Using Erlang for Distributed Simulation for the Derivation of Fault Tolerance Measures

Nils Müllner

August 19, 2008

CARL VON OSSIETZKY universität OLDENBURG

Outline

- Motivation
- Theory
- Erlang
- Simulation
- Conclusion

Motivation

- Why Fault Tolerance?

Motivation

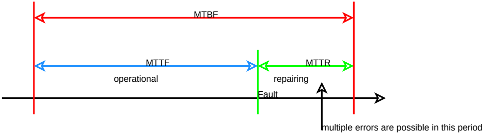
- Why Fault Tolerance?
- Why Simulation?

Motivation

- Why Fault Tolerance?
- Why Simulation?
- Why Erlang?

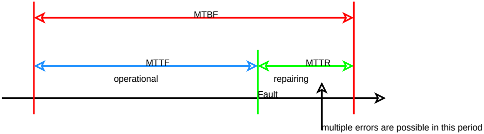
Fault Tolerance Measures

- Reliability, Availability, Safety, Trustworthiness



Fault Tolerance Measures

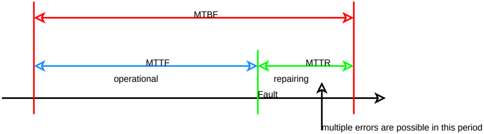
- Reliability, Availability, Safety, Trustworthiness



- Essential for Critical Systems

Fault Tolerance Measures

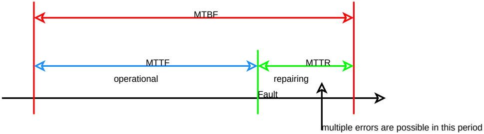
- Reliability, Availability, Safety, Trustworthiness



- Essential for Critical Systems
- Masking, Nonmasking and Failsafe

Fault Tolerance Measures

- Reliability, Availability, Safety, Trustworthiness



- Essential for Critical Systems
- Masking, Nonmasking and Failsafe
  - Masking: Safety and Liveness
  - Nonmasking: Liveness
  - Failsafe: Safety

Simulation

- Easy and fast to implement

Simulation

- Easy and fast to implement
- More accurate than analysis

Simulation

- Easy and fast to implement
- More accurate than analysis
- Extremely scalable

Simulation

- Easy and fast to implement
- More accurate than analysis
- Extremely scalable
- Suitable for a large class of problems

Simulation

- Easy and fast to implement
- More accurate than analysis
- Extremely scalable
- Suitable for a large class of problems

- BUT: Requires (many) resources

Erlang

- Distributed

Erlang

- Distributed
- Concurrent

Erlang

- Distributed
- Concurrent
- Functional

Erlang

- Distributed
- Concurrent
- Functional
- λ-calculus [Barendregt and Barendsen, 2000]

Erlang

- Distributed
- Concurrent
- Functional
- λ-calculus [Barendregt and Barendsen, 2000]
- *pure* (no side-effects, lazy evaluation) and *eager*

Functional Languages

- Lisp, Haskell, Scheme, Erlang

Functional Languages

- Lisp, Haskell, Scheme, Erlang
- Often combined with other paradigms (logical, imperative, object-oriented, constraint, distributed, and concurrent programming)

Functional Languages

- Lisp, Haskell, Scheme, Erlang
- Often combined with other paradigms (logical, imperative, object-oriented, constraint, distributed, and concurrent programming)
- Functions are algorihms

Functional Languages

- Lisp, Haskell, Scheme, Erlang
- Often combined with other paradigms (logical, imperative, object-oriented, constraint, distributed, and concurrent programming)
- Functions are algorihms
- Algorithms can be splitted into subalgorithms

Functional Languages

- Lisp, Haskell, Scheme, Erlang
- Often combined with other paradigms (logical, imperative, object-oriented, constraint, distributed, and concurrent programming)
- Functions are algorihms
- Algorithms can be splitted into subalgorithms
- Parallelization by modularizing programs

Functional Languages

- Lisp, Haskell, Scheme, Erlang
- Often combined with other paradigms (logical, imperative, object-oriented, constraint, distributed, and concurrent programming)
- Functions are algorihms
- Algorithms can be splitted into subalgorithms
- Parallelization by modularizing programs
- Easy to verify

So, what do we want?

