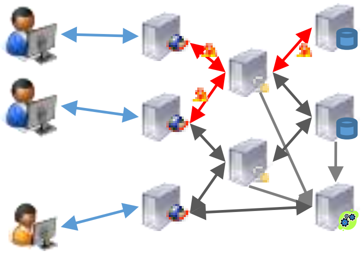


Verteilte Systeme/ Distributed Systems

Artur Andrzejak

10

Self-stabilizing Systems

Stabilizing Algorithms - Intuition

- An algorithm is **stabilizing** if it can finally bring the system to a correct state (according to the specification) regardless of the initial configuration
- Stabilizing algorithms are "optimistic“
  - They assume that intermittent errors can occur, which then disappear (transient errors)
  - The correct processes react without checks to the (incorrect) messages, which can cause arbitrary corrupt system states
  - However, it is assumed that any process will eventually return to normal execution
  - The "stabilization property" of the algorithm causes the system to finally transition to a correct configuration (= state)

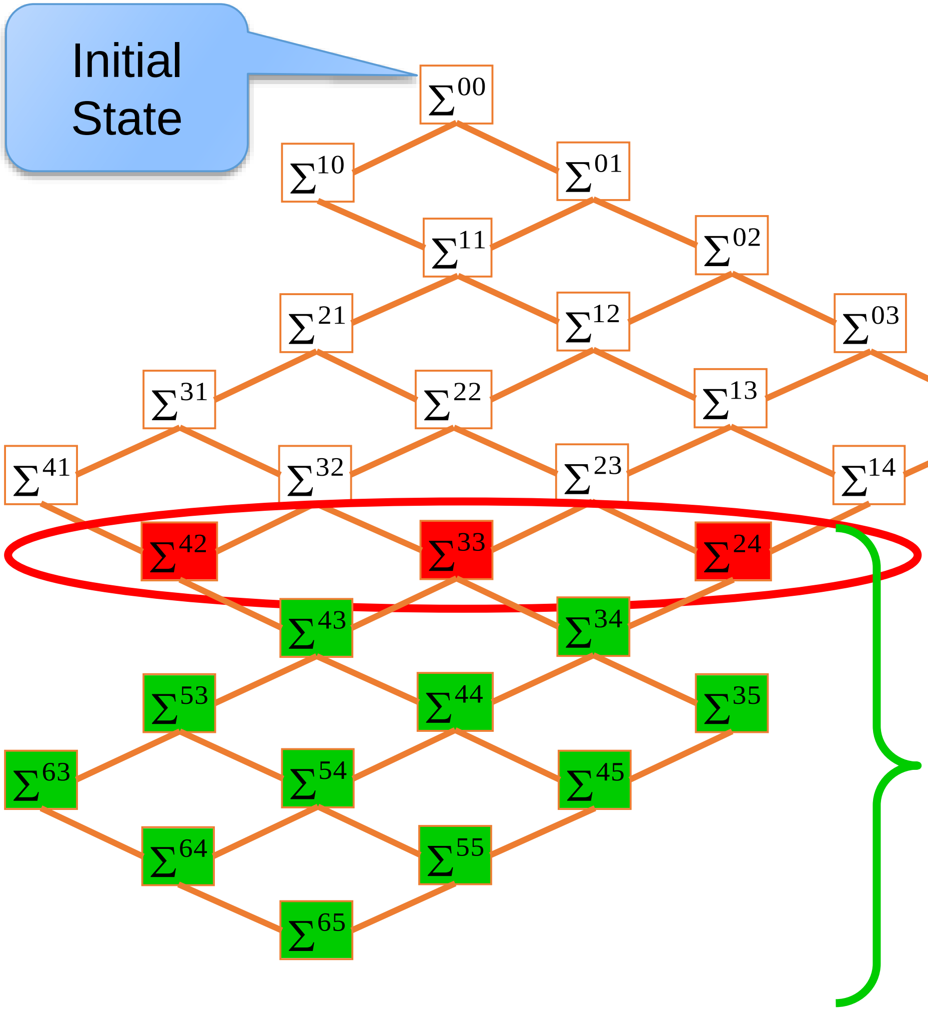
System, Execution, Specification

- A **system** is a pair $S=(C, \rightarrow)$, where
  - C is a set of ***configurations*** (i.e. global states)
  - "$\rightarrow$" is a binary relation on C, a **state transition relation**
    - I.e. $g_i \rightarrow g_{i+1}$ holds if one can move from the state g_i to g_{i+1}
- An **execution** of S is a maximal sequence $g_0, g_1, ..$, so that:
  - $g_i \rightarrow g_{i+1}$ holds for all $i \geq 0$
- The desired consistent behavior of the processes is modeled as a predicate **P** on the <u>sequences</u> of configurations
- **P** is called a **specification**
  - Example of P: "each configuration is the same"
    - I.e. one execution satisfies P when system enters a configuration and never leaves it again
    - Note: to verify P, we must consider a sequence of configurations!

Stabilization

- A system S **stabilizes** to the specification **P** if there is a set **L** $\subseteq$ C of so-called **legitimate configurations** with the following properties:
- Correctness:
  - Any execution starting from a configuration in **L** satisfies **P**
    - I.e. Each configuration in **L** can be seen as a "reset to a clean state", after which **P** holds
