies the predicate P or not, denoted as $P_i = 0$ if P_i is in an illegitimate state and $P_i = 1$ if P_i is in a legitimate state.

Accordingly, there are 2^n combinations for the union of the local states of the components of a system S. Since these are the possible states and all possible states are reachable, all these system states from $S_0 = \{0, 0, \cdots, 0\}$ to $S_{2^n-1} = \{1, 1, \cdots, 1\}$ are reachable.

One way to model system behavior are Markov-chains. However, it is reasonable to introduce Markov-processes first, which are discrete-time stochastic processes.

In a *Markov-process* the future behavior of the system depends only on the current state and not on the history. The future states depend on the current state.

3.2.1 *A stochastic process $\{X(t)|t \in T\}$ is called a Markov-process if for any $t_0 < t_1 < t_2 < \ldots < t_n < t$, the conditional distribution of $X(t)$ for given values of $X(t_0), X(t_1), \ldots, X(t_n)$ depends only on $X(t_n)$:*

$$P[X(t) \leq x | X(t_n) = x_n] = P[X(t - t_n) \leq x | X(0) = x_n] \tag{3.8}$$

[Tri82, p.296]

As the simulator features a discrete *state space I* as already discussed in section 2.3.2, the system can be modeled as a *Markov-chain*. Furthermore, as the simulator also features a discrete *parameter space T*, the Markov-chain that models the behavior of the simulator is a *discrete time Markov-chain* (DTMC) [Tri82, p.337]. This means, further observations are not required to cope with *continuous-time Markov-chains* (CTMC) [Tri82, p.405].

Consequently, the state of a system is observed at discrete set of time points. According to equation 3.8 follows:

$$P(X_n = i_n | X_0 = i_0, X_1 = i_1, \ldots, X_{n-1} = n-1) = P(X_n = i_n | X_{n-1} = n-1). \tag{3.9}$$

Thus, future states depend only on the current state and are independent of states previous than the current state.

3.2.1 Probability Mass Function

Let S_{sample} be the countable sample space with $S \subseteq \mathbb{R}$. Accordingly, the probability mass function (PMF) is

$$f_x(x) = \begin{cases} Pr(X = x) & \text{for} \quad x \in S, \\ 0 & \text{for} \quad x \in \mathbb{R} \backslash S \end{cases} \tag{3.10}$$

For example, a coin toss with 50% probability for each side to be facing the observer leads to the following PMF:

$$f_x(x) = \begin{cases} \frac{1}{2} & \text{for} \quad x \in \{0, 1\}, \\ 0 & \text{for} \quad x \in \mathbb{R} \backslash \{0, 1\} \end{cases} \tag{3.11}$$

For Markov-chains this means that for the PMF of the random variable $p_j(n)$ the according equation is:

$$p_j(n) = P(X_n = j). \tag{3.12}$$

Since the future states depend on the current state, the conditional PMF is required which is given by:

$$p_{jk}(m, n) = P(X_n = k | X_m = i), 0 =\leq m \leq n, \tag{3.13}$$

where $p_{jk}(m, n)$ denotes the probability, *that the process makes a transition from state j at step m to state k at step n* [Tri82, p.337].

3.2.2 Mathematical Representation

$p_{jk}(m, n)$ is also called the *transition probability function* (TPF) of the Markov-chain. Since the simulator only features homogeneous Markov-chains and $p_{jk}(m, n)$ is only depending on the difference $n - m$, $p_{jk}(m, n)$ is static. This means, the transition probabilities do not change (depending on the step). Although the simulator uses static (or stationary) transition probabilities in this thesis, it is implemented to be capable of dynamic environments, too, as discussed in section 4.2.2 in the enclosed manual.

The transition-space of a Markov-chain can be depicted with a matrix:

$$P = [p_{ij}] = \begin{bmatrix} p_{00} & p_{01} & \cdots & p_{0n} \\ p_{10} & p_{11} & \cdots & p_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n0} & p_{n1} & \cdots & p_{nn} \end{bmatrix}$$

While index i references state s_i as current current state, index j references state s_j as following state. Accordingly states have a probability of p_{ij} to reach the following state. Note that $i = j$ is possible which means, that although a step is executed the configuration does not change (with regards to the message queue as discussed in section 2.3). Since the current state has to be left each step, the sum of all outgoing edges representing the transition probabilities adds up to 1 such that the sum of all elements of each row $p_{i0} + \ldots + p_{in}$ of the matrix is 1. The sum of all incoming edges $p_{0i} + \ldots + p_{ni}$ represented by the matrix' columns has a value greater or equal 0. If the sum of all probabilities of all incoming edges equals 0, the state cannot be reached except as initial state. Since the whole state space, regarding the system model used in this thesis, is reachable due to transient faults, the sum of the probabilities of the incoming edges for each distinct node must be greater 0.

Obviously, the originating state can be regained after executing a step. These self-targeting edges are represented by the diagonal from p_{00} to p_{nn} denoted in the previous matrix.

3.2.3 Markov-Chains representing Self-Stabilizing Algorithms

To return to self-stabilization, the proofs for self-stabilization mentioned in section 2.4 can be directly applied to the appropriate Markov-chains.

Lemma 3.2.1 *In a system S consisting of $n \in \mathbb{N}, n > 0$ processors $P_1, \cdots, P_n$ and unidirectional connections $e_{i,j}$ such that each processor reads at least from one other processor, in every fair execution, the probability to converge to the legal set of states is greater zero: $p > 0$.*

Proof:

1. If the system is in a legitimate state (that is $S_{2^n-1} = \{1, 1, \cdots, 1\}$) and no transient faults occur, the corresponding TPF claims that the corresponding line in the matrix representing the transition space sums up to 0. Obviously, once a legitimate state has been reached and no faults occur, the system will remain in the legitimate state in the absence of transient faults.

2. If the system is not in the predefined legitimate state, in every fair execution one node will execute a step. Each execution leads to a new state after a finite amount of time. In every possible sequence of executions E the legitimate state S_{2^n-1} is reached after a finite amount of time. The system converges to the predefined legitimate state and by this is self-stabilizing.

As discussed in 2.3.2, it is not important, whether the sequential execution of processors is arbitrary interleaving or concurrent. Considering different semantics like maximal parallelism only effects the number of bits that are able to flip per step. Yet, being in the illegitimate set of states forces the system to execute steps until the legitimate state is reached, disregarding the execution manner.

3.2.4 Graphical Representation

The topologies featured in this thesis consist of eight processors each. Since the state space is $2^8 = 256$ accordingly, the Markov-chains representing the topologies consist of 256 elements.

The following table indicates the behavior for larger execution semantics regarding the number of featured processors:

Exec. Sem./#Nodes	2	3	4	5	6	7	8
0	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8
2	1	3	6	10	15	21	28
3		1	4	10	20	35	56
4			1	5	15	35	70
5				1	6	21	56
6					1	7	28
7						1	8
8							1
SUM:	4	8	16	32	64	128	256

While the columns represent the total number of processors featured, the rows represent the number of nodes selected by the daemon. Obviously it is not reasonable to use higher execution semantics than number of nodes since no more processors can execute than processors are existing in the topology.

The entries are the gain from one value for execution semantics to the next higher value. For example, with an execution semantics of 0, a node can only reach itself. Using serialized execution semantics it can reach all states that differ in one bit. Since the length of the tuple containing the bits is equal to the total number of processors in the graph, a node can reach as many states as are nodes in the graph plus its originating state.

The last row equals the states reachable from each single state using the maximal applicable value for execution semantics. To calculate the overall number of edges featured in the appropriate Markov-chains, this value has to be multiplied with 2 since the weighted edges are unidirectional because probabilities differ for the ways forth and back and also the value has to be multiplied with the number of processors used as indicated in the top row. For example, a topology featuring eight nodes requires using maximal execution semantics of eight requires a Markov chain with 256 states and 4096 unidirectional edges.

Each entry can be calculated with the following recursive algorithm:

```
routine(Entry_{i,j})
        Entry_{i,j} = subroutine(0,Entry_{i-1,j}),
        Entry_{i,j}.
subroutine(Result,Entry_{i,j})->
        Value = routine(Entry_{i,j}),        New_Result = Result + Value,
        New_Entry_{i,j} = Entry_{i-1,j-1},
        if
                New_Entry_{i,j} == 1 ->
                        Result + 1;              true ->
                        subroutine(New_Result,New_Entry_{i,j}).
```

Figure 3.2: Calculation of the reachability matrix.

This property of Markov-chains is also known as *Pascal's triangle*.

Bernoulli Process

A Bernoulli process is a discrete stochastic process and a special case of Markov-chains. Let p be the possibility that a particular incident happens, e.g. at a coin toss that the coin shows head or at a dice throw that the dice shows an even number as shown in equation 3.11. In both cases the possibility for each event is $p = 50\% = 0.5$. So the counterpossibility is $q = 0.5$ accordingly since the sum of all possibilities is 1 as stated in chapter 3.2.2.

