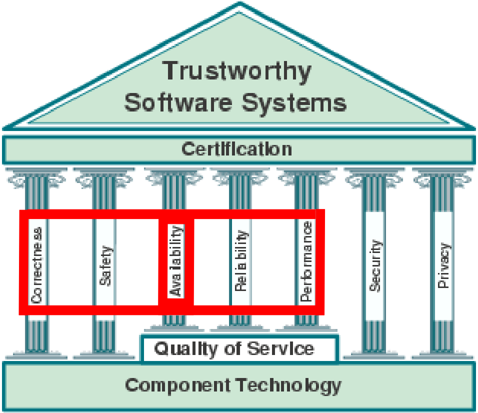
Unmasking Fault Tolerance: Masking vs. Non-masking Fault-tolerant Systems

Nils Müllner

nils.muellner@informatik.uni-oldenburg.de
Abteilung Systemsoftware und verteilte Systeme
Department für Informatik
Carl von Ossietzky Universität Oldenburg

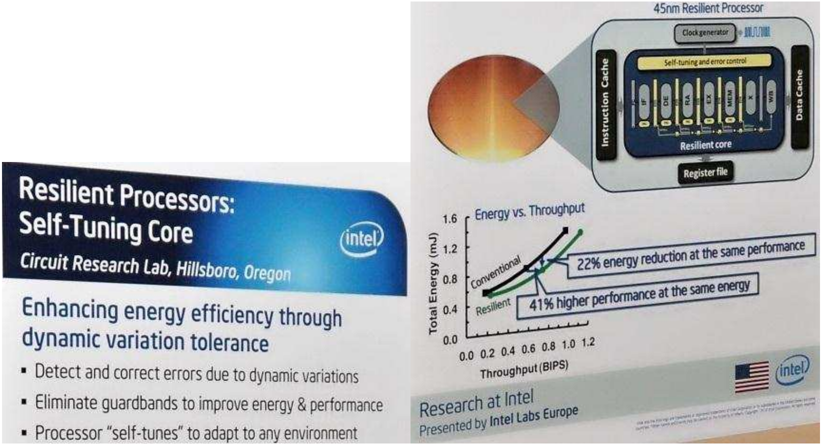
February 22, 2011

Orientation



Outline

Intel: Palisades



[BTL+10]

Focus: Basic Research

- fault tolerance in distributed systems
  is important for a variety of systems like CPU, WSN, ...

Focus: Basic Research

- fault tolerance in distributed systems
  is important for a variety of systems like CPU, WSN, ...
- focus: not **system specific** fault tolerance methods,
  but fundamental principles.

Focus: Basic Research

- fault tolerance in distributed systems
  is important for a variety of systems like CPU, WSN, ...

- focus: not **system specific** fault tolerance methods,
  but fundamental principles.

$\Rightarrow$: relation between quality (degree of masking)
and cost.

1. Motivation
2. **Basics**
3. Computation of *LWAV*
4. Lumping
5. Decomposition
6. Status and Outlook

Outline

1. fault tolerance demands redundancy
2. fault tolerance classification
3. the fault masker concept
4. unmasking fault tolerance
5. redundancy classification
6. self-stabilization

Fault Tolerance Demands Redundancy

- to tolerate faults, they must be detected and/or corrected
- detection and correction are functions that require resources
- typically either space (functional or information redundancy) or time (but commonly both)
- sometimes convertible (e.g., TMR)

Fault Tolerance Demands Redundancy

- to tolerate faults, they must be detected and/or corrected
- detection and correction are functions that require resources
- typically either space (functional or information redundancy) or time (but commonly both)
- sometimes convertible (e.g., TMR)

Example: **Cyclic Redundancy Checks (CRC)**
requires space (extends the package, information redundancy),
and more space (code for the computation of CRC, functional redundancy),
and time (for the computation, and transmission, temporal redundancy)

Three Kinds of FT.: Focus on Masking and Non-masking

	safe	not safe
live	masking	non-masking
not live	failsafe	intolerant

Table: Fault Tolerance Classes [KA97, Gär99]

Three Kinds of FT.: Focus on Masking and Non-masking

	safe	not safe	← detectors
live	masking	non-masking	
not live	failsafe	intolerant	

↑ correctors

Table: Fault Tolerance Classes [KA97, Gär99]

Three Kinds of FT.: Focus on Masking and Non-masking

	safe	not safe	$\leftarrow$ detectors
live	masking	non-masking	
not live	failsafe	intolerant	
$\uparrow$ correctors			

Table: Fault Tolerance Classes [KA97, Gär99]

non-masking fault tolerance

- requires correction
- relatively cheap

Three Kinds of FT.: Focus on Masking and Non-masking

	safe	not safe	← detectors
live	masking	non-masking	
not live	failsafe	intolerant	
↑ correctors			

Table: Fault Tolerance Classes [KA97, Gär99]

non-masking fault tolerance

- requires correction
- relatively cheap

masking fault tolerance

- requires detection and correction
- most desirable

Three Kinds of FT.: Focus on Masking and Non-masking

	safe	not safe	← detectors
live	masking	non-masking	
not live	failsafe	intolerant	
↑ correctors			

Table: Fault Tolerance Classes [KA97, Gär99]

non-masking fault tolerance

- requires correction
- relatively cheap

masking fault tolerance

- requires detection and correction
- most desirable

non-/masking fault tolerant with regards to a distinct fault class

Example: CRC

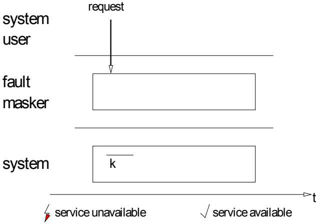
- intolerant: corrupted packet contained matching checksum
- non-masking fault tolerant: faults were detected, but could not be corrected / re-request violates temporal boundaries
- masking: correct transmission / or faults could be corrected on the spot

Example: CRC

- intolerant: corrupted packet contained matching checksum
- non-masking fault tolerant: faults were detected, but could not be corrected / re-request violates temporal boundaries
- masking: correct transmission / or faults could be corrected on the spot

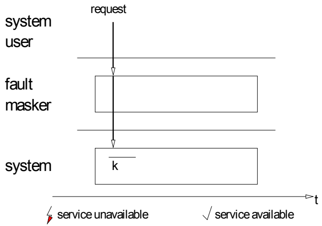
non-/masking fault tolerant with regards to a distinct fault class

The Fault Masker



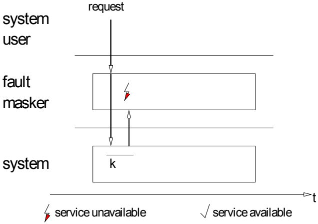
[MDT09]

The Fault Masker



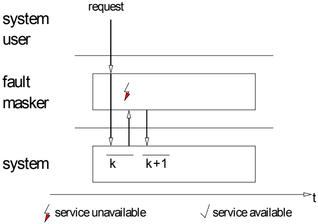
[MDT09]