- *Simulation* with

So, what do we want?

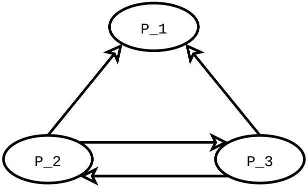
- *Simulation* with
- a *Functional Language* to

So, what do we want?

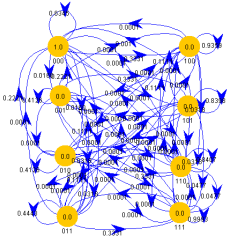
- *Simulation* with
- a *Functional Language* to
- derive *Fault Tolerance Measures*

Getting Results with Analytic Methods: Theory

- Model Distributed System as Markov Chain

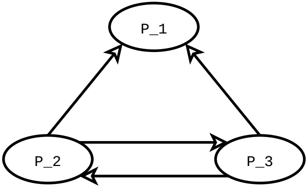


=>

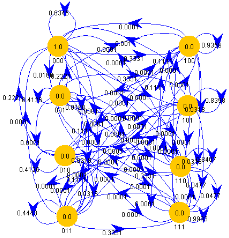


Getting Results with Analytic Methods: Theory

- Model Distributed System as Markov Chain



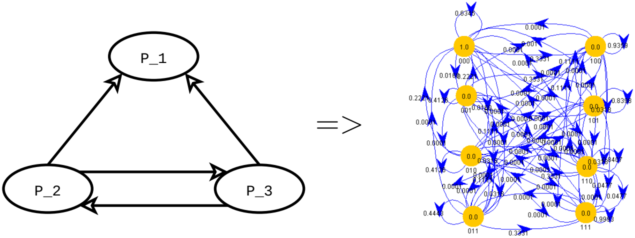
=>



- Suffers from state space explosion

Getting Results with Analytic Methods: Theory

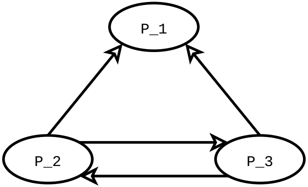
- Model Distributed System as Markov Chain



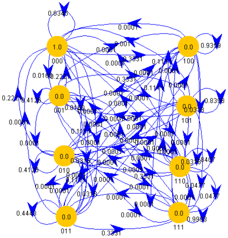
- Suffers from state space explosion
- Solution: Partition state space

Getting Results with Analytic Methods: Theory

- Model Distributed System as Markov Chain



=>



- Suffers from state space explosion
- Solution: Partition state space
- Problem: Abstraction hinders accuracy of results derived tremendously

Theory

- Only conservative estimations

Theory

- Only conservative estimations
- Not even close to reality... (cf. [Dhama et al., 2006])

Theory

- Only conservative estimations
- Not even close to reality... (cf. [Dhama et al., 2006])
- Size of applicable topologies very limited

Theory

- Only conservative estimations
- Not even close to reality... (cf. [Dhama et al., 2006])
- Size of applicable topologies very limited
- Advantage: results are proven...

Erlang 1/5


- Development started in 1986 as Prolog Interpreter at Ericsson CSLab

Erlang 1/5


- Development started in 1986 as Prolog Interpreter at Ericsson CSLab
- A language for programming distributed fault-tolerant soft real-time non-stop applications.

Erlang 1/5

- Development started in 1986 as Prolog Interpreter at Ericsson CSLab
- A language for programming distributed fault-tolerant soft real-time non-stop applications.
- Purely Functional Language

Erlang 1/5

- Development started in 1986 as Prolog Interpreter at Ericsson CSLab
- A language for programming distributed fault-tolerant soft real-time non-stop applications.
- Purely Functional Language
- Interpreted or compiled

Erlang 1/5

- Development started in 1986 as Prolog Interpreter at Ericsson CSLab
- A language for programming distributed fault-tolerant soft real-time non-stop applications.
- Purely Functional Language
- Interpreted or compiled
- Hot Code Plugging

Erlang 2/5

- Focuses on parallelism and fault tolerance

Erlang 2/5

- Focuses on parallelism and fault tolerance
- Highly reliable (Switch AXD301 is 99.9999999% reliable, 31 ms/yr downtime)