A Bernoulli process delivers a countable infinite sequence of results. Let $X = 1$ denote success and $X = 0$ denote failure. Further let $q = 1 - p$ be the counter possibility with which a failure will occur such that q is the possibility that the system will converge to a specified legal set of states. The process can be described by a sequence of random numbers $\{X_1, X_2, X_2, \ldots\}$.

Obviously, a sequence of steps executed, defined as execution E in section 3.2 by the simulator has analogue features as a *Bernoulli Process*.

3.3 Conclusion

In the following, a simple topology is introduced and all introduced techniques are applied as done for the featured topologies. To cope with the high degree of complexity as discussed in section 3.2.4, a simple topology is used for demonstration.

Topology

The topology used consists of three nodes a,b and c that are connected bidirectional such that a can communicate with b and a can communicate with c but b and c have no common communication channel.

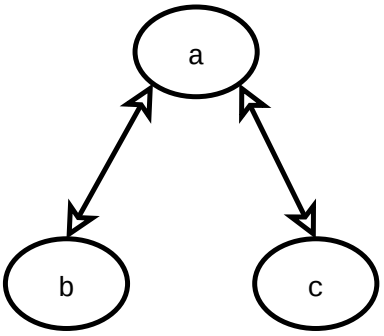


Figure 3.3: The graph representing the exemplified topology.

$$M_{example1} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Algorithm

Since the example does not satisfy the requirements of mutual exclusion as it is not in the shape of a unidirectional ring, one of the three algorithms left has to be chosen. As DFS was the first algorithm implemented, it is chosen for this example.

The spanning tree is represented by the following graph:

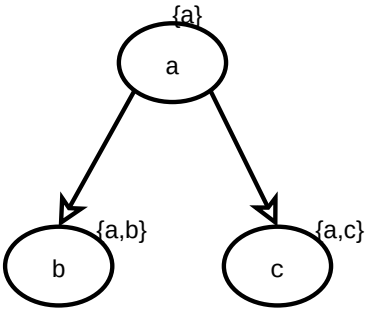


Figure 3.4: The graph representing the exemplified DFS spanning tree.

Markov Chain

The graph consists of three nodes. According to section 3.2 the appropriate Markov-chain has $2^3 = 8$ states

$S_i = \{a, b, c\} = \{P_1, P_2, P_3\}$

$S_0 = \{0, 0, 0\}$
$S_1 = \{1, 0, 0\}$
$S_2 = \{0, 1, 0\}$
$S_3 = \{1, 1, 0\}$
$\dots$
$S_7 = \{1, 1, 1\}$

The first bit represents the status of the node with the lowest ID, the second bit represents the bit with the second lowest ID and so on. If a processor P_i is in a legitimate state as shown in the figure above (i.e. a has the value $\{a\}$, b has the value $\{a, b\}$ and c has the value $\{a, c\}$) it is represented with the bit set to 1. Otherwise the bit is set to 0.

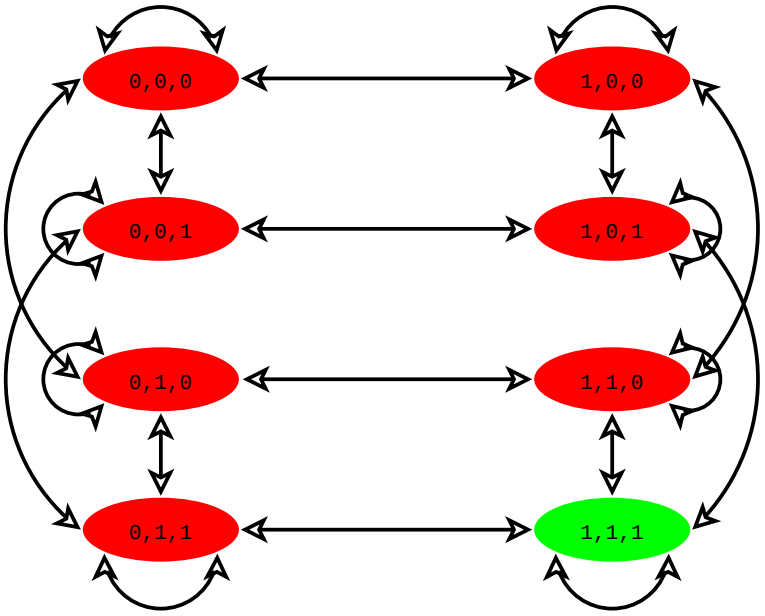


Figure 3.5: The Markov-chain representing the example.

The green state is the legitimate state where all local states satisfy the predicate P. Since execution semantics are set to 1, only states that differ in maximal 1 bit are reachable. Following table 3.2.4 in reference to column 2, each state can reach itself (column 2 row 1) and three neighbors (column 2 row 2). Setting the execution semantics to 2 (column 2 row 3), three more neighbors would become reachable. The appropriate graph is not presented since it is hard to view as a whole.

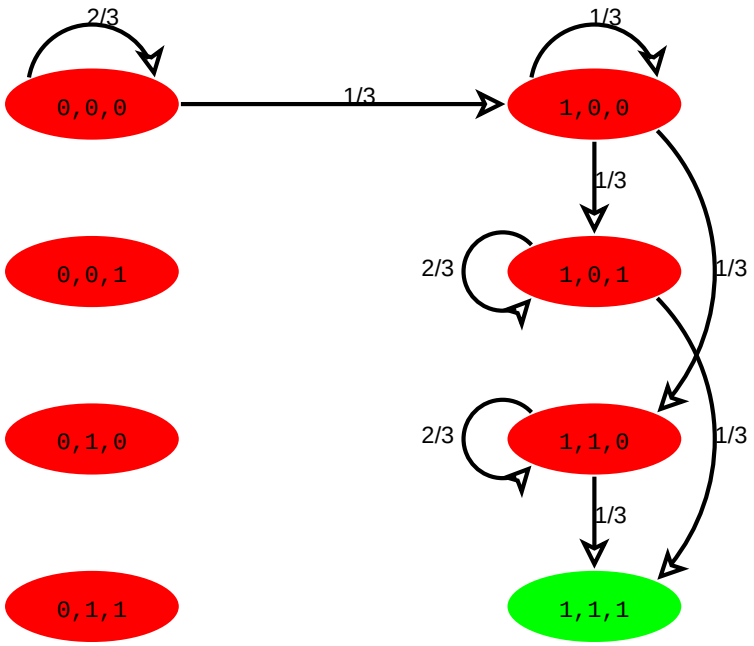


Figure 3.6:
The Markov-chain representing the example disregarding fault-probabilities.

To present a valid Markov-chain, fault-probabilities have been disregarded to exemplify self-stabilization. Furthermore it is assumed, that the system starts in state $S_0 = \{0, 0, 0\}$.

Obviously, the sum of the weights of all outgoing edges is always 1. Furthermore, several states are unreachable in this model. Since nodes b and c rely on node a, the root node has to acquire the legitimate state first. Chances are $\frac{1}{3}$ since each node has the same possibility to execute a step. After a is granted an execution step it stabilizes and global state reaches state $\{1, 0, 0\}$.

From now on, two paths are possible to reach the legitimate state. Either b executes a step and stabilizes or c is granted execution and reaches the legitimate state. Yet, chances are $\frac{1}{3}$ again that the simulator will remain in the current state since a can also execute a step. Note, that these are the medial chances. Actual chances depend on the aging strategy supported by the scheduler as discussed in section 4.2.8 in the enclosed manual.

After one of the children executed a step, chances are $\frac{1}{3}$ for the remaining node in an illegitimate state to execute a step, disregarding if the current state that followed $\{1, 0, 0\}$ is $\{1, 1, 0\}$ or $\{1, 0, 1\}$.

The MTTR for this model (starting in $\{0, 0, 0\}$, disregarding the scheduler behavior and neglecting fault-probabilities) is calculated as follows:

medial number of steps required to reach $\{1, 0, 0\}$: 3
medial number of steps required to reach $\{1, 1, 0\}$ or $\{1, 0, 1\}$: 2
medial number of steps required to reach $\{1, 1, 1\}$: 3
medial number of steps required: 8

Things details out are the fault-probabilities and initial states other than $\{0, 0, 0\}$.

There are two kinds of faults that can occur in a graph:

- node faults and
- edge faults.

For this example, each fault-possibility is set to 1% = 0.01 which means, that one of 100 executions or transmissions throws a fault that is not catched since only non-masking fault-tolerance is covered.