The Fault Masker



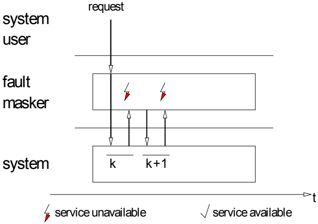
[MDT09]

The Fault Masker



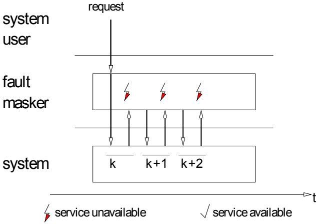
[MDT09]

The Fault Masker



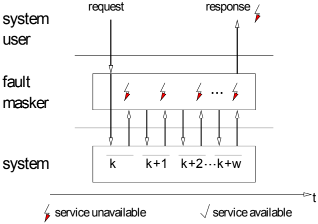
[MDT09]

The Fault Masker



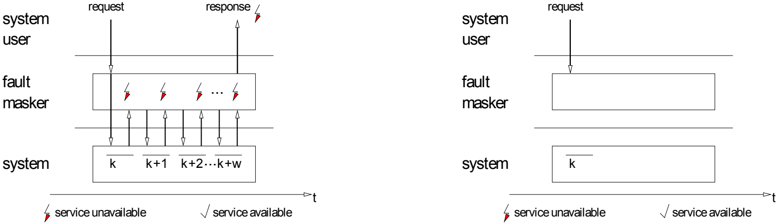
[MDT09]

The Fault Masker



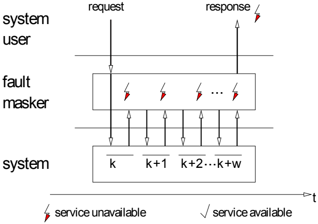
[MDT09]

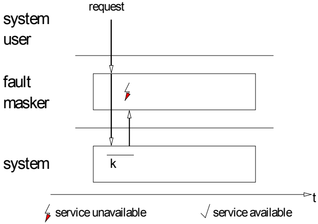
The Fault Masker



[MDT09]

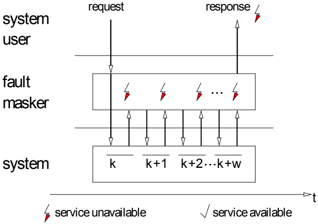
The Fault Masker

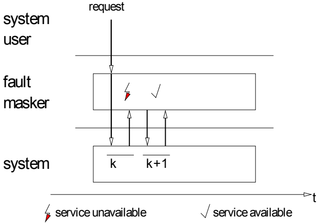




[MDT09]

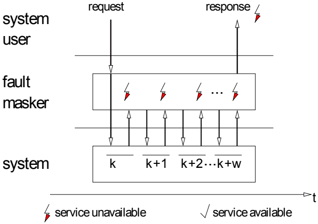
The Fault Masker

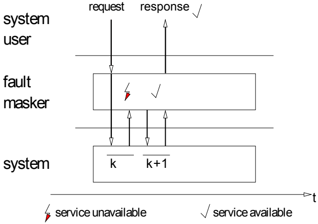




[MDT09]

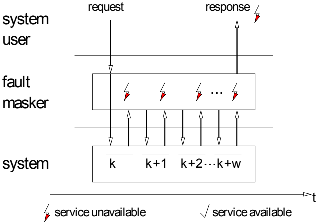
The Fault Masker

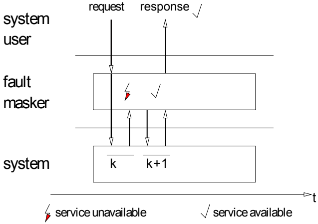




[MDT09]

The Fault Masker

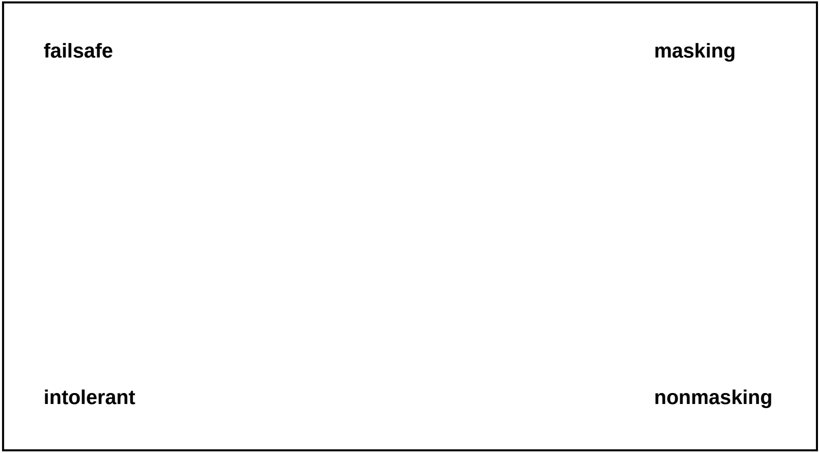




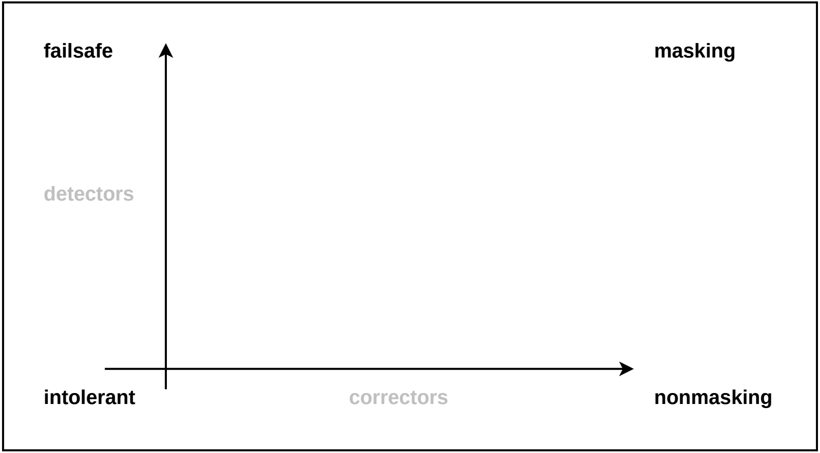
[MDT09]

the fault masker detects all faults

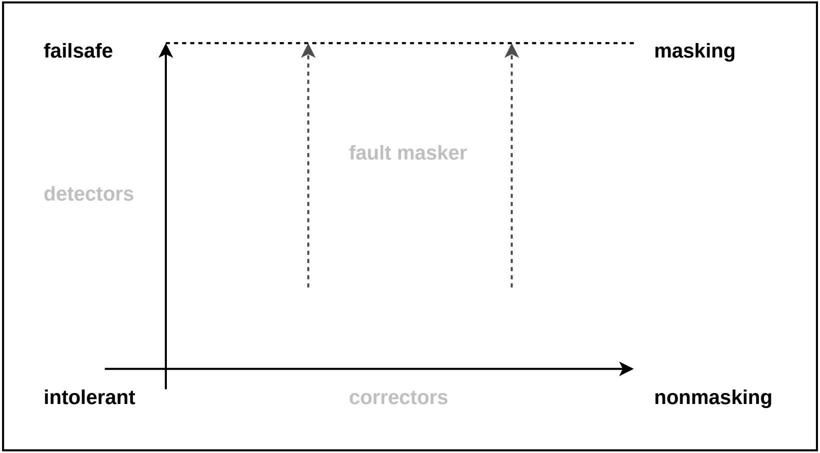
Detection ⇒ Safety, Correction ⇒ Liveness