- Convergence:
  - Each execution contains a configuration of **L**
    - I.e. No matter how we execute, we will "pass" through a "cleaning configuration" from **L**

Example: a Stabilizing System



1. Correctness?
2. Convergence?

Legitimate configurations

Every execution over these configurations fulfills **P**

Good Properties of Stabilizing Systems

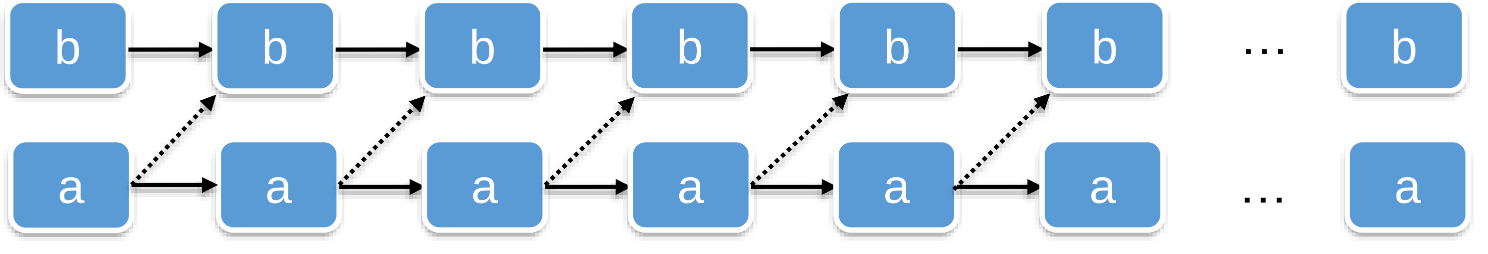
- Fault tolerance
  - Full and automatic protection against all transient process errors, because the system finds itself in a correct state from any - and also so corrupt - configuration
- Initialization
  - The need for consistent initialization is eliminated because the processes can start in arbitrary states and ultimately achieve coordinated behavior
- Dynamic topology
  - A stabilizing algorithm that is topology-dependent (e.g. uses routing tables) converges to a new solution after the occurrence of a topological change

Bad Properties of Stabilizing Systems

- Initial inconsistency
  - Before a *legitimate configuration is* achieved, the algorithm may have an inconsistent output
- Inefficiency
  - Self-stabilizing algorithms are typically less efficient than classical algorithms for the same problem
- Ignorance of the stable state
  - It is not possible to determine within the system that a legitimate configuration has been achieved
  - Therefore, the processes can not tell whether their behavior has become reliable

Pseudo-Stabilizing Systems

- Theorem : *If a system stabilizes to **P**, then each execution has a non-empty suffix that satisfies **P**.*
  - Suffix = final part („Nachsilbe“, „Endsilbe“)
- A system is **pseudo-stabilizing** to **P** if each execution has a non-empty suffix that satisfies **P**
- Example
  - System S with configs a, b and transitions a $\rightarrow$ a, a $\rightarrow$ b, b $\rightarrow$ b
  - Let **P** be the predicate "*all configurations are the same*"
  - Is S pseudo-stabilizing? Is S stabilizing?