To start with the root node, the fault-possibility is 0.01 since the root node does not depend on its neighbors. Obviously this makes the edges $e_{b,a}$ and $e_{c,a}$ obsolete for the measurement of fault-tolerance properties.

The fault possibilities for the child nodes b and c is calculated equally:

- 0.01 that parent node is in illegitimate state and since delivers a wrong status which leads to an illegitimate state.
- 0.01 that the result delivered by the parent leads to an illegitimate state.
- 0.01 that the node itself is malfunctioning and sets value to the illegitimate state.

The interesting part is, that these values must not be simply added since faults can also occur in the presence of faults as shown in figure 3.1. Analogously to the calculation of the resulting resistance in combinatorial circuits, the resulting fault-probability for nodes b and c is calculated as following:

$$1 - ((1 - P_{parent\_fault}) \cdot (1 - P_{com\_fault}) \cdot (1 - P_{own\_fault}))$$
$$= 1 - ((1 - 0.01) \cdot (1 - 0.01) \cdot (1 - 0.01))$$
$$= 1 - (0.99^3) = 1 - 0.970299 = 0,029701$$

The appropriate weighted Markov-chain is not presented since the bidirectional edges have two values and the whole structure is too complex. Nevertheless, the resulting availability can be calculated:

- a is 99% of the time in its legitimate state,
- b is 97.0299% of the time in its legitimate state and
- c is 97.0299% of the time in its legitimate state.

The resulting availability is $0.99 \cdot 0.970299 \cdot 0.970299 = 0.93206534790699$. This means, that approximately $93,2\%$ of the steps executed the system is in the green labeled legitimate state and the availability of the system is $93,2\%$.

As another example section 5.3 offers an approach traversed to its full extent.

4. Erlang and SiSSDA

The *simulator for self-stabilizing distributed algorithms* (SiSSDA) is available on the CD-ROM enclosed with this thesis and a detailed discussion is presented in the enclosed manual. This chapter deals with theoretical issues only.

4.1 Choice of Tools

Several technologies are required for the implementation of a simulator. Due to limited capabilities of the language that is most commonly used at the University of Oldenburg, Java, an alternative language was chosen that surpasses especially by means of distributed computing. Further advantages of the language of choice are

- rapid prototyping (the final version is the third branch of development),
- effective deployment of distributed techniques,
- high degree of reliability [Arm, P.29],
- scalability of scenarios due to the ability to cope with dynamic environments,
- performance [1],
- hot code-update,
- dynamic size of all objects,
- sufficient importance for industrial sector and
- availability to the public since Erlang is released under the Erlang Public License (EPL).

Furthermore, Erlang is a *lazy purely-functional programming language to develop distributed fault-tolerant soft real-time non-stop applications.*

[1] http://www.sics.se/~joe/ericsson/du98024.html

4.1.1 Erlang

Erlang was chosen for implementation as programming language and runtime environment. Erlang was chosen for the implementation of the simulator due to built-in support for concurrency control, distribution and fault tolerance.

One of the inventors of Erlang is Joe Armstrong, who is currently employed by the *Swedish Institute of Computer Science* and still contributing to Erlang. The Ericsson Computer Science Laboratory (cs-lab) where Erlang was initially developed was shut down in 2002. Since then Erlang was continued first by Bluetail and afterwards by Alteon who bought Bluetail.

Erlang was originally a modification of the logical language *Prolog* and its evolution was mostly influenced by the requirements of Ericsson, who required a language that is optimal for communication-related tasks. One advantage of Erlang is the practice of light-weight processes with minimal overhead which grants massively scalable systems such as the web-server *YaWS* (Yet another web-server) [2].

The language was named after the danish mathematician and engineer Agner Krarup Erlang and is not an abbreviation of *Er*icsson *Lang*uage.

Figure 4.1: `Agner Krarup Erlang` (*Jan/1/1878 †Feb/3/1929)

The simulator also uses the *OTP* (Open Telecom Platform) , a potent library that is enclosed in Erlang. The OTP features many required functions such as list handling and a pseudo-random number generator. It is similar in scope to *.NET* but limited to Erlang.

If further environments should demand verification of the source code, there is also a free verification tool available, called *Erlang Verification Tool* (EVT) [3] [Mar06] that facilitates the approval of correctness due to requirements formulated in a specification language. Furthermore, the tool *Dialyzer* (*Di*screpancy *Ana*l*yz*er for *Er*lang Programs) facilitates development of complex programs and is a reasonable extension to use[4].

Erlang's main advantages are features that are not common for programming languages but rather for operating systems:

- concurrent processes,

[2] http://www.sics.se/~joe/apachevsyaws.html
[3] http://www.sics.se/fdt/vericode/evt.html
[4] http://www.it.uu.se/research/group/hipe/dialyzer/

- scheduling,
- memory management,
- distribution and
- networking.

A short introduction is presented in section 4.1 of the enclosed manual. Yet it is advised to consult standard literature that is also mandatory for a qualified understanding of Erlang.

4.1.2 Design and Conception

The simulator was implemented to cope with physically distributed topologies but is also able to simulate a distributed environment on a single computer. Other languages would rather intend a sequential execution limiting the use of distribution to the simulation since the execution semantics used for this thesis are serialized. In the face of real world behavior (*the world is distributed and parallel* [Arm, P.9]), applicability to similar scenarios and future purposes the simulator was implemented using processes. The reliability of the results basically relies on

1. the random-number generator and
2. reliability of the programming language used.

As covered earlier in [Arm, P.29], Erlang is sufficiently reliable. Besides, Erlang is fault-tolerant and can also detect and report errors. For the random-number generator the common method to test the reliability is to implement a χ^2 test.

χ^2 Test

The χ^2 test is a *test for the goodness of fit* [5], (it tests, how well a statistical model fits a set of observations) that is calculated as shown in the following equation:

$$\chi^2 = \sum_{i=1}^{n} \frac{(h_i - h_E)^2}{h_E} \tag{4.1}$$

h_i is called the *Null Hypothesis* for category i. All possible events are categorized. Each category has an expected value h_E and a measured value h_i.

To test the random-number generator a set of ten categories was chosen. The source code is available on the enclosed CD-ROM in folder `/svn/erlang/chisqr`. The test has to be compiled first with `c(chisqr).` from within the Erlang shell or with `erlc chisqr.erl` from the console. The method can be executed with `chisqr(`*INTEGER*`).` where *INTEGER* is the number of runs.

The results indicate that the random-number generator is highly reliable to generate numbers that are sufficiently random.

Another more practical related observation of random-number generation is the number of steps each process was granted execution. After each simulation, processes report the number of steps they were chosen. These values only show minimal deviations.

[5] http://biomet.oxfordjournals.org/cgi/reprint/66/3/585.pdf

4.2 Settings

To gather results, topologies and algorithms presented in section 2.4 were used in connection with certain settings adjusted in the configuration file described in section 4.2 in the enclosed manual.

The succeeding chapter compares

- the topologies based on the behavior of the algorithms and
- the algorithms based on the behavior of the topologies.

For all tests the same configuration was used that is given in chapter 4.2 of the manual, except for the tests concerning the mutual exclusion algorithm. As the results for the employed parameters indicate, the setting for accuracy was not strict enough as the resulting Figure 5.11 indicates. The graphs have to be strictly monotonic decreasing.

5. Simulation Results

After describing the theoretical way of deriving fault-tolerance measures and motivating the solution to use a simulator since complexity of certain scenarios makes calculation infeasible as discussed in [War06], case studies presented in section 2.4 are compared by means of topologies and algorithms used. The results have been acquired with the simulator whose implementation is a core artifact of this thesis.

For further information on the simulator the enclosed manual is helpful. Not only the practical method of operation is discussed regarding several theoretical issues. Also instructions are presented on how to append new algorithms and topologies and even dynamic fault-environments are introduced.

Notably, measurement of reliability was skipped. One major difference in measuring availability and reliability is the initial state. While availability is measured with the use of arbitrary initial states, reliability requires the system to start in a legitimate state to measure the attributes MTTF, MTTR and MTBF. Since an average execution consists of approximately 500000 steps and the legitimate state for the scenarios tested is reached within the first 1000 steps (according to section 4.2.1 in the manual regarding the minimal number of steps), the influence is lower than $\frac{1}{500}$ so differences to the graphs representing the degree of availability would almost not be visible.

5.1 Practical Results

A scenario consists of

- one algorithm,
- one topology and
- a set of parameters, centralized in the global configuration file.

Since the influence of the configuration can be almost neglected (with regards to section 5.3), the scenarios are combined by means of the topology used in chapter 5.1.1 as well as by means of the algorithm used as presented in chapter 5.1.2.

5.1.1 Comparing Topologies

In this section, topologies are compared. On each topology the three algorithms *BFS*, *DFS* and *LE* were executed since these algorithms can cope with trees in general. In contrast, *MutEx* requires a special case of a tree that is in the shape of a unidirectional ring as featured by the topology *RING8*. On this topology, all four algorithms are feasible and compared.