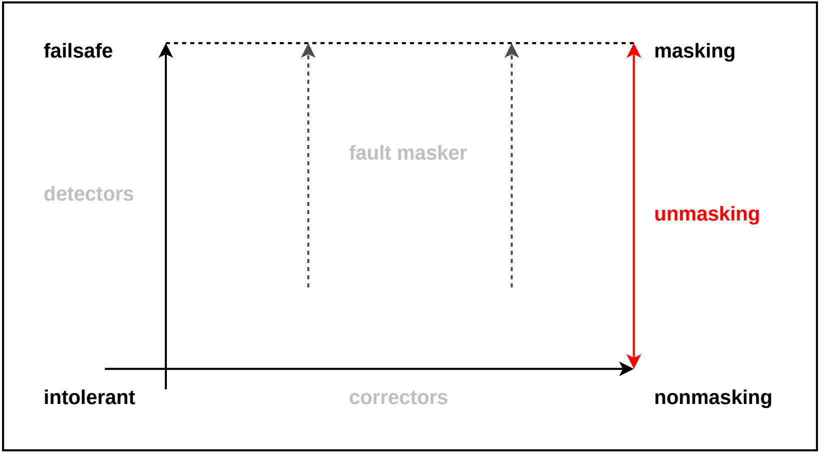
Detection ⇒ Safety, Correction ⇒ Liveness



Detection ⇒ Safety, Correction ⇒ Liveness



Detection ⇒ Safety, Correction ⇒ Liveness



Redundancy Establishes Detection and Correction

- information redundancy
  - error correcting or detecting codes
  - N-Modular Redundancy
- temporal Redundancy
  - self-stabilization
  - re-requests
  - N-Modular Redundancy

Focus: Correction Based on Temporal Redundancy (e.g., Self-Stabilization)

information redundancy: thoroughly discussed

- we can compute the quality of spatial redundancy (i.e., number and severity of faults covered, either in a masking or non-masking fashion)
- spatial redundancy commonly used to ensure data integrity

Focus: Correction Based on Temporal Redundancy (e.g., Self-Stabilization)

information redundancy: thoroughly discussed

- we can compute the quality of spatial redundancy (i.e., number and severity of faults covered, either in a masking or non-masking fashion)
- spatial redundancy commonly used to ensure data integrity

temporal redundancy (assuming detection as given):

- commonly used for system integrity
- how good can time heal/cure the system from faults?
- what is a proper metric?
- how can we calculate this metric?

Self-Stabilization

Definition (Self-Stabilization [Dol00, Dij74])

A system is self-stabilizing wrt. a safety predicate $\mathcal{P}$ iff:

1. *Starting from any state, it is guaranteed that the system will eventually reach a state that satisfies the safety predicate $\mathcal{P}$ (**convergence** property), provided that no fault happens.*
2. *Given that the system satisfies the safety predicate, it is guaranteed to stay in a state that satisfies the safety predicate $\mathcal{P}$ (**closure** property), provided that no fault happens.*

1. Motivation
2. Basics
3. Computation of *LWAV*
4. Lumping
5. Decomposition
6. Status and Outlook

A Suitable Metric 1/2

Definition (Limiting Window Availability (*LWA*))

Assume that at time $t = 0$, an initial system state distribution holds that corresponds to the steady state distribution of a system. Then, Limiting Window Availability of window size w (of this system), denoted by LWA_w, $w \geq 0$, is the probability that the system has at least once reached a state satisfying $\mathcal{P}$ within the following w time steps:

$$LWA_w = prob\{\exists k, 0 \leq k \leq w : c_k \models \mathcal{P}\}$$

w is called window size.

A Suitable Metric 2/2

Definition (Limiting Window Availability Vector (*LWAV*))

The limiting window availability vector of size i (of a system), denoted by $LWAV_i$, is an i-dimensional vector of probabilities. The element in the i^{th} position is the limiting window availability of window size $i-1$ of that system: $LWAV_i := \langle LWA_0, LWA_1, \ldots, LWA_{i-1} \rangle$.

A Suitable Metric 2/2

Definition (Limiting Window Availability Vector (*LWAV*))

The limiting window availability vector of size i (of a system), denoted by $LWAV_i$, is an i-dimensional vector of probabilities. The element in the i^{th} position is the limiting window availability of window size $i-1$ of that system: $LWAV_i := \langle LWA_0, LWA_1, \ldots, LWA_{i-1} \rangle$.

Definition (Limiting Window Availability Vector Gradient (*LWAVG*))

The limiting window availability vector gradient of size i (of a system), denoted by $LWAVG_i$, is an i-dimensional vector of probabilities. The element in the i^{th} position is the limiting window availability of window size i minus the limiting window availability of window size $i-1$ of that system:

$$LWAVG_i := \langle LWA_1 - LWA_0, LWA_2 - LWA_1, \ldots, LWA_i - LWA_{i-1} \rangle.$$

Test Set-Up: Algorithm and Topology

const **id** $:= 0$,
var **reg**,
repeat{
  reg $:= 0$}

Figure: Broadcast Sub-Algorithm for the Root Process

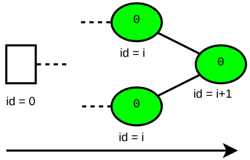
const **neighbors** $:= \{\pi_i, \ldots\}$,
const **id** $:= \min\{\mathbf{id}(\pi_i), \ldots\} + 1$,
var **reg**,
var **set** $:= \{\mathbf{reg}_i, \pi(\mathbf{reg}_i) \in \mathbf{neighbors} | \forall i : \mathrm{id}(\pi_i) = \mathrm{id} - 1\}$
repeat{
$\neg((\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2)$
$xor(\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0))$
  $\rightarrow \mathbf{reg} := 1$
$\square \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0$
  $\rightarrow \mathbf{reg} := 0$
$\square \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2$
  $\rightarrow \mathbf{reg} := 2$}

Figure: Broadcast Sub-Algorithm for Non-Root Processes

Test Set-Up: Algorithm and Topology

$\underline{\text{const}}$ $\mathbf{id} := 0$,
$\underline{\text{var}}$ $\mathbf{reg}$,
repeat$\{$
 $\mathbf{reg} := 0\}$