Erlang 2/5

- Focuses on parallelism and fault tolerance
- Highly reliable (Switch AXD301 is 99.9999999% reliable, 31 ms/yr downtime)
- employs OpenSSL (χ^2-test)

Erlang 2/5

- Focuses on parallelism and fault tolerance
- Highly reliable (Switch AXD301 is 99.9999999% reliable, 31 ms/yr downtime)
- employs OpenSSL (χ^2-test)
- No variables $=>$ instantiated constants

Erlang 2/5

- Focuses on parallelism and fault tolerance
- Highly reliable (Switch AXD301 is 99.9999999% reliable, 31 ms/yr downtime)
- employs OpenSSL (χ^2-test)
- No variables => instantiated constants
- No loops => recursive function calls

Erlang 2/5

- Focuses on parallelism and fault tolerance
- Highly reliable (Switch AXD301 is 99.9999999% reliable, 31 ms/yr downtime)
- employs OpenSSL (χ^2-test)
- No variables => instantiated constants
- No loops => recursive function calls
- No variable declarations => duck types

Erlang 2/5

- Focuses on parallelism and fault tolerance
- Highly reliable (Switch AXD301 is 99.9999999% reliable, 31 ms/yr downtime)
- employs OpenSSL (χ^2-test)
- No variables => instantiated constants
- No loops => recursive function calls
- No variable declarations => duck types
- Prolog Style Syntax, but not a logic language!

```
-module(math).
-export([fac/1]).

fac(N) when N > 0 -> N * fac(N-1);
fac(0) -> 1.
```

,,

```
-module(pingpong).
-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
        Pong_PID ! finished,
        io:format("ping finished ~n", []);

ping(N, Pong_PID) ->
        Pong_PID ! {ping, self()},
        receive
                pong ->
                        io:format("Ping received pong~n", [])
        end,
        ping(N - 1, Pong_PID).

,,
```

```
pong() ->
        receive
                finished ->
                        io:format("Pong finished~n", []);
                {ping, Ping_PID} ->
                        io:format("Pong received ping~n", []),
                        Ping_PID ! pong,
                        pong()
        end.

start() ->
        Pong_PID = spawn(pingpong, pong, []),
        spawn(pingpong, ping, [3, Pong_PID]).
```

"

Simulation Framework 1/5

- monitoring facility (prints every n^{th} step)

Simulation Framework 1/5

- monitoring facility (prints every n^{th} step)
- runs until desired accuracy is reached (maximal acceptable deviation within last n turns)

Simulation Framework 1/5

- monitoring facility (prints every n^{th} step)
- runs until desired accuracy is reached (maximal acceptable deviation within last n turns)
- four distributed self-stabilizing algorithms provided

Simulation Framework 1/5

- monitoring facility (prints every n^{th} step)
- runs until desired accuracy is reached (maximal acceptable deviation within last n turns)
- four distributed self-stabilizing algorithms provided
  - Breadth First Search

Simulation Framework 1/5

- monitoring facility (prints every n^{th} step)
- runs until desired accuracy is reached (maximal acceptable deviation within last n turns)
- four distributed self-stabilizing algorithms provided
  - Breadth First Search
  - Depth First Search

Simulation Framework 1/5

- monitoring facility (prints every n^{th} step)
- runs until desired accuracy is reached (maximal acceptable deviation within last n turns)
- four distributed self-stabilizing algorithms provided
  - Breadth First Search
  - Depth First Search
  - Leader Election

Simulation Framework 1/5

- monitoring facility (prints every n^{th} step)
- runs until desired accuracy is reached (maximal acceptable deviation within last n turns)
- four distributed self-stabilizing algorithms provided
  - Breadth First Search
  - Depth First Search
  - Leader Election
  - Mutual Exclusion

Simulation Framework 1/5

- monitoring facility (prints every n^{th} step)
- runs until desired accuracy is reached (maximal acceptable deviation within last n turns)
- four distributed self-stabilizing algorithms provided
  - Breadth First Search
  - Depth First Search
  - Leader Election
  - Mutual Exclusion
- easy to extend

Simulation Framework 2/5

- exact fault environments (specify distinct values for each vertex and edge)

Simulation Framework 2/5

- exact fault environments (specify distinct values for each vertex and edge)
- dynamic fault environments