The results are measured until the degree of availability is $< 1\%$. Reaching this limit in a scenario is a sufficient *stop criterion* since the number of steps required to measure availability decreases rapidly and reliability of results cannot be guaranteed for fault probabilities too high.

On the one hand, results are not precisely visible if a certain low degree is reached. On the other hand, the simulation requires a more subtle granularity set for accuracy when high fault-probabilities are simulated. A third argument for limiting off further simulation when a certain degree of availability is reached, is the direct comparison of the value for fault-probability that accounts to reaching this limit.

The values labeled as global fault possibility (GFP) refer to all fault possibilities set to one value. For example, if GFP is set to 0.1, each node has a probability of 10% to converge to an illegitimate state and each edge has a chance of 10% to cause an erroneous submission which also leads to an illegitimate state if no further redundancy is provided.

The first topology presented to compare algorithms is *SERIAL8*.

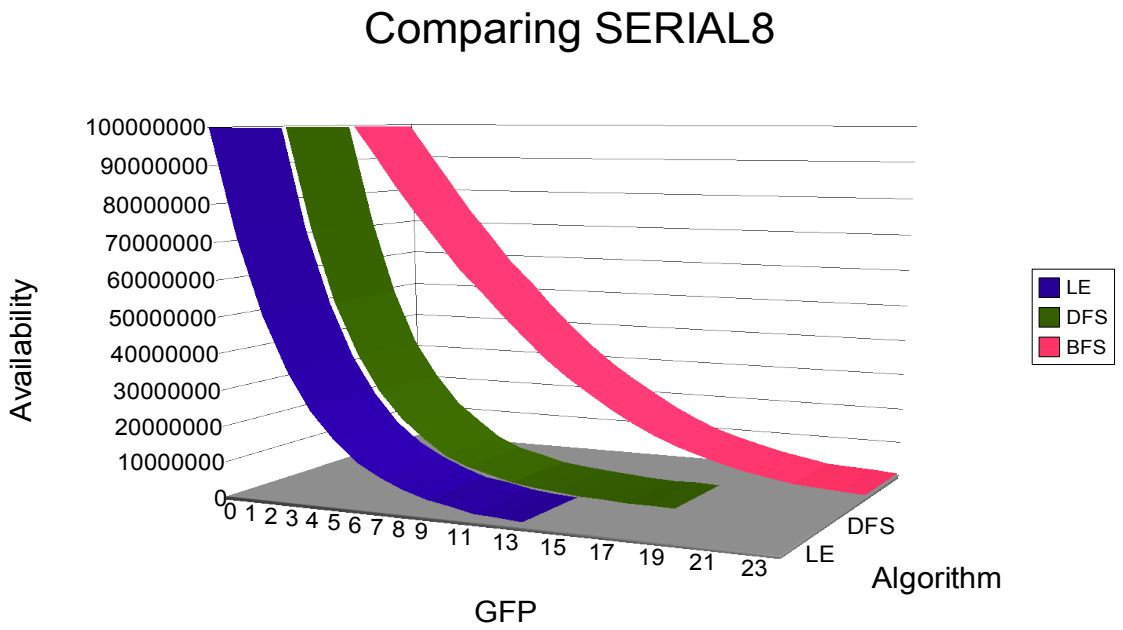


Figure 5.1: Comparing algorithms BFS, DFS and LE with topology SERIAL8.

While availability drops below 1% for LE at a GFP of 13%, DFS reaches the stop criterion at a GFP of 17% and BFS, thanks to the redundancy provided by bidirectional edges, even executes with a GFP of 22% until availability is too low.

Obviously, BFS is more endurable to faults in this kind of topology while LE suffers soonest from fault-propagation.

While fault-propagation was an important factor for the previous topology, the following topology disregards this influence since all nodes only rely on the root node and also redundancy is abolished.

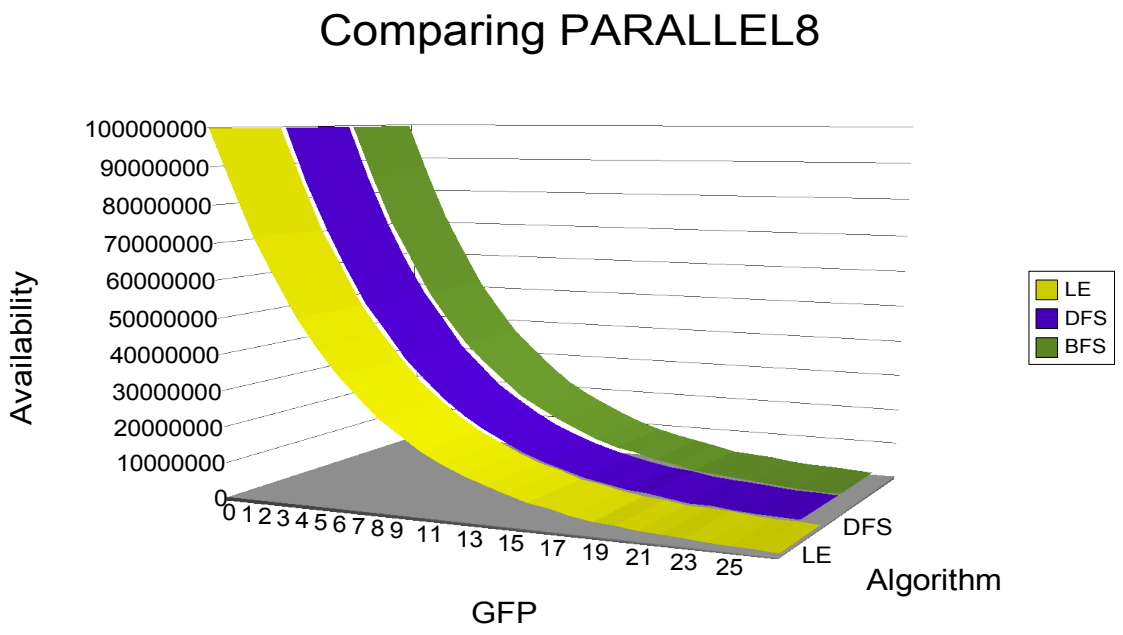


Figure 5.2: Comparing algorithms BFS, DFS and LE with topology PARALLEL8.

Disposing these two effects obviously levels the fault-tolerance to a common base. The winning margin of LE is minimal, yet DFS gains endurance not only in contrast to other algorithms, but also compared to its own previous result. A direct comparison presented in the next section will provide further more exact information.

The topology *COMPLEX8* features a high degree of complexity as presented in table 3.1.1, where fault-propagation and redundancy reach a higher level than in the previous topologies.

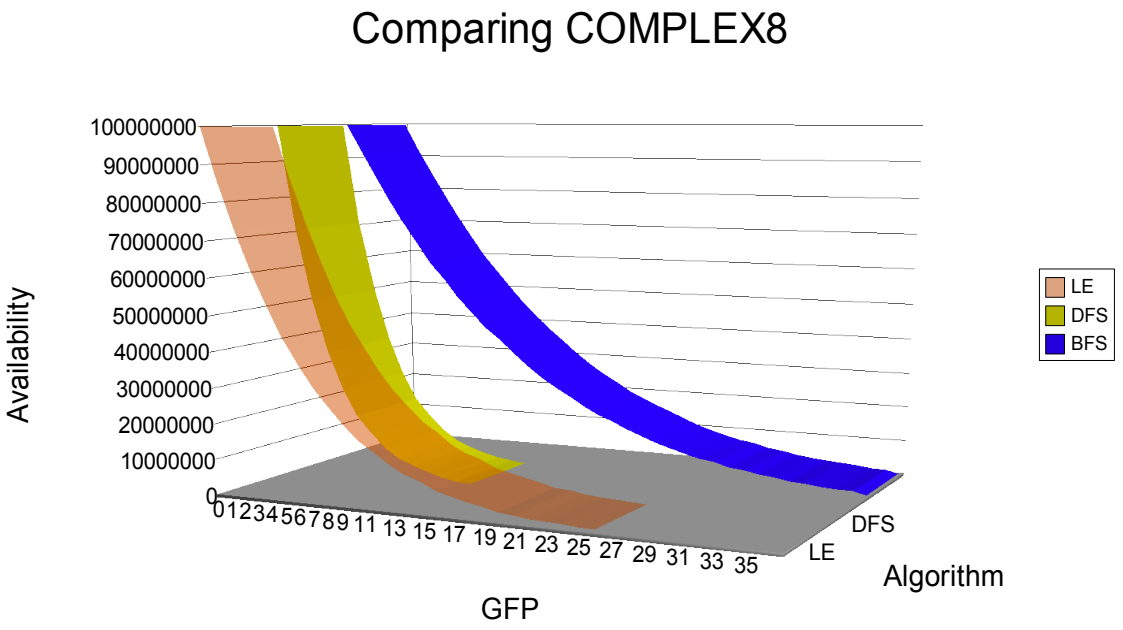


Figure 5.3: Comparing algorithms BFS, DFS and LE with topology COMPLEX8.