Figure: Broadcast Sub-Algorithm for the Root Process



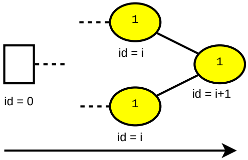
$\underline{\text{const}}$ $\mathbf{neighbors} := \{\pi_i, \ldots\}$,
$\underline{\text{const}}$ $\mathbf{id} := \min\{\mathbf{id}(\pi_i), \ldots\} + 1$,
$\underline{\text{var}}$ $\mathbf{reg}$,
$\underline{\text{var}}$ $\mathbf{set} := \{\mathbf{reg}_i, \pi(\mathbf{reg}_i) \in$
 $\mathbf{neighbors} | \forall i : \text{id}(\pi_i) = \text{id} - 1\}$
repeat$\{$
$\neg((\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2)$
$\mathit{xor}(\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0))$
 $\rightarrow \mathbf{reg} := 1$
$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0$
 $\rightarrow \mathbf{reg} := 0$
$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2$
 $\rightarrow \mathbf{reg} := 2\}$

Figure: Broadcast Sub-Algorithm for Non-Root Processes

Test Set-Up: Algorithm and Topology

<u>const</u> **id** $:= 0$,
<u>var</u> **reg**,
repeat{
  reg $:= 0$}

Figure: Broadcast Sub-Algorithm for the Root Process



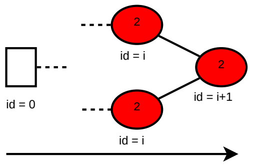
<u>const</u> **neighbors** $:= \{\pi_i, \ldots\}$,
<u>const</u> **id** $:= \min\{\mathbf{id}(\pi_i), \ldots\} + 1$,
<u>var</u> **reg**,
<u>var</u> **set** $:= \{\mathbf{reg}_i, \pi(\mathbf{reg}_i) \in$
  $\mathbf{neighbors} | \forall i : \mathrm{id}(\pi_i) = \mathrm{id} - 1\}$
repeat{
$\neg((\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2)$
$xor(\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0))$
  $\rightarrow \mathbf{reg} := 1$
$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0$
  $\rightarrow \mathbf{reg} := 0$
$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2$
  $\rightarrow \mathbf{reg} := 2$}

Figure: Broadcast Sub-Algorithm for Non-Root Processes

Test Set-Up: Algorithm and Topology

const **id** $:= 0$,
var **reg**,
repeat{
  reg $:= 0$}

Figure: Broadcast Sub-Algorithm for the Root Process



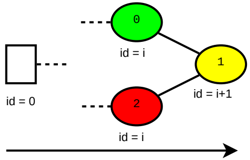
const **neighbors** $:= \{\pi_i, \ldots\}$,
const **id** $:= \min\{\mathbf{id}(\pi_i), \ldots\} + 1$,
var **reg**,
var **set** $:= \{\mathbf{reg}_i, \pi(\mathbf{reg}_i) \in$
  $\mathbf{neighbors} | \forall i : \mathrm{id}(\pi_i) = \mathrm{id} - 1\}$
repeat{
$\neg((\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2)$
$xor(\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0))$
  $\rightarrow \mathbf{reg} := 1$
$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0$
  $\rightarrow \mathbf{reg} := 0$
$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2$
  $\rightarrow \mathbf{reg} := 2$}

Figure: Broadcast Sub-Algorithm for Non-Root Processes

Test Set-Up: Algorithm and Topology

$\underline{\text{const}}$ $\mathbf{id} := 0$,
$\underline{\text{var}}$ $\mathbf{reg}$,
repeat$\{$
$\mathbf{reg} := 0\}$

Figure: Broadcast Sub-Algorithm for the Root Process



$\underline{\text{const}}$ $\mathbf{neighbors} := \{\pi_i, \ldots\}$,
$\underline{\text{const}}$ $\mathbf{id} := \min\{\mathbf{id}(\pi_i), \ldots\} + 1$,
$\underline{\text{var}}$ $\mathbf{reg}$,
$\underline{\text{var}}$ $\mathbf{set} := \{\mathbf{reg}_i, \pi(\mathbf{reg}_i) \in \mathbf{neighbors} | \forall i : \text{id}(\pi_i) = \text{id} - 1\}$
repeat$\{$
$\neg((\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2)$
$xor(\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0))$
$\rightarrow \mathbf{reg} := 1$
$\square \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0$
$\rightarrow \mathbf{reg} := 0$
$\square \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2$
$\rightarrow \mathbf{reg} := 2\}$

Figure: Broadcast Sub-Algorithm for Non-Root Processes

Test Set-Up: Algorithm and Topology

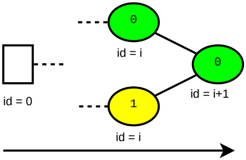
$\underline{\text{const}}$ $\mathbf{id} := 0,$

$\underline{\text{var}}$ $\mathbf{reg},$

repeat$\{$

$\quad\mathbf{reg} := 0\}$

Figure: Broadcast Sub-Algorithm for the Root Process



$\underline{\text{const}}$ $\mathbf{neighbors} := \{\pi_i, \ldots\},$

$\underline{\text{const}}$ $\mathbf{id} := \min\{\mathbf{id}(\pi_i), \ldots\} + 1,$

$\underline{\text{var}}$ $\mathbf{reg},$

$\underline{\text{var}}$ $\mathbf{set} := \{\mathbf{reg}_i, \pi(\mathbf{reg}_i) \in$
$\quad\mathbf{neighbors} | \forall i : \text{id}(\pi_i) = \text{id} - 1\}$

repeat$\{$

$\neg((\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2)$

$xor(\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0))$

$\quad\rightarrow \mathbf{reg} := 1$

$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0$

$\quad\rightarrow \mathbf{reg} := 0$

$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2$

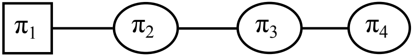
$\quad\rightarrow \mathbf{reg} := 2\}$

Figure: Broadcast Sub-Algorithm for Non-Root Processes

Test Set-Up: Algorithm and Topology

const **id** $:= 0$,
var **reg**,
repeat$\{$
  reg $:=$ 0$\}$

Figure: Broadcast Sub-Algorithm for the Root Process



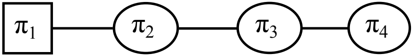
const **neighbors** $:= \{\pi_i, \ldots\}$,
const **id** $:= \min\{\mathbf{id}(\pi_i), \ldots\} + 1$,
var **reg**,
var **set** $:= \{\mathbf{reg}_i, \pi(\mathbf{reg}_i) \in$
  $\mathbf{neighbors} | \forall i : \mathrm{id}(\pi_i) = \mathrm{id} - 1\}$
repeat$\{$
$\neg((\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2)$
$xor(\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0))$
  $\rightarrow \mathbf{reg} := 1$
$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0$
  $\rightarrow \mathbf{reg} := 0$
$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2$
  $\rightarrow \mathbf{reg} := 2\}$