Each execution has a suffix of the same configuration => S is pseudo-stabilizing

Pseudo-Stabilizing Systems

- Pseudo-stabilization: A weaker property than stabilizing systems
- Good: one can find pseudo-stabilizing algorithms for problems that do not have stabilizing algorithms
  - Example: Transferring a sequence of data
- Bad: For some pseudo-stabilizing systems there is no limit on the number of steps to reach the correct state
- For stabilizing systems such a barrier can always be specified

Mutual Exclusion / Wechselseitiger Ausschluss

Recall: Dijkstra's Token Ring

System model and conventions

- Let *N* be the number of processes
- A state $\boldsymbol{s_i}$ of process i is an integer $s_i \in \{0, \dots, K-1\}$
  - Here *K* is an arbitrary integer larger *N* $(K > N)$
- Process $i > 0$ can read $\boldsymbol{s_{i-1}}$ (and of course $\boldsymbol{s_i}$)
- Process $i = 0$ can read $\boldsymbol{s_{N-1}}$ (and of course $\boldsymbol{s_0}$)

Privilege rules (i.e. right to execute critical section):

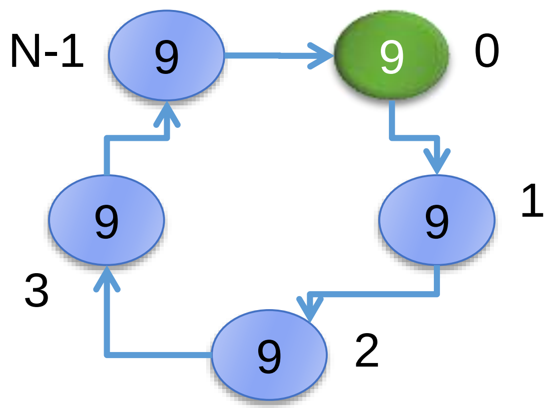
- Process $i > 0$ has a privilege iff $\boldsymbol{s_{i-1} \neq s_i}$
- Process $i = 0$ has a privilege iff $\boldsymbol{s_0 = s_{N-1}}$

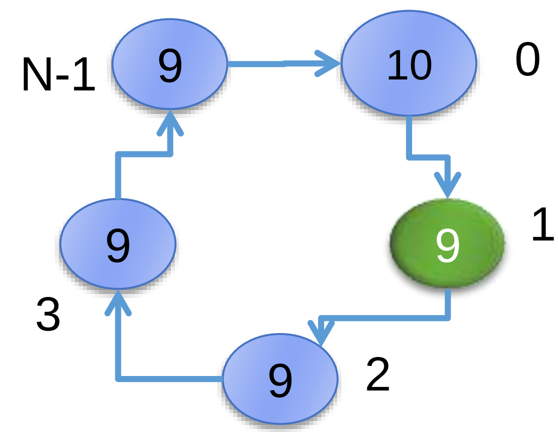
Algorithm

- A privileged process can change its state only via:
  - For $i > 0$: set $\boldsymbol{s_i \coloneqq s_{i-1}}$
  - For $i = 0$: set $\boldsymbol{s_0} \coloneqq (\boldsymbol{s_{N-1} + 1}) \bmod \boldsymbol{K}$
- Note: by this change a process loses its privilege

Recall: Dijkstra‘s Token Ring - Example

Case 1: Here $K = 11, \mathrm{N} = 5$ Case 2:





Privilege rules (i.e. right to execute critical section):

- Process $i > 0$ has a privilege iff $\boldsymbol{s_{i-1}} \neq \boldsymbol{s_i}$
- Process $i = 0$ has a privilege iff $\boldsymbol{s_0} = \boldsymbol{s_{N-1}}$