Again BFS benefits from redundancy which the other algorithms are deprived of. Yet fault-propagation seems to have a great influence on DFS, while LE is almost perfectly in the middle between the other algorithms. This topology leads to the greatest differences so far. In contrast to the preceeding topology where the algorithms showed a similar degree of availability, differences in this environment are even greater than measured in the first topology.

Notably, the sequence of reaching the stop criterion is not the same comparing this topology to SERIAL8. While in topology SERIAL8 first LE, then DFS and finally BFS reached the limit, the sequence for topology COMPLEX8 is DFS, LE and then BFS.

Using topology RING8 for a scenario enables also the MutEx algorithm to participate in the comparison.

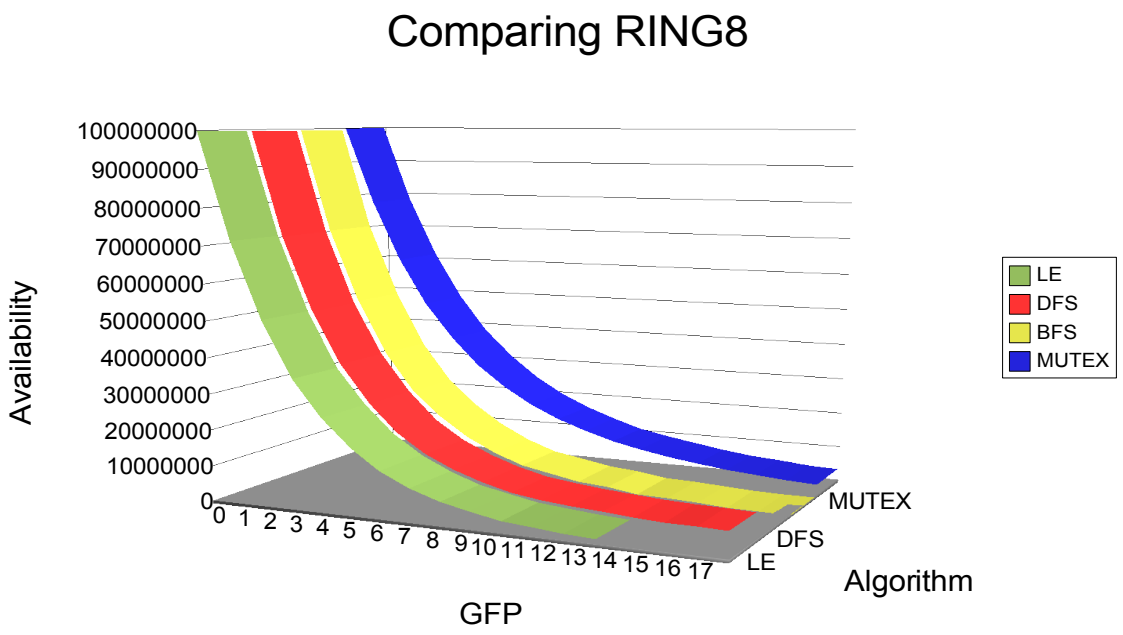


Figure 5.4:
Comparing algorithms BFS, DFS, LE and MutEx with topology RING8.

Compared to all other algorithms which show a similar rate of availability, MutEx shows to be more endurable. Nevertheless, accuracy settings were different for MutEx since a stricter setting was mandatory as discussed in section 5.3.

5.1.2 Comparing Algorithms

After differences of the topologies on the resulting availability have been tracked, the same results are put in a different order to compare the influence of the algorithms.

The first algorithm that is executed on several topologies is BFS.

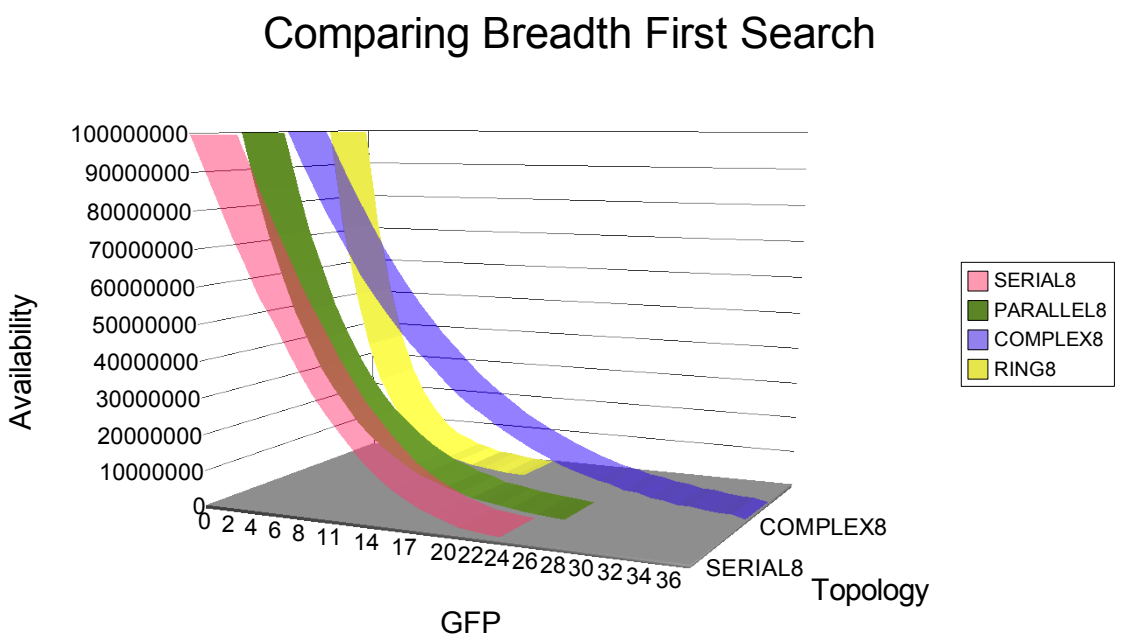


Figure 5.5:
Comparing topologies SERIAL8, PARALLEL8 and COMPLEX8 with algorithm BFS.

While PARALLEL8 and SERIAL8 are in the midfield with only a little difference, RING8 obviously suffers from fault-propagation and a lack of redundancy. In contrast, COMPLEX8 grants a high degree of availability and reaches the stop criterion not until a GFP of 35%. The high degree of interconnectability leads to such a tremendous result since nodes have multiple possibilities to converge to a legitimate state in contrast to RING8 topology.

The next algorithm that is compared is DFS.

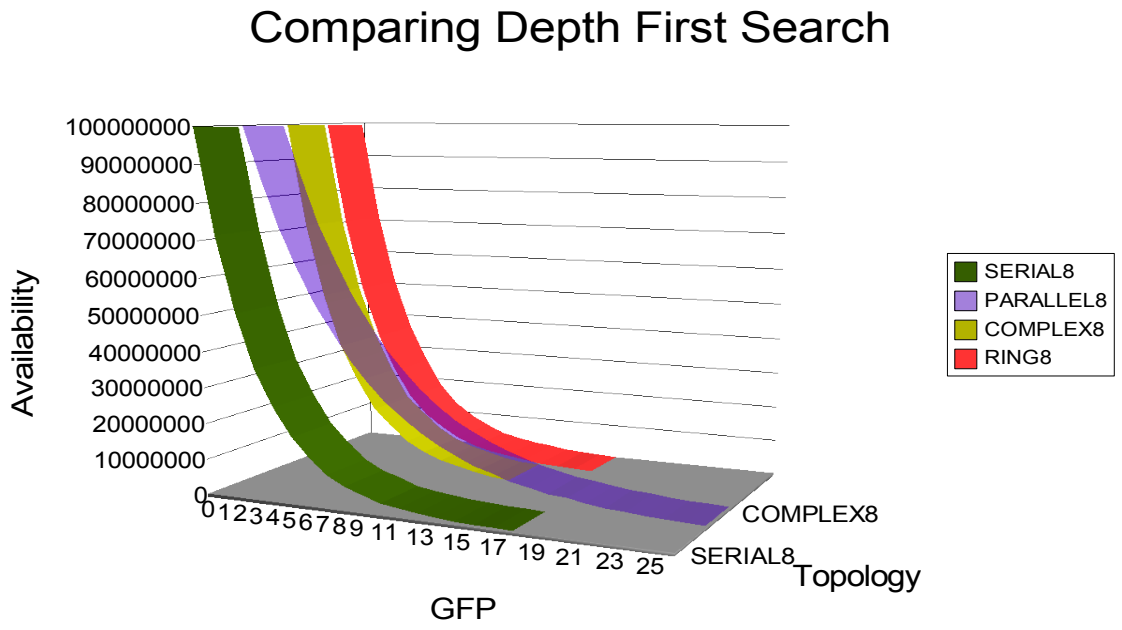


Figure 5.6:
Comparing topologies SERIAL8, PARALLEL8 and COMPLEX8 with algorithm DFS.