Figure: Broadcast Sub-Algorithm for Non-Root Processes

Test Set-Up: Algorithm and Topology

<u>const</u> **id** $:= 0$,
<u>var</u> **reg**,
repeat$\{$
  reg $:=$ 0$\}$

Figure: Broadcast Sub-Algorithm for the Root Process



$$c_t \models \mathcal{P} : \text{reg}_1 = 0 \wedge \text{reg}_2 = 0 \wedge \text{reg}_3 = 0 \wedge \text{reg}_4 = 0$$

<u>const</u> **neighbors** $:= \{\pi_i, \ldots\}$,
<u>const</u> **id** $:= \min\{\mathbf{id}(\pi_i), \ldots\} + 1$,
<u>var</u> **reg**,
<u>var</u> **set** $:= \{\mathbf{reg}_i, \pi(\mathbf{reg}_i) \in \mathbf{neighbors} | \forall i : \text{id}(\pi_i) = \text{id} - 1\}$
repeat$\{$
$\neg((\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2)$
$xor(\exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0))$
  $\rightarrow \mathbf{reg} := 1$
$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 0$
  $\rightarrow \mathbf{reg} := 0$
$\Box \exists \mathbf{reg}_i : \pi(\mathbf{reg}_i) \in \mathbf{set} \wedge \mathbf{reg}_i = 2$
  $\rightarrow \mathbf{reg} := 2\}$

Figure: Broadcast Sub-Algorithm for Non-Root Processes

The Resulting Markov Chain

$\downarrow$from/to$\rightarrow$	$\langle 0,0,0,0\rangle$	$\langle 0,0,0,2\rangle$	$\langle 0,0,2,0\rangle$	$\langle 0,2,0,0\rangle$	$\langle 2,0,0,0\rangle$
$\langle 0,0,0,0\rangle$	$p(e_1+e_2+e_3+e_4)$	qe_4	qe_3	qe_2	qe_1
$\langle 0,0,0,2\rangle$	pe_4	$p(e_1+e_2+e_3)+qe_4$			
$\langle 0,0,2,0\rangle$	pe_3		$p(e_1+e_2)+qe_3$		
$\langle 0,2,0,0\rangle$	pe_2			$p(e_1+e_4)+qe_2$	
$\langle 2,0,0,0\rangle$	pe_1				$p(e_3+e_4)+qe_1$
$\langle 0,0,2,2\rangle$		pe_3			
$\langle 0,2,0,2\rangle$		pe_2		pe_4	
$\langle 0,2,2,0\rangle$			pe_2		
$\langle 2,0,0,2\rangle$		pe_1			pe_4
$\langle 2,0,2,0\rangle$			pe_1		pe_3
$\langle 2,2,0,0\rangle$				pe_1	

Table: Transitions Grouped by Number of Operational Processes

e_i: probability, that π_i is elected for execution

q: probability, that a fault occurs

$p = 1 - q$

$e_1 = e_2 = e_3 = e_4 = 0.25,\ q = 0.01$

Compound	State	Steady State Probability
$\mathbb{0}$	$\langle 0,0,0,0 \rangle$	0.936254913358677
$\mathbb{1}$	$\langle 0,0,0,2 \rangle$	0.020767040703947
$\mathbb{1}$	$\langle 0,0,2,0 \rangle$	0.006443085000445
$\mathbb{1}$	$\langle 0,2,0,0 \rangle$	0.005896554367512
$\mathbb{1}$	$\langle 2,0,0,0 \rangle$	0.004721801275921
$\mathbb{2}$	$\langle 0,0,2,2 \rangle$	0.011734460936930
$\mathbb{2}$	$\langle 0,2,0,2 \rangle$	0.000103249069863
$\mathbb{2}$	$\langle 0,2,2,0 \rangle$	0.003596242185866
$\mathbb{2}$	$\langle 2,0,0,2 \rangle$	0.000101514623954
$\mathbb{2}$	$\langle 2,0,2,0 \rangle$	0.000028052478081
$\mathbb{2}$	$\langle 2,2,0,0 \rangle$	0.002411422793886
$\mathbb{3}$	$\langle 0,2,2,2 \rangle$	0.005204454376759
$\mathbb{3}$	$\langle 2,0,2,2 \rangle$	0.000049131622044
$\mathbb{3}$	$\langle 2,2,0,2 \rangle$	0.000042503806239
$\mathbb{3}$	$\langle 2,2,2,0 \rangle$	0.001243938539611
$\mathbb{4}$	$\langle 2,2,2,2 \rangle$	0.001401634860264

Table: Steady State Probability Distribution

How to Get There: State Space Analysis

1. compute transition probabilities between each pair of states
   - ⇒ (ergodic) Markov chain
2. compute steady state probability distribution
3. use steady state distribution as initial probability distribution for modified chain
4. transform set of legal states into sink
5. probability mass in set of legal states after i computation steps is LWA_i

The Markov Chain Yielding the *LWA*

↓from/to→	$\langle 0,0,0,0\rangle$	$\langle 0,0,0,2\rangle$	$\langle 0,0,2,0\rangle$	$\langle 0,2,0,0\rangle$	$\langle 2,0,0,0\rangle$
$\cancel{\langle 0,0,0,0\rangle}$	$\cancel{p(e_1+e_2+e_3+e_4)}$	$\cancel{qe_4}$	$\cancel{qe_3}$	$\cancel{qe_2}$	$\cancel{qe_1}$
$\langle 0,0,0,0\rangle$	1	0	0	0	0
$\langle 0,0,0,2\rangle$	pe_4	$p(e_1+e_2+e_3)+qe_4$			
$\langle 0,0,2,0\rangle$	pe_3		$p(e_1+e_2)+qe_3$		
$\langle 0,2,0,0\rangle$	pe_2			$p(e_1+e_4)+qe_2$	
$\langle 2,0,0,0\rangle$	pe_1				$p(e_3+e_4)+qe_1$
$\langle 0,0,2,2\rangle$		pe_3			
$\langle 0,2,0,2\rangle$		pe_2		pe_4	
$\langle 0,2,2,0\rangle$			pe_2		
$\langle 2,0,0,2\rangle$		pe_1			pe_4
$\langle 2,0,2,0\rangle$			pe_1		pe_3
$\langle 2,2,0,0\rangle$				pe_1	

Table: Transitions Grouped by Number of Operational Processes

Limitations

- computation works for example
- but what about larger systems?
  - state space explosion is obvious
- solution: two ways
  - lumping
  - decomposition

Markov Chain Abstraction (Lumping)

- goal: smaller Markov chains

Markov Chain Abstraction (Lumping)

- goal: smaller Markov chains

- lumping aggregates states and transitions

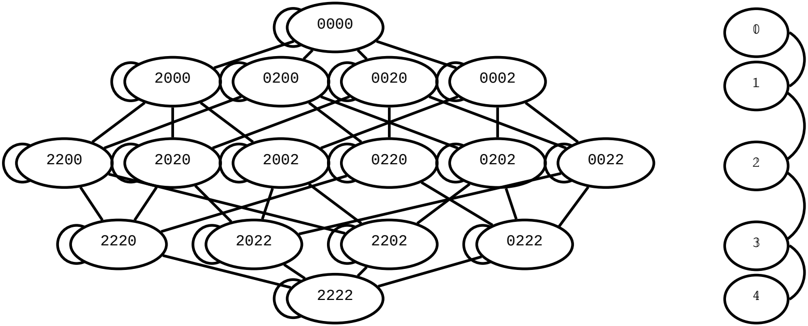
Markov Chain Abstraction (Lumping)

- goal: smaller Markov chains
- lumping aggregates states and transitions
- question: what states (and transitions) can be lumped still being the *LWAV*?