Algorithm

- A privileged process can change its state only via:
  - For $i > 0$: set $\boldsymbol{s_i} \coloneqq \boldsymbol{s_{i-1}}$
  - For $i = 0$: set $\boldsymbol{s_0} \coloneqq (\boldsymbol{s_{N-1}} + \mathbf{1}) \bmod \boldsymbol{K}$

Dijkstra‘s Token Ring: Observations

- Obs. A: *At least one process has the KS privilege*
  - If no process $i > 0$ has privilege, then $s_i = s_{i-1}$ for all $i > 0$, so that $s_0 = s_{N-1}$, and thus process 0 has privilege

- Obs. B: *The number of privileged processes never increases*
  - The only process that can be privileged is the successor of a privileged process that gives up its privilege

Dijkstra's Token Ring: Observations

- Obs. A: *At least one process has the KS privilege*
  - If no process $i > 0$ has privilege, then $s_i = s_{i-1}$ for all $i > 0$, so that $s_0 = s_{N-1}$, and thus process 0 has privilege
- Obs. B: *The number of privileged processes never increases*
  - The only process that can be privileged is the successor of a privileged process that gives up its privilege

- Let the set of legitimate configurations **L** contain the configurations in which exactly one process has the privilege
  - Configuration here = a union of states of all processes

Lemma: L is OK

- Let the set of legitimate configurations L contain the those configurations in which exactly one process has the privilege
  - Configuration here = a union of states of all processes
- What is our specification P here?
  - *safety*: only one process is privileged in any configuration (of a configuration sequence)
  - *liveness*: Any request to become privileged is (eventually) successful
    - Note: It is essential that P "works" on sequences of config's!
- **Lemma**: *Executions that start in a legitimate configuration meet the (problem) specification **P***

Lemma: L is OK

- **Lemma**: *Executions that start in a legitimate configuration meet the (problem) specification* ***P***
- **Proof** :
  - In a configuration of **L**, only the (one) privileged process can take a step
  - It loses the privilege, but - since there is no configuration without privilege - his successor becomes privileged
  - Thus, there is always at least one privileged process and each process gets the privilege infinitely often (all N configurations)
    - Assuming that we take steps at all…

Lemma: Dijkstra's Token Ring Converges

- **Lemma**: *The algorithm converges to* ***L*** *(i.e. each execution contains a configuration of* ***L****)*
  - So we always reach to a state where just one process is privileged.
- **Proof**:

1. Process 0 performs infinite many steps:

   A. There can be at most $N(N-1)/2$ steps without process 0:

   - Consider the function $F(\text{config}) = \sum_{i \in S}(N-i)$
     - where $S = \{i : i > 0 \text{ and } i \text{ is privileged}\}$
     - *F*'s argument is the current configuration
   - B. *F* becomes *smaller* with each step of a process $i > 0$, as the privilege goes from i to $i+1$
   - C. Obviously $F(\text{config}) < N(N-1)/2$
   - From B. and C. we follow A.

Dijkstra‘s Token Ring Converges /2

- Proof (2):

2. Process 0 reaches a state ***w*** which did not occur in the initial configuration g_0 after at most *N* of (its own) steps:
   - In g_0 there are at most *N* different values of states (different numbers among $s_0, s_1, \dots, s_{N-1}$), so there are still $K - N > 0$ other possible states "outside" g_0
   - Process 0 increases its state modulo *K* at each of its steps, so it will sometime reach a state ***w*** not in g_0 (a bit more thinking required…)

3. When process 0 reaches such a state ***w*** for the first time, ***w*** does not occur as a state of any other process in the ring:
   - Since other processes only copy states, they can only have states from g_0

Dijkstra's Token Ring Converges /3

- Proof (3):

4. When process 0 gets the privilege the next time, the configuration satisfies $w = s_0 = s_1 =, \ldots, = s_{N-1}$ and is thus in **L**
    - While process 0 has reached the state ***w*** for the first time, it has lost a privilege and process 1 obtained it
    - Then process 1 lost own privilege by copying value ***w*** from process 0
    - Then process 2 lost own privilege by copying value ***w*** from process 1
    - …
    - Finally, process $N-1$ got the value ***w*** from process $N-2$
        - Now $\boldsymbol{s_0 = s_{N-1}}$ and process 0 is privileged again
    - Note that we have only copied since process 0 has reached the state ***w*** , so statement #4 is correct!