The difference to BFS is obvious. DFS performs best in PARALLEL8 and worst in COMPLEX8. The degree of availability of almost inversely proportional to the complexity provided by the topology, yet RING8 performs better than COMPLEX8.

The following figure shows the fault-tolerance of the three topologies with LE.

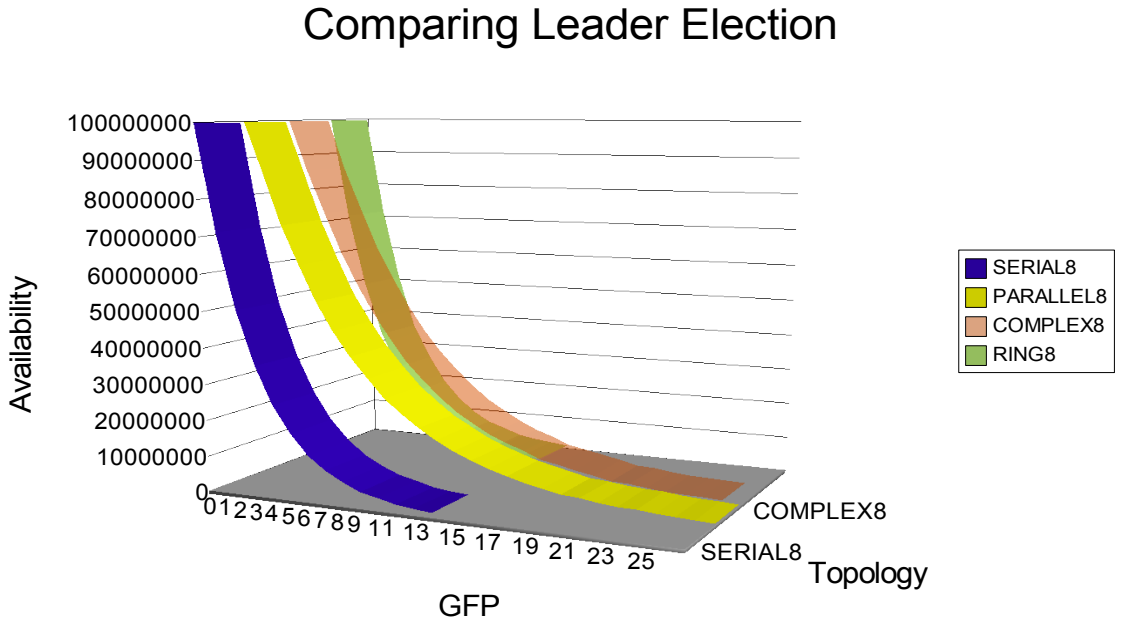


Figure 5.7:
Comparing topologies SERIAL8, PARALLEL8 and COMPLEX8 with algorithm LE.

Obviously PARALLEL8 and COMPLEX8 deliver a similar high degree of availability while RING8 and SERIAL8 have a similar low level. Again, the effects of fault-propagation versus redundancy become clearly visible in this setting.

5.2 Theoretical Results

To exemplify theoretical results, simpler scenarios had to be chosen. For this reason, the model used in [War06, Ch.4] is adopted in combination with DFS.

As discussed in the paper, the approach consists of three steps:

1. calculation of the transition probability matrix for transitions between configuration classes due to a single computation step of a process,
2. determination of the transition probability matrix for transitions between configuration classes due to fault steps and
3. combination of both matrices to obtain a transition probability matrix representing an appropriate Markov chain of the entire system [War06, P.255].

The model instanced represents a system consisting of three processors P_1, P_2 and P_3 connected as presented in the following figure.

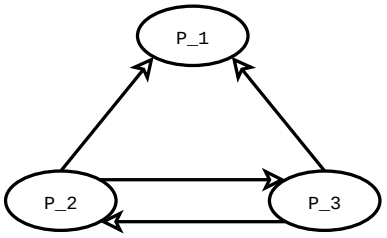


Figure 5.8: Theoretical Example Markov Chain Preparation

The appropriate adjacency matrix is

$$M_{AM} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

After traversing through the three steps, the resulting transition probability matrix is given by

$$M_{TPM} = \begin{bmatrix} 0.6344 & 0.0160 & 0.0160 & 0.0001 & 0.3331 & 0.0001 & 0.0001 & 0.0001 \\ 0.2221 & 0.4124 & 0.0001 & 0.0318 & 0.0001 & 0.3331 & 0.0001 & 0.0001 \\ 0.2221 & 0.0001 & 0.4124 & 0.0318 & 0.0001 & 0.0001 & 0.3331 & 0.0001 \\ 0.0001 & 0.1111 & 0.1111 & 0.4441 & 0.0001 & 0.0001 & 0.0001 & 0.3331 \\ 0.0001 & 0.0001 & 0.0001 & 0.0001 & 0.9357 & 0.0318 & 0.0318 & 0.0001 \\ 0.0001 & 0.0001 & 0.0001 & 0.0001 & 0.1111 & 0.8406 & 0.0001 & 0.0477 \\ 0.0001 & 0.0001 & 0.0001 & 0.0001 & 0.1111 & 0.0001 & 0.8406 & 0.0477 \\ 0.0001 & 0.0001 & 0.0001 & 0.0001 & 0.0001 & 0.0001 & 0.0001 & 0.9991 \end{bmatrix}$$

This matrix can also be represented as Markov-chain:

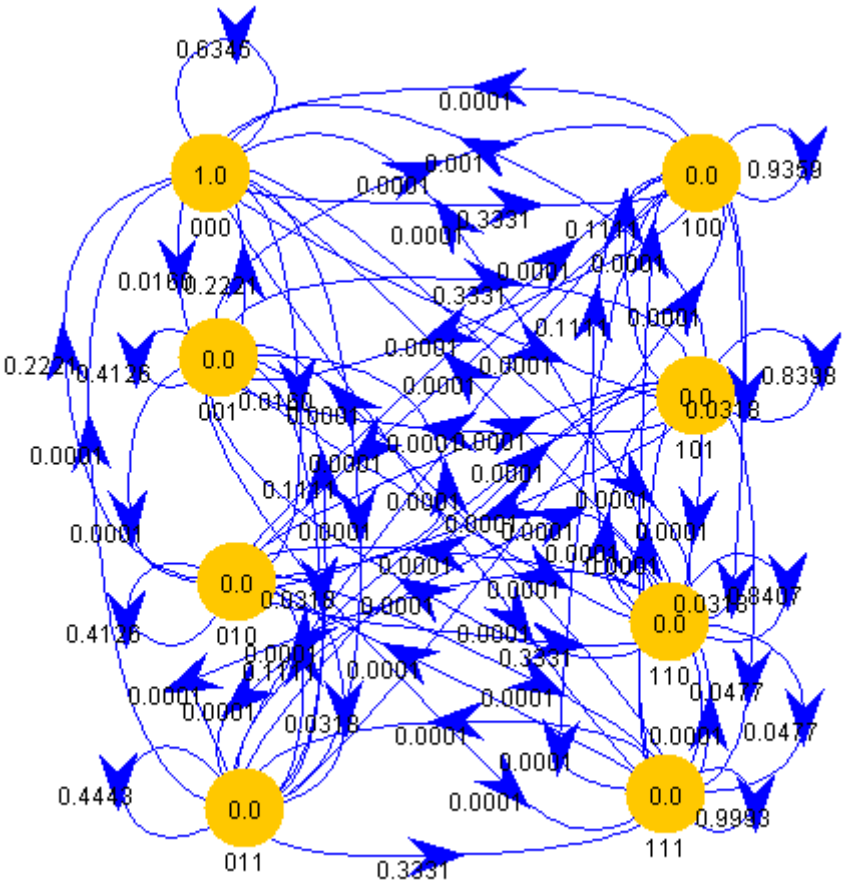


Figure 5.9: Theoretical Example Markov Chain

Using the German tool *jAndrei* by Fabian Grüning [1], the matrix can be tested for for validity and ergodicity. The model is available on the enclosed CD-Rom under `/Tools/papermain.mc`.