Markov Chain Abstraction (Lumping)

- goal: smaller Markov chains
- lumping aggregates states and transitions
- question: what states (and transitions) can be lumped still being the *LWAV*?
- answer (for this example): all states that have the same amount of incorrect processes

Lumping Example 1/3



Lumping Example 2/3

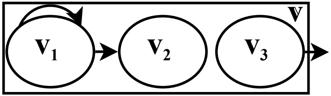
Lumping aggregates states and transitions.

Lumping Example 2/3

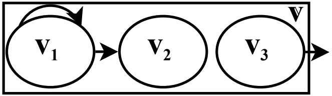
Lumping aggregates states and transitions.

$$prob(\overrightarrow{\mathbb{v},\mathbb{w}}) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} p(\overrightarrow{v_i, w_j}) \cdot p(v_i)}{\sum_{i=0}^{n} p(v_i)}$$

Lumping Example 3/3

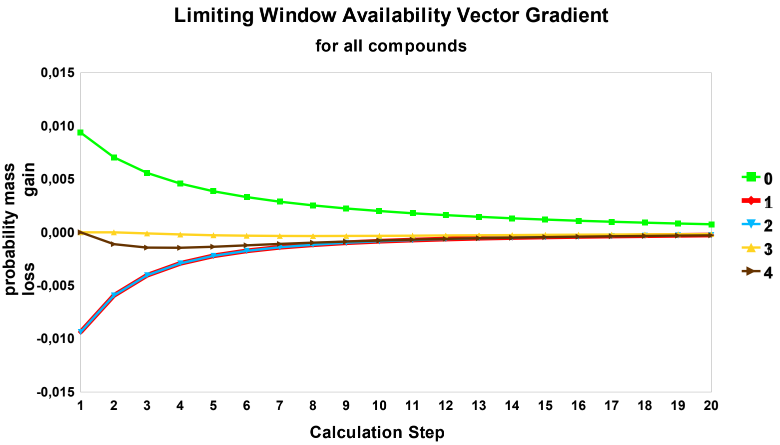


Lumping Example 3/3

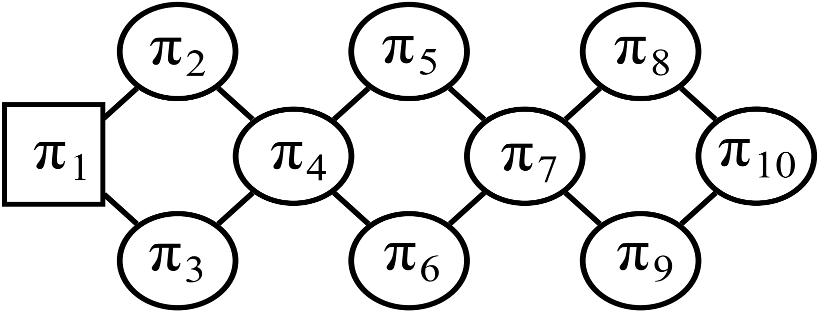


$$p(\overrightarrow{\mathbb{V}, \mathbb{V}}) = \frac{p(\overrightarrow{v1, v1}) \cdot p(v1) + p(\overrightarrow{v1, v2}) \cdot p(v1)}{p(v1) + p(v2) + p(v3)}$$

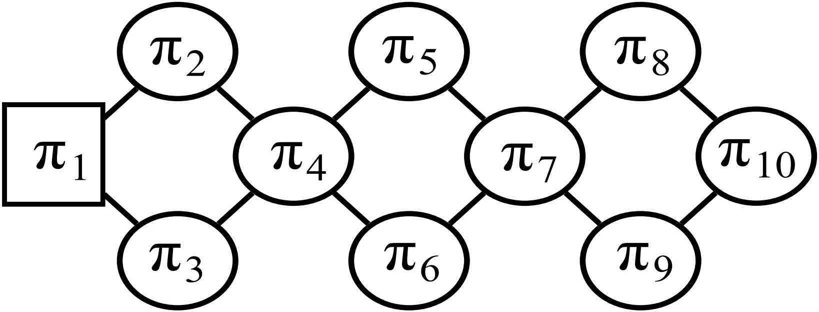
Small Example: Result



LWA at Large



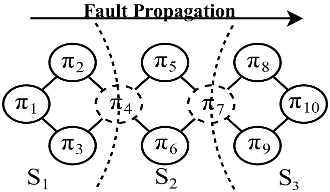
LWA at Large

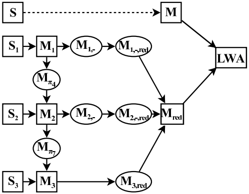


17496 state Markov chain

Decomposing and Lumping

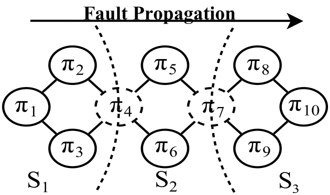
- lumping aggregates states that have something in common

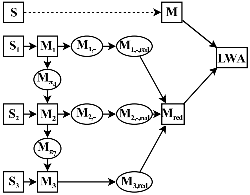




Decomposing and Lumping

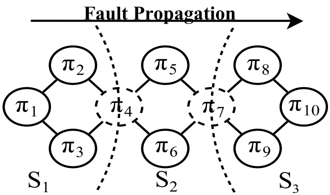
- lumping aggregates states that have something in common
- here: lumping of states that have the same amount of defective processes in common

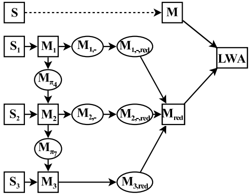




Decomposing and Lumping

- lumping aggregates states that have something in common
- here: lumping of states that have the same amount of defective processes in common
- decomposition allows the construction of (much) smaller sub-Markov chains