Dijkstra's Token Ring Converges /4

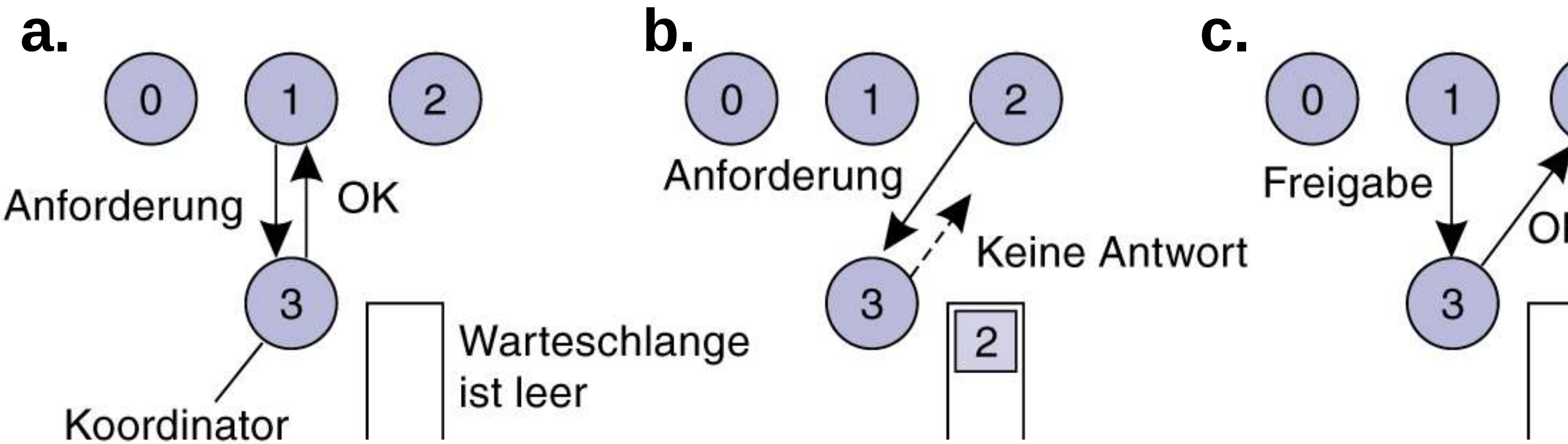
- Proof (4):

5. This will allow us to achieve a legitimate configuration after at most $N + 1$ steps of process 0
6. Each step of process 0 occurs after at most $N(N-1)/2$ steps "primitive" steps (statement #1), so the number of steps to stabilization is $O(N^3)$
7. A more detailed analysis can show that this runtime is only $O(N^2)$

Dijkstra‘s Token Ring is not Uniform

- In this algorithm, the processes are not the same
- A stabilizing algorithm is called **uniform** if all processes are the same and have no identity
  - Uniform solutions are attractive, e.g. you can then equip all with the same "EPROM"
- Dijkstra showed in 1982 that there is no (uniform) solution to the token ring mutual exclusion problem if the size of the ring is not prime
  - A uniform solution for ring size = prime p (p is known to all processes) was then found by Burns & Pachl in 1988

Centralized Algorithm for Mutual Exclusion



a. Process 1 asks the coordinator for permission to access a shared resource (granted)
b. Process 2 then asks for permission to access the same resource; the coordinator does not answer.
c. When process 1 releases the resource, it informs the coordinator, who then answers the request of 2