Further simulation shows the resulting distribution for the model:

[1] http://www.eco-software.org/software

Startverteilung Übergangsmatrix

		000	001	010	011	100	101	110	111
1.0	000	0.6345	0.016	0.016	1.0E-4	0.3331	1.0E-4	1.0E-4	1.0E-4
0.0	001	0.2221	0.4126	1.0E-4	0.0318	1.0E-4	0.3331	1.0E-4	1.0E-4
0.0	010	0.2221	1.0E-4	0.4126	0.0318	1.0E-4	1.0E-4	0.3331	1.0E-4
0.0	011	1.0E-4	0.1111	0.1111	0.4443	1.0E-4	1.0E-4	1.0E-4	0.3331
0.0	100	1.0E-4	1.0E-4	1.0E-4	1.0E-4	0.9359	0.0318	0.0318	1.0E-4
0.0	101	0.0010	1.0E-4	1.0E-4	1.0E-4	0.1111	0.8398	1.0E-4	0.0477
0.0	110	1.0E-4	1.0E-4	1.0E-4	1.0E-4	0.1111	1.0E-4	0.8407	0.0477
0.0	111	1.0E-4	1.0E-4	1.0E-4	1.0E-4	1.0E-4	1.0E-4	1.0E-4	0.9993

Die Matrix ist valide.
Die Matrix ist ergodisch:

000	5.61E-4
001	2.24E-4
010	2.24E-4
011	2.05E-4
100	0.026246
101	0.00628
110	0.006315
111	0.959944

Figure 5.10: Theoretical Example Markov Chain Result

5.3 The MutEx-Scenario

After executing the scenarios, values obtained for MutEx indicated a different behavior than the other algorithms. While all graphs shown above featured a strictly monotonic development, the graph representing the MutEx algorithm showed a wavy not monotonic development for higher GFPs. Referring to the number of steps that had to be executed to acquire the availability for a certain GFP, there was no notable difference to the other algorithms. So the number of steps that have to be executed to reach a certain degree of stability is similar for all algorithms.

Yet, comparing the development of single execution scenarios featuring one GFP as presented in figure 4.1 in the enclosed manual, showed a different kind of conversion.

While development of BFS, DFS and LE is like a swinging around the desired value comparable to a pendulum, MutEx converges to a certain value approaching strictly from the value 1.0 like a cushioned fall (as described in [Ng]).

This leads to a lower degree of *swinging-in convergence* for the values obtained for MutEx and hence to a premature achievement of the accuracy guards. Therefore, accuracy parameters have to be clamped down. Comparing the two graphs presented in the following figure, accuracy parameters for the monotonic graph are obviously sufficiently precise.

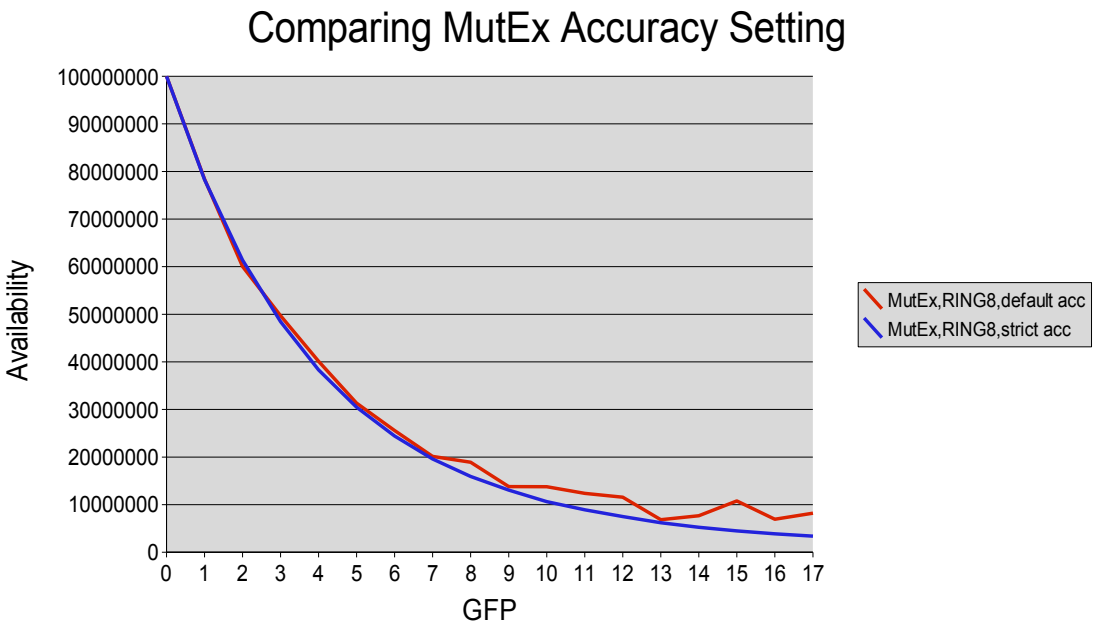


Figure 5.11: Mutual Exclusion measured with two sets of accuracy settings.

5.4 Conclusion

The theoretical approach presented in section 5.3 shows that complexity grows exponential proportional to the number of nodes and edges. Calculating models consisting of three nodes suffices to deliver a high level complexity that is barely processable. To cope with large models, it is reasonable to use a simulator as presented in section 5.1.1.

Another advantage of the simulator is, that approximations as used in the theoretical approach, that hinder the accuracy of the results obtained, are obsolete. Yet, accuracy is an important matter for the simulation, too, as discussed in Ch. 4.2.1 in the manual.

6. Conclusion

6.1 Summary

In this thesis, we implemented a simulator to determine fault-tolerance measures for different self-stabilizing distributed algorithms in the presence of transient faults in non-masking fault-tolerant systems. In particular, the aspect of self-stabilization as well as the derivation of appropriate Markov-chains were discussed in reference to the cited papers and sources to motivate the necessity of the simulator.

Further, several representative topologies were tested thoroughly in combination with the algorithms to compare properties of these scenarios. It was shown, that fault-propagation and redundancy have a converse effect on the resulting availability and all parameters of a scenario are important for the outcome.

Also, discussion of execution semantics has been shown to be an important component for the practical appliance. Furthermore, the simulator features dynamic environments that were not regarded by the theoretical approach. Using Erlang as programming language for implementation has proven useful. After implementing two prototypes, a third implementation-branch was started that surpasses its predecessors in performance, design and code-efficiency.

Analyzing the results of the different settings for the case-studies was insightful and showed the profitableness of the simulator.

6.2 Outlook

The implemented simulator offers a wide range of possibilities for further investigation. It is not only expandable by means of topologies and algorithms, but, for instance, simulation was only executed with serialized execution semantics. Furthermore, environmental influences as featured by the fault-environment class were used neither.

But not only the scope was not exploited to its full extent. Classification of simulated scenarios by means of characteristic behavior was not part of this thesis, too.

Further areas of investigation might be:

- a topology-validator that automatically checks a given topology for the feasibility for all algorithms
- further distributed algorithms
  - byzantine failure algorithm
  - ant colony algorithm
  - hill climbing algorithm
  - Viterbi algorithm
- dynamic fault-environments featuring burn-in and burn-out phases or seasonal influence for example
- employment of algorithms using waiting strategies that are already available
- the effect of (dynamic) execution semantics > 1

Bibliography

[AB98] Yehuda Afek and Anat Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 1998(3), December 1998.

[ADI03] Chirayu S. Amin, Florentin Dartu, and Yehea I. Ismail. Weibull based analytical waveform model. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 161, Washington, DC, USA, 2003. IEEE Computer Society.

[AFG93] Paul C. Attie, Nissim Francez, and Orna Grumberg. Fairness and hyperfairness in multi-party interactions. *Distrib. Comput.*, 6(4):245–254, 1993.

[Ala03] K. Alagarsamy. Some myths about famous mutual exclusion algorithms. *SIGACT News*, 34(3):94–103, 2003.

[Arm] Joe Armstrong. *Concurrency Oriented Programming in Erlang*. SICS. http://ll2.ai.mit.edu/talks/armstrong.pdf, last visited: 2006-09-25.

[Arm03] Dr. Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[Att00] Hagit Attiya. Efficient and robust sharing of memory in message-passing systems. *Journal of Algorithms*, 34(1):109–129, 2000.

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

[BT] Paul E. Black and Paul J. Tanenbaum. *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology.

[CD94] Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.

[CDK99] Zeev Collin, Rina Dechter, and Shmuel Katz. Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*, 1999.

[Dij02] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. pages 198–227, 2002.

[DIM97] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 08(4):424–440, 1997.

[DLN06] Michael E. Orrison David L. Neel. The linear complexity of a graph. *The Electronic Journal of Combinatorics*, 2006.

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

[DW04] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM*, 51(5):780–799, 2004.

[Frä] Martin Fränzle. *Folienskript Eingebettetet Systeme*. Universität Oldenburg.

[Gar04] Vijay K. Garg. *Concurrent and Distributed Computing in Java*. John Wiley & Sons, 2004.