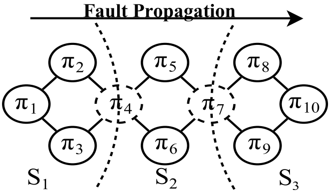


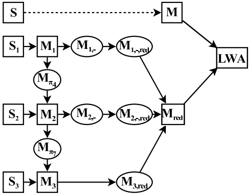


Decomposing and Lumping

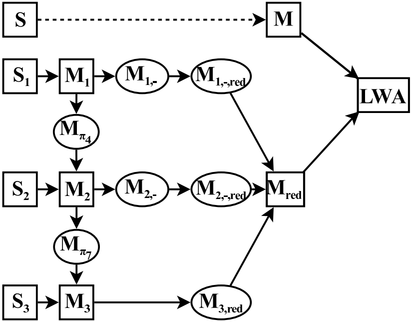
- lumping aggregates states that have something in common
- here: lumping of states that have the same amount of defective processes in common

- decomposition allows the construction of (much) smaller sub-Markov chains
- recomposition of smaller lumped Markov chains yields the exact result (80 instead of 17496 states)



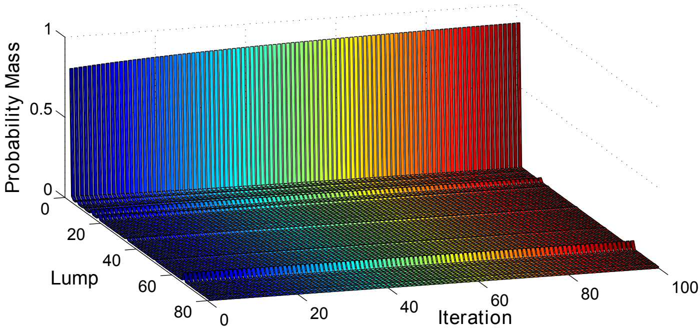


Decomposition Scheme

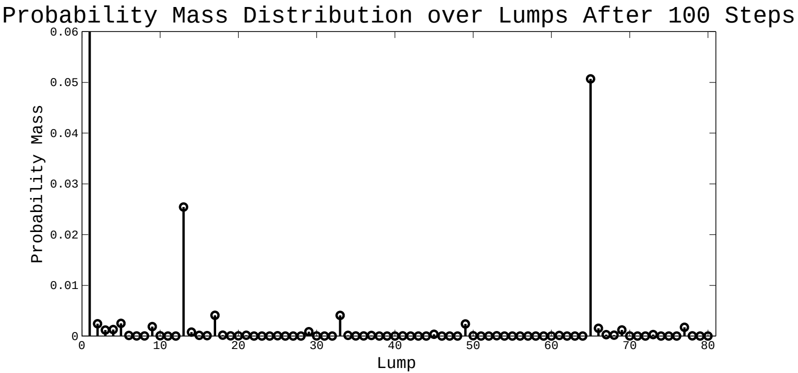


M	comprises 17496 states	$M_{1,-}$	comprises 8 states
M_1	comprises 24 states	$M_{2,-}$	comprises 27 states
M_2	comprises 81 states	$M_{1,-,red}$	comprises 3 states
M_3	comprises 81 states	$M_{2,-,red}$	comprises 3 states
M_{π_4}	comprises 3 states	$M_{3,red}$	comprises 4 states
M_{π_7}	comprises 3 states	M_{red}	comprises 80 states

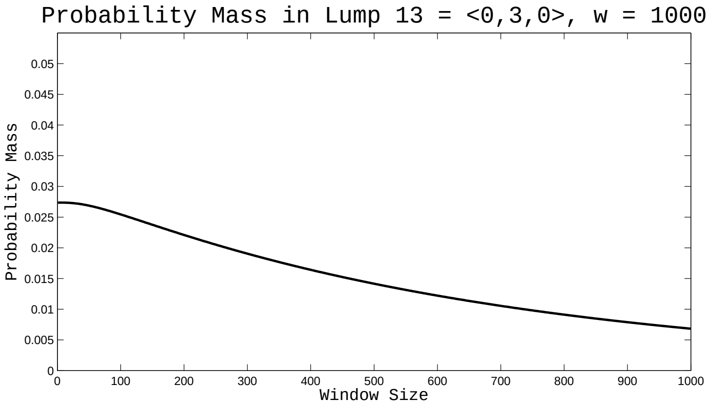
LWAV Over lumps



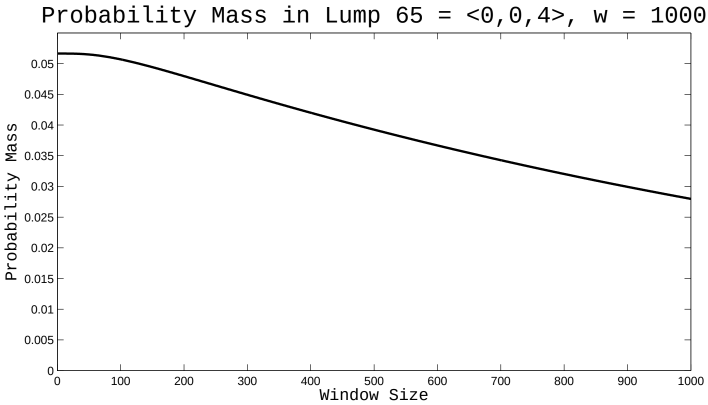
LWAV Over lumps



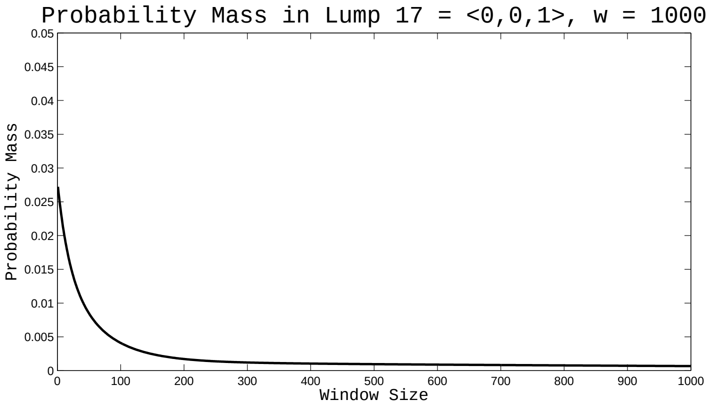
LWAV Over lumps



LWAV Over lumps



LWAV Over lumps



Hierarchical Towards Heterarchical Systems 1/2

- fault propagation unidirectional

Hierarchical Towards Heterarchical Systems 1/2

- fault propagation unidirectional
- decomposition easy: no cyclic dependencies

Hierarchical Towards Heterarchical Systems 1/2

- fault propagation unidirectional
- decomposition easy: no cyclic dependencies
- what about any-way propagation

Hierarchical Towards Heterarchical Systems 2/2

- hierarchical self-stabilizing systems demand a hierarchy (order) among the processes. Fault propagation strictly occurs from root towards leafs.