Algorithm of Ricart und Agrawala

- Ricart and Agrawala have found in 1981 an analogous decentralized algorithm for this problem
- Idea:
  - Processes that want to enter the critical section (CS) send a message via multicasting (with a logical time stamp)
  - They can not enter the CS until all other processes have responded
- Every process
  - Has a Lamport clock
  - Sends messages of the form <Lamport timestamp, ID>
  - And is in one of the following states:
    - **RELEASED**: is outside the CS
    - **WANTED**: process wants to join the CS
    - **HELD**: is in the CS

Algorithm of Ricart und Agrawala

On initialization

state := RELEASED;

To enter the critical section

state := WANTED;
Multicast request to all processes;
T := request's timestamp;
Wait until (number of replies received = (N – 1));
state := HELD;

On receipt of a request $<T_i, p_i>$ at p_j $(i \neq j)$

if (state = HELD or (state = WANTED and $(T, p_j) < (T_i, p_i)$))
then
    queue request from p_i without replying;
else
    reply immediately to p_i;
end if

i.e. p_j has sent request before p_i

To exit the critical section

state := RELEASED;
reply to any queued requests;

Algorithmus – Evaluierung

- N potential points of failure
  - That is, *all* processes must work intact for the algorithm to work
- Much communication needed
- But shows that a fully distributed algorithm is possible for this problem
- An improved algorithm was developed by Maekawa in 1985
  - Use the observation that a process does not have to get permission from all other processes, but only from a subset of the processes of size approx. $\sqrt{N}$

Thank you.

Additional Slides

Arten von fehlertoleranten Algorithmen

Fehlertolerante Algorithmen

- In VS kann man korrektens Verhalten nicht so leicht garantieren wie in zentralisierten Systemen
  - Man kann aber die Eigenschaft des **partiellen Ausfalls** (**partial failure**) zum Positiven nutzen
  - „Make a bug to a feature“
- Es ist sehr unwahrscheinlich, dass alle Komponenten gleichzeitig ausfallen (Formel?)
  - Damit kann man den vollständigen Fehlerausfall mit dem Preis der eventuellen Degradation der Systemleistung verhindern
    - Degradation: geringere Verarbeitungskapazität oder System verhält sich für eine gewisse Zeit inkorrekt
  - Das ist in zentralisierten Systemen kaum möglich

Robuste vs. Selbststabilisierende Algorithmen

- **Robuste Algorithmen**
  - Garantieren korrektes Verhalten der nicht-ausgefallenen Prozesse trotzt der Ausfälle / Fehler anderer Prozesse
  - Zu keiner Zeit verhält sich das Gesamtsystem inkorrekt
  - Sie tolerieren permanente Fehler einer begrenzten Teilmenge der Komponenten
- **Selbststabilisierende Algorithmen**
  - Tolerieren ein zeitweilig inkorrektes Verhalten des Systems (transiente Fehler)
  - Mehrheit der Komponenten kann sich zeitweise inkorrekt Verhalten
  - Nach einer Weile nimmt das Gesamtsystem korrektes Verhalten an
    - Wie ein Stehaufmännchen

Robuste Algorithmen

- Robuste Algorithmen sind „pessimistisch“
  - jede Empfangene Information ist verdächtig
  - vor jedem Schritt wird eine Reihe von Checks durchgeführt, um die Korrektheit zu gewährleisten
- Robuste Algorithmen sind jederzeit korrekt
  - zu jeder Zeit der Ausführung wird Korrektheit garantiert
  - Preis dafür:
    - die Anzahl der erlaubten fehlerhaften Prozesse ist beschränkt
    - ggf. die Fehlermodelle schränken mögliche Fehlerarten ein

Entscheidungsprobleme

- Robusten Algorithmen versuchen meistens, **Entscheidungsprobleme** zu lösen
  - Jeder Prozess hat ein Input, und alle versuchen daraus zu einer Entscheidung zu kommen
  - Jeder korrekte Prozesse soll am Ende einen „Entscheidungswert“ in eine spezielle Variable schreiben
  - Dieser Entscheidungsprozess (z.B. Veto-Wahl) ist trivial ohne Fehler
  - Fungieren als Bausteine (Primitives) für andere Algorithmen
- Anforderungen für die Entscheidungen sind meistens:
  - **Terminierung** (termination)
  - **Konsistenz** (consistency)
  - **Nicht-Trivialität** (non-triviality)