[Gei] B. Geib. *Fehlertolerante Systeme*. Fachhochschule Wiesbaden. http://www-1.informatik.fh-wiesbaden.de/~rnlab/LVweb/FTS_Kap_1.pdf, last visited: 2006-09-25.

[GGHS96] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 45–54, New York, NY, USA, 1996. ACM Press.

[Gär99] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.

[GS97] Charles M. Grinstead and Laurie J. Snell. *Introduction to Probability*. American Mathematical Society, July 1997.

[GT90] Robert Geist and Kishor S. Trivedi. Reliability estimation of fault-tolerant systems: Tools and techniques. *Computer*, 23(7):52–61, 1990.

[Jhu] Arshad Jhumka. *Automated Design of Efficient Fail-Safe Fault Tolerance*. PhD thesis, Technische Universität Darmstadt.

[KJG] Ph.D. Kenneth J. Goldman. *Leader Election*. Washington University. http://www.cs.wustl.edu/~kjg/CS333_SP97/leader.html, last visited: 2006-09-25.

[Kok05] Can Emre Koksal. An analysis of blocking switches using error control codes. *Submitted to IEEE Transactions on Information Theory*, October 2005.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

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

[Mi06] Jie Mi. Limiting availability of system with non-identical lifetime distributions and non-identical repair time distributions. *Statistics & Probability Letters*, 76(7):729–736, 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.

[Muh] Rashid Bin Muhammad. *Design and Analysis of Algorithms*. Kent State University. http://www.personal.kent.edu/~rmuhamma/Algorithms/algorithm.html, last visited: 2006-09-25.

[Ng] Chung-Sang Ng. *Introduction to Physics - Impulse and Momentum*. The University of New Hampshire. http://plasma4.sr.unh.edu/ng/phys401/phys401-6-9-04.pdf, last visited: 2006-09-25.

[oEE90] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. IEEE, New York, NY, USA, 1990.

[PGV] Claude-Pierre Jeannerod Pascal Giorgi and Gilles Villard. *On the Complexity of Polynomial Matrix Computations*. École Normale Supérieure De Lyon. http://perso.ens-lyon.fr/claude-pierre.jeannerod/papers/issac03.pdf, last visited: 2006-09-25.

[Rela] ReliaSoft. *Introduction to Repairable Systems*. ReliaSoft Corporation. http://www.weibull.com/SystemRelWeb/introduction_to_repairable_systems.htm, last visited: 2006-09-25.

[Relb] ReliaSoft. *The Reliability Function*. ReliaSoft Corporation. http://weibull.com/hotwire/issue7/relbasics7.htm, last visited: 2006-09-25.

[Sch93] Marco Schneider. Self-stabilization. *ACM Comput. Surv.*, 25(1):45–67, 1993.

[Ski98] Steven S. Skiena. *The algorithm design manual*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[SS92] S. Sur and P. K. Srimani. A self-stabilizing distributed algorithm to construct BFS spanning trees of a symmetric graph. *Parallel Processing Letters*, 2(2-3):171–179, 1992.

[Tea] Erlang Product Team. *Erlang White Paper*. Ericsson. http://erlang.org/white_paper.html, last visited: 2006-09-25.

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

[Tz] Gustavo Alonso Tamer Özsu. *Parallel and Distributed Databases*. Swisss Federal Institute of Technology Zurich. http://www.iks.inf.ethz.ch/education/ss06/PDDBS/Streams.pdf, last visited: 2006-09-25.

[VS] Ute Vogel and Michael Sonnenschein. *Folienskript Modellbildung und Simulation ökologischer Systeme*. Universität Oldenburg.

[War06] Abhishek Dhama; Oliver Theel; Timo Warns. Reliability and availability analysis of self-stabilizing systems. In *8th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, page 17. Springer, 2006.

[WCWH03] Fu-Hsing Wang, Jou-Ming Chang, Yue-Li Wang, and Sun-Jen Huang. Distributed algorithms for finding the unique minimum distance dominating set in directed split-stars. *J. Parallel Distrib. Comput.*, 63(4):481–487, 2003.

[WR03] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. 2003.

List of Figures

Glossary

BFS	Breadth-First Search
cs-lab	Ericsson Computer Science Laboratory
CTMC	Continuous Time Markov-chain
DFS	Depth-First Search
Dialyzer	Discrepancy Analyzer for Erlang Programs
DTMC	Discrete Time Markov-chain
EVR	Erlang Verification Tool
FIFO	first-in-first-out
GFP	Global Fault Possibility
MTBF	Mean-Time Between Failures
MTTF	Mean-Time To Failure
MTTR	Mean-Time To Repair
MutEx	Mutual Exclusion
OTP	Open Telecom Platform
PMF	Probability Mass Function
SICS	Swedish Institute of Computer Science
SiSSDA	Simulator for Self-Stabilizing Distributed Algorithms
TPF	Transition Probability Function
YaWS	Yet another web-server

Availability	BFS,SERIAL^	BFS,PARALLE	BFS,COMPLE	BFS,RING8	DFS,SERIAL^	DFS,PARALLE	DFS,COMPLE	DFS,RING8
0	99575800	99755100	99750100	99612800	99695900	99819800	99676000	99650100
1	90919300	84977900	92001800	71335500	71774700	84920500	71229300	71127200
2	82137900	72177500	84953300	51408200	51160700	71890300	51118300	50866100
3	73878300	61546400	77860900	36435600	36354200	61021400	36491100	36467100
4	66166800	51648000	71722400	25877100	26507400	51252000	26277000	26059800
5	58305200	43750500	65608300	18894100	18660600	43724300	18355000	18214200
6	52014700	36880800	59721800	13531900	13546600	36413800	13636200	12981600
7	45421800	31450900	54737200	9578160	9236400	30756900	9487430	9391820
8	39120300	26444500	49835100	6642030	6587610	25522500	6745830	6658010
9	33434600	22008400	45384800	5093070	4918010	21506000	4893820	4624860
10	28261600	18634800	41158200	3588470	3248790	17800200	3524850	3464030
11	23874300	15693600	37627100	2855640	2389500	14939700	2382680	2577730
12	19808800	13060200	33873100	1645400	1722160	12338500	1642360	2029100
13	16022100	10727400	30492600	1399040	1239890	10202100	1195750	1241920
14	12749300	8865470	27313200	964305	812333	8510230		848456
15	10123200	7563840	24556500	415262	558959	6875800		792098
16	7990080	6186200	22085700	212064	499251	5817360		441526
17	6173460	5258500	19497400	141368	316156	4686100		
18	4564200	4029830	17329900			4221620		
19	3366310	3416810	15465400			3324460		
20	2161960	2912830	13646700			2619100		
21	1802010	2289580	11956500			2274250		
22	1311430	1890710	10351300			1806860		
23	973342	1464510	8946240			1405290		
24		1202010	7739440			1246940		
25		989899	6662140			996186		
26			5986390					
27			4832630					
28			4326320					
29			3420830					
30			2948660					
31			2279180					
32			2072980					
33			1534530					
34			1179980					
35			1085050					
36			64016					

Availability	LE,SERIAL8	LE,PARALLEL	LE,COMPLEX	LE,RING8	MutEx,RING^,defa	MutEx,RING^,
0	99691800	99765500	99799900	99730800	100000000	100000000
1	71461900	85975700	85930300	71502700	78403100	78273500
2	50881600	71975100	73761500	51039800	60034600	61374800
3	36154100	61264100	62837300	35825800	49710300	48446000
4	25940500	51110800	53286800	26018300	40133000	38303100
5	18554200	43444600	45098600	18458900	31328500	30478700
6	13106400	36519600	38175000	13126000	25568400	24438300
7	9428210	30704800	32039500	9304510	20146900	19604700
8	6861970	25580400	27055200	6487610	18919000	15924200
9	4716400	21641600	22474800	4795000	13786800	13063100
10	3598000	17957600	18674900	3128050	13775500	10646000
11	2391060	15117900	15681400	2404100	12382700	8928270
12	1744940	12584000	12781300	1727660	11563300	7499400
13	1357320	10467500	10620900	1223670	6840810	6212470
14		8705530	9005220		7670350	5243100
15		7028090	6988500		10771800	4504350
16		5893580	5903530		6952990	3884000
17		4763600	4934520		8208930	3392580
18		3727550	3996500		6339100	
19		3345090	2906830		4629180	
20		2660750	2619100		5471230	
21		2222550	1982370		4922580	
22		2006180	1537790		5244690	
23		1571570	1322830		4855340	
24		1208230	1060600		3600290	
25		1003690	859300		4827210	
26		708820			2366160	
27						
28						
29						
30						
31						
32						
33						
34						
35						
36						

Figure 6.1: Simulation Results