Hierarchical Towards Heterarchical Systems 2/2

- hierarchical self-stabilizing systems demand a hierarchy (order) among the processes. Fault propagation strictly occurs from root towards leafs.

- semi-hierarchical self-stabilizing systems possess the ability to dynamically reassign the role of the root. Switching the root is called an epoch. Fault propagation during an epoch is unidirectional.

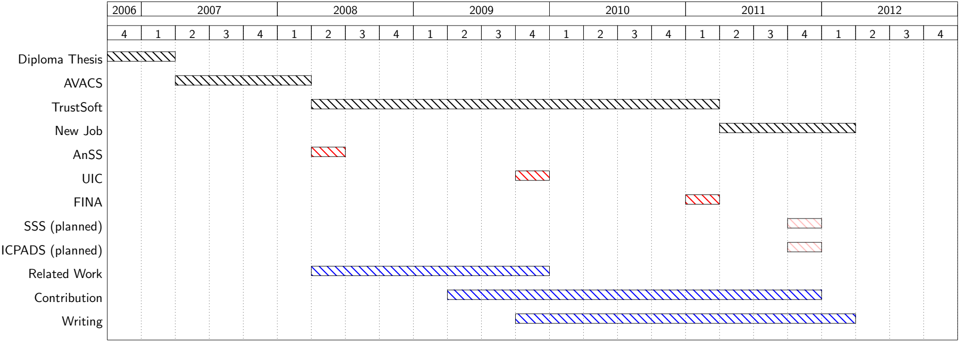
Hierarchical Towards Heterarchical Systems 2/2

- hierarchical self-stabilizing systems demand a hierarchy (order) among the processes. Fault propagation strictly occurs from root towards leafs.

- semi-hierarchical self-stabilizing systems possess the ability to dynamically reassign the role of the root. Switching the root is called an epoch. Fault propagation during an epoch is unidirectional.

- heterarchical self-stabilizing systems achieve their goal in the absence of any order among the processes. Fault propagation can occur in any direction at any time.

Timeline


2006
2007
2008
2009
2010
2011
2012
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
Diploma Thesis
AVACS
TrustSoft
New Job
AnSS
UIC
FINA
SSS (planned)
ICPADS (planned)
Related Work
Contribution
Writing

Current Focus

- FINA - 22.-25. March
  - *LWA*, *LWAV*, and *LWAVG*
  - the computation thereof,
  - basics of lumping
    ⇒ will be presented next month at 7^{th} Int'l Symposium on Frontiers of Systems and Network Applications
- SSS - 22. April: system decomposition of hierarchical self-stabilizing systems
- ICPADS - 24. June: system decomposition of heterarchical self-stabilizing systems
  either by iterations, or maybe flow equations...
- writing it up

Unmasking Fault Tolerance

goal: determination of the sweet spot

- be as masking as possible
- with as little effort as possible

Unmasking Fault Tolerance

goal: determination of the sweet spot

- be as masking as possible = maximize degree of masking fault tolerance
- with as little effort as possible = minimize time and space redundancy

Unmasking Fault Tolerance

goal: determination of the sweet spot

- be as masking as possible = maximize degree of masking fault tolerance
- with as little effort as possible = minimize time and space redundancy

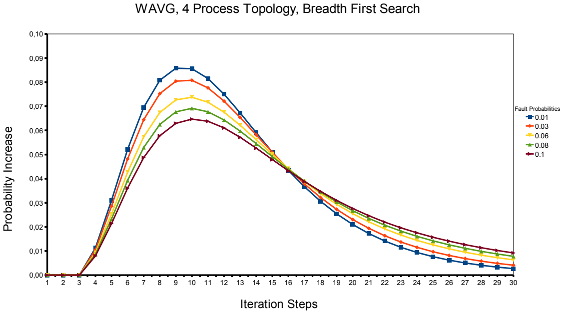
$\Rightarrow$ determination of the optimal trade-off thereof

Unmasking Fault Tolerance

goal: determination of the sweet spot

- be as masking as possible = maximize degree of masking fault tolerance
- with as little effort as possible = minimize time and space redundancy

⇒ determination of the optimal trade-off thereof



Edsger W. Dijkstra.
Self-Stabilizing Systems in Spite of Distributed Control.
Commun. ACM, 17(11):643–644, 1974.

Shlomi Dolev.
Self-Stabilization.
MIT Press, Cambridge, MA, USA, 2000.

Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita.
Weak vs. Self vs. Probabilistic Stabilization.
In ICDCS'08: Proc. of the 28th International Conference on Distributed Computing Systems, pages 681–688, Washington, DC, USA, 2008. IEEE Computer Society.

Felix C. Gärtner.
Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments.
ACM Computing Surveys, 31(1):1–26, 1999.

Sandeep S. Kulkarni and Anish Arora.
Compositional Design of Multitolerant Repetitive Byzantine Agreement.
Lecture Notes in Computer Science, 1346:169–182, 1997.

A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y-R. Choi, T. Herman, S. S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita.
A Line in the Sand: A Wireless Sensor Network for Target Detection, Classification, and Tracking.
Computer Networks, pages 605–634, 2004.

Keith A. Bowman, James W. Tschanz, Shih-Lien L. Lu, Paolo A. Aseron, Muhammad M. Khellah, Arijit Raychowdhury, Bibiche M. Geuskens, Carlos Tokunaga, Chris B. Wilkerson, Tanay Karnik, and Vivek K. De.
Resilient Microprocessor Design for High Performance & Energy Efficiency.
In ISLPED'10: Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design, pages 355–356, New York, NY, USA, 2010. ACM.

Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber.
DRAM Errors in the Wild: A Large-Scale Field Study.
In SIGMETRICS'09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, pages 193–204, New York, NY, USA, 2009. ACM.

Nils Müllner, Abhishek Dhama, and Oliver Theel.
Derivation of Fault Tolerance Measures of Self-Stabilizing Algorithms by Simulation.
In AnSS'08: Proceedings of the 41st Annual Symposium on Simulation, pages 183–192. IEEE Computer Society Press, April 2008.

Nils Müllner, Abhishek Dhama, and Oliver Theel.
Deriving a Good Trade-off Between System Availability and Time Redundancy.
In Proceedings of the Symposia and Workshops on Ubiquitious, Automatic and Trusted Computing, number E3737, pages 61–67. IEEE Computer Society Press, July 2009.

Nils Müllner and Oliver Theel.
The Degree of Masking Fault Tolerance vs. Temporal Redundancy.
To appear, In Proceedings of the 2011 IEEE 25th International Conference on Advanced Information Networking and Applications Workshops, FINA'11, Singapore, 2011. IEEE Computer Society.

Unmasking Fault Tolerance: Masking vs. Non-masking Fault-tolerant Systems

Nils Müllner

nils.muellner@informatik.uni-oldenburg.de
Abteilung Systemsoftware und verteilte Systeme
Department für Informatik
Carl von Ossietzky Universität Oldenburg

February 22, 2011