Simulation Framework 2/5

- exact fault environments (specify distinct values for each vertex and edge)
- dynamic fault environments
- dynamic execution semantics possible (number of nodes executing per step in parallel)

Simulation Framework 2/5

- exact fault environments (specify distinct values for each vertex and edge)
- dynamic fault environments
- dynamic execution semantics possible (number of nodes executing per step in parallel)
- external fault injection and monitoring facilities

Simulation Framework 2/5

- exact fault environments (specify distinct values for each vertex and edge)
- dynamic fault environments
- dynamic execution semantics possible (number of nodes executing per step in parallel)
- external fault injection and monitoring facilities
- event logging (if needed)

Simulation Framework 2/5

- exact fault environments (specify distinct values for each vertex and edge)
- dynamic fault environments
- dynamic execution semantics possible (number of nodes executing per step in parallel)
- external fault injection and monitoring facilities
- event logging (if needed)
- choice of schedulers (three provided)

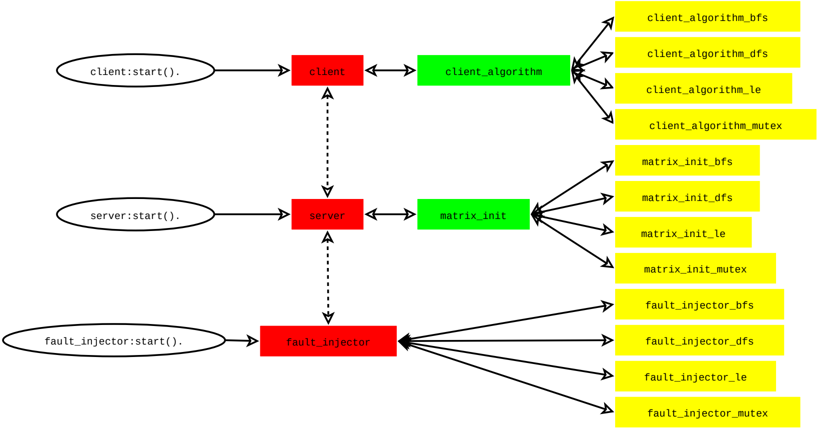
Simulation Framework 2/5

- exact fault environments (specify distinct values for each vertex and edge)
- dynamic fault environments
- dynamic execution semantics possible (number of nodes executing per step in parallel)
- external fault injection and monitoring facilities
- event logging (if needed)
- choice of schedulers (three provided)
- Load balancing (each client a lightweight process, can be mapped to any processor/computer)

Simulation Framework 3/5

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

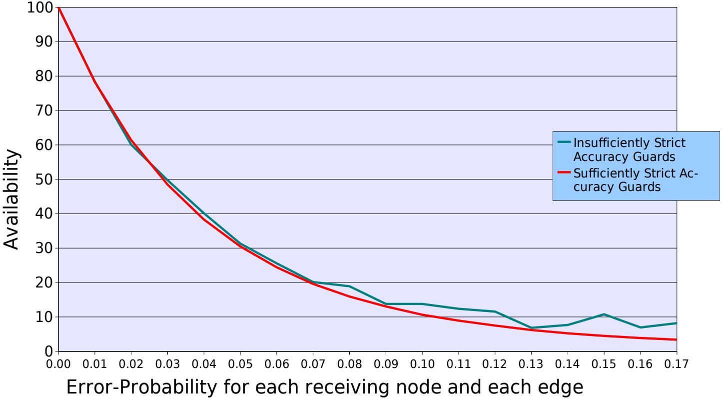
Simulation Framework 4/5



Accuracy 1/2

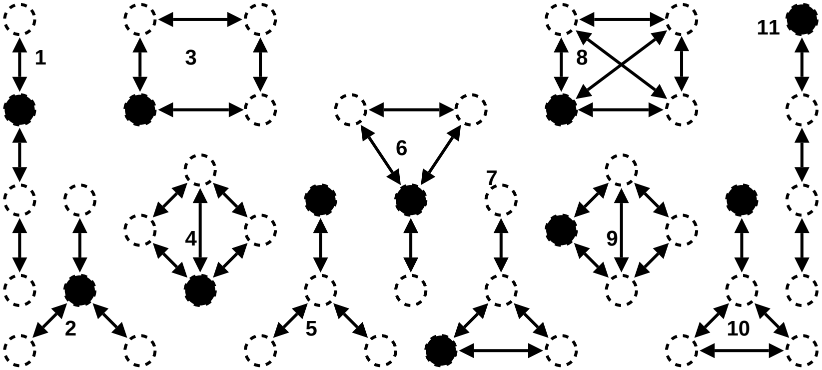
This figure exemplifies availability for first $20,000$ steps of an eight-processor system. The desired accuracy is reached if maximum the deviation within last n steps is lower than a certain threshold. The Results presented in the following feature about $1,000,000$ steps per system node.

Accuracy 2/2

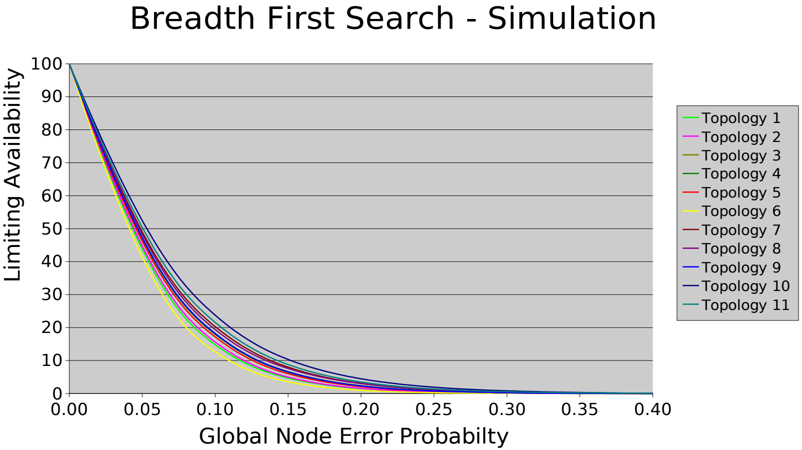


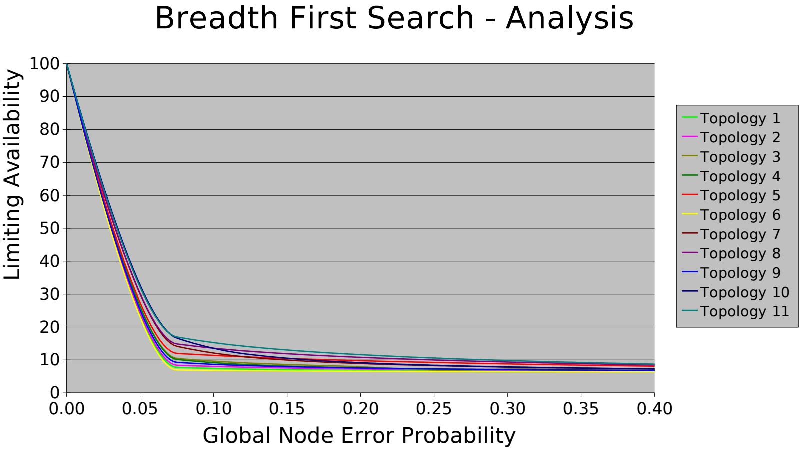
Strictness of accuracy guards is crucial for reliability of results!

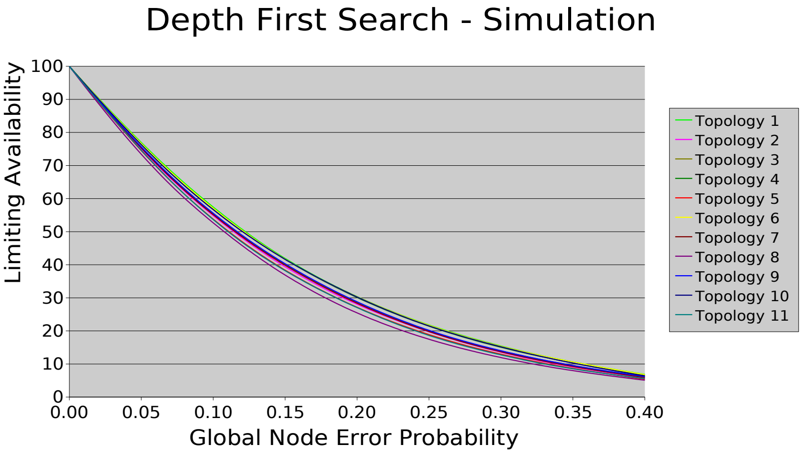
Test Case: All Possible 4-node Graphs

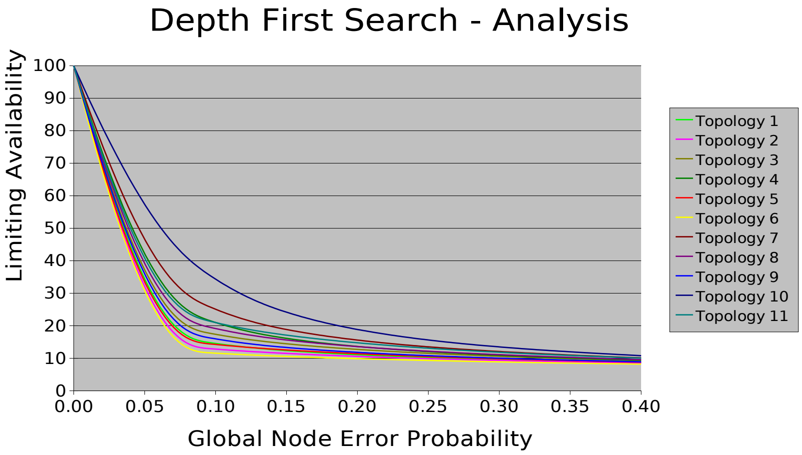


We chose *depth first search* (DFS) and *breadth first search* (BFS) algorithms for comparison with the analytic approach, executed on all possible 4-node graphs.


Breadth First Search - Simulation
Limiting Availability
100
90
80
70
60
50
40
30
20
10
0
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
Global Node Error Probabilty
Topology 1
Topology 2
Topology 3
Topology 4
Topology 5
Topology 6
Topology 7
Topology 8
Topology 9
Topology 10
Topology 11


Breadth First Search - Analysis
Limiting Availability
100
90
80
70
60
50
40
30
20
10
0
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
Global Node Error Probability
Topology 1
Topology 2
Topology 3
Topology 4
Topology 5
Topology 6
Topology 7
Topology 8
Topology 9
Topology 10
Topology 11


Depth First Search - Simulation
Limiting Availability
100
90
80
70
60
50
40
30
20
10
0
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
Global Node Error Probability
Topology 1
Topology 2
Topology 3
Topology 4
Topology 5
Topology 6
Topology 7
Topology 8
Topology 9
Topology 10
Topology 11


Depth First Search - Analysis
Limiting Availability
100
90
80
70
60
50
40
30
20
10
0
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
Global Node Error Probability
Topology 1
Topology 2
Topology 3
Topology 4
Topology 5
Topology 6
Topology 7
Topology 8
Topology 9
Topology 10
Topology 11

Conclusions

Derivation of fault tolerance measures by simulation

- reason: analytic method is insufficient

Conclusions

Derivation of fault tolerance measures by simulation

- reason: analytic method is insufficient
- method: simulation of self-stabilizing distributed algorithms

Conclusions

Derivation of fault tolerance measures by simulation

- reason: analytic method is insufficient
- method: simulation of self-stabilizing distributed algorithms
- features: modular design, scalability, performance, reliability of results

Barendregt, H. and Barendsen, E. (2000).
Introduction to lambda calculus.
In *Aspenäs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology.

Dhama, A., Theel, O., and Warns, T. (2006).
Reliability and Availability Analysis of Self-Stabilizing Systems.
In *8th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, page 17. Springer.

Dolev, S. (2000).
Self-Stabilization.
MIT Press.

Müllner, N., Dhama, A., and Theel, O. (2008).
Derivation of Fault Tolerance Measures of Self-Stabilizing Algorithms by Simulation.
In *ANSS '08: Proceedings of the 41st annual symposium on Simulation*, Ottawa, Ontario, Canada. IEEE Computer Society Press.

Schneider, M. (1993).
Self-stabilization.
ACM Comput. Surv., 25(1):45–67.

Trivedi, K. S. (1982).
Probability and Statistics with Reliability, Queuing and Computer Science Applications.
Prentice Hall PTR, Upper Saddle River, NJ, USA.