Entscheidungsprobleme /2

- **Terminierung**:
  - Jeder korrekte Prozess muss schließlich entscheiden
- **Konsistenz**:
  - Eine Relation zwischen den Ergebnissen der Prozesse
  - Z.B. **Konsensproblem**: alle Entscheidungen korrekter Prozesse sind gleich
  - Z.B. **Electionproblem**: einer der korrekten Prozesse ist gewählt („leader“), alle anderen haben „verloren“
- **Nicht-Trivialität**:
  - Schliesst Lösungen aus, bei denen die Algorithmen gar nicht miteinander kommunizieren bzw. „fest verdrahtet“ sind
  - Z.B. bei Konsensproblem: verboten, dass alle Prozesse sofort „0“ als den Entscheidungswert schreiben

Fehlermodelle

Herbert! Du tot im Vorzimmer? So kenne ich Dich aber gar nicht!

- Hierarchie von Fehlerarten:
  - Initial Tote Prozesse (initially dead processes): Prozess führt keinen Schritt des lokalen Algorithmus aus
  - Crash (crash model): führt den lokalen Algorithmus für eine Weile korrekt aus, danach stoppt vollständig
  - Byzantinisches Verhalten (Byzantine behaviour): Prozess kann sich beliebig Verhalten (und beliebige Nachrichten aussenden)
- Merke: jede Art ist ein Spezialfall der nächsten Art
  - Damit ist ein byzantinisch-robuster Algorithmus auch Crash-robust etc.
  - Umgekehrt: *Nichtexistenz* eines Initial-Tot-robusten Algorithmus impliziert die Nichtexistenz der anderen beiden Arten

Beispiel: Konsensproblem

- Wir möchten einen Algorithmus haben, der einen (binären) Konsens erreicht und dabei <u>bis zu t Crashes</u> toleriert:
  - Terminierung: in jeder *t-crash fairen Ausführung* (d.h. wo mindestens N-t Prozesse unendlich viele Ereignisse ausführen) entscheiden schließlich alle korrekten Prozesse (d.h. setzen unwiderruflich ihren Entscheidungsregister auf einen Wert 0/1)
  - Konsistenz: Keine zwei korrekten Prozesse entscheiden sich für verschiedene Werte
  - Nicht-Trivialität: Es gibt eine Ausführung, bei der die Entscheidung 0 lautet und eine Ausführung, bei die Entscheidung 1 lautet

Unser Systemmodell

- <u>Asynchron</u>
  - Keine obere Schranke auf die Zeit der Ausführung eines Prozessorschritts
  - Keine obere Schranke auf die Zeit, die eine Nachricht braucht
  - Damit: keine synchronisierten Uhren möglich
- <u>Nachrichtenübermittlungsnetzwerk</u> (point-to-point)
- <u>Keine Übermittlungsfehler</u>
- Prozesse arbeiten entweder korrekt oder fallen total aus (**crash model**, kein byzantinisches Verhalten)

Bad News: Unmöglichkeit des Konsens

- Fisher, Lynch, Paterson (1985): Es gibt keinen deterministischen Konsensalgorithmus in diesem Systemmodell, sogar für t = 1.
  - Journal of the ACM, Vol. 32, No. 2, April 1985
- D.h. sogar unter recht starken Annahmen (keine Übermittlungsfehler, nur ein fehlerhafter Prozessor, keine Byzantinischen Fehler) ist eine „Abstimmung“ unmöglich!
  - Desto „mehr unmöglich“ für mehr schwerwiegende Fehlerarten
- Andererseits: kleine Änderungen des Systemmodells lassen Lösungen zu, z.B. Bracha-Toueg-Algorithmus:
  - Terminierung wird ersetzt durch „Konvergenz“:
  - Für jede Anfangskonfiguration:

$$\lim_{k \to \infty} \Pr[\text{ ein korrekter Prozess hat nach k Schritten noch nicht entschieden}] = 